

Isabelle/HOLCF — Higher-Order Logic of Computable Functions

May 23, 2024

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 <code>cpo</code> and <code>pcpo</code>	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 pcpes	18
4	Admissibility and compactness	18
4.1	Definitions	19
4.2	Admissibility on chain-finite types	19
4.3	Admissibility of special formulae and propagation	19
4.4	Compactness	21
5	Subtypes of pcpes	22
5.1	Proving a subtype is a partial order	22
5.2	Proving a subtype is finite	23
5.3	Proving a subtype is chain-finite	23

5.4	Proving a subtype is complete	23
5.4.1	Continuity of <i>Rep</i> and <i>Abs</i>	25
5.5	Proving subtype elements are compact	25
5.6	Proving a subtype is pointed	26
5.6.1	Strictness of <i>Rep</i> and <i>Abs</i>	26
5.7	Proving a subtype is flat	27
5.8	HOLCF type definition package	27
6	Class instances for the full function space	27
6.1	Full function space is a partial order	28
6.2	Full function space is chain complete	28
6.3	Full function space is pointed	29
6.4	Propagation of monotonicity and continuity	29
7	The cpo of cartesian products	30
7.1	Unit type is a pcpo	30
7.2	Product type is a partial order	31
7.3	Monotonicity of <i>Pair</i> , <i>fst</i> , <i>snd</i>	31
7.4	Product type is a cpo	32
7.5	Product type is pointed	33
7.6	Continuity of <i>Pair</i> , <i>fst</i> , <i>snd</i>	34
7.7	Compactness and chain-finiteness	35
8	The type of continuous functions	36
8.1	Definition of continuous function type	36
8.2	Syntax for continuous lambda abstraction	37
8.3	Continuous function space is pointed	38
8.4	Basic properties of continuous functions	38
8.4.1	Beta-reduction simproc	38
8.5	Continuity of application	40
8.6	Continuity simplification procedure	41
8.7	Miscellaneous	42
8.8	Continuous injection-retraction pairs	43
8.9	Identity and composition	44
8.10	Strictified functions	44
8.11	Continuity of let-bindings	45
9	Continuous deflations and ep-pairs	46
9.1	Continuous deflations	46
9.2	Deflations with finite range	48
9.3	Continuous embedding-projection pairs	49
9.4	Uniqueness of ep-pairs	52
9.5	Composing ep-pairs	53

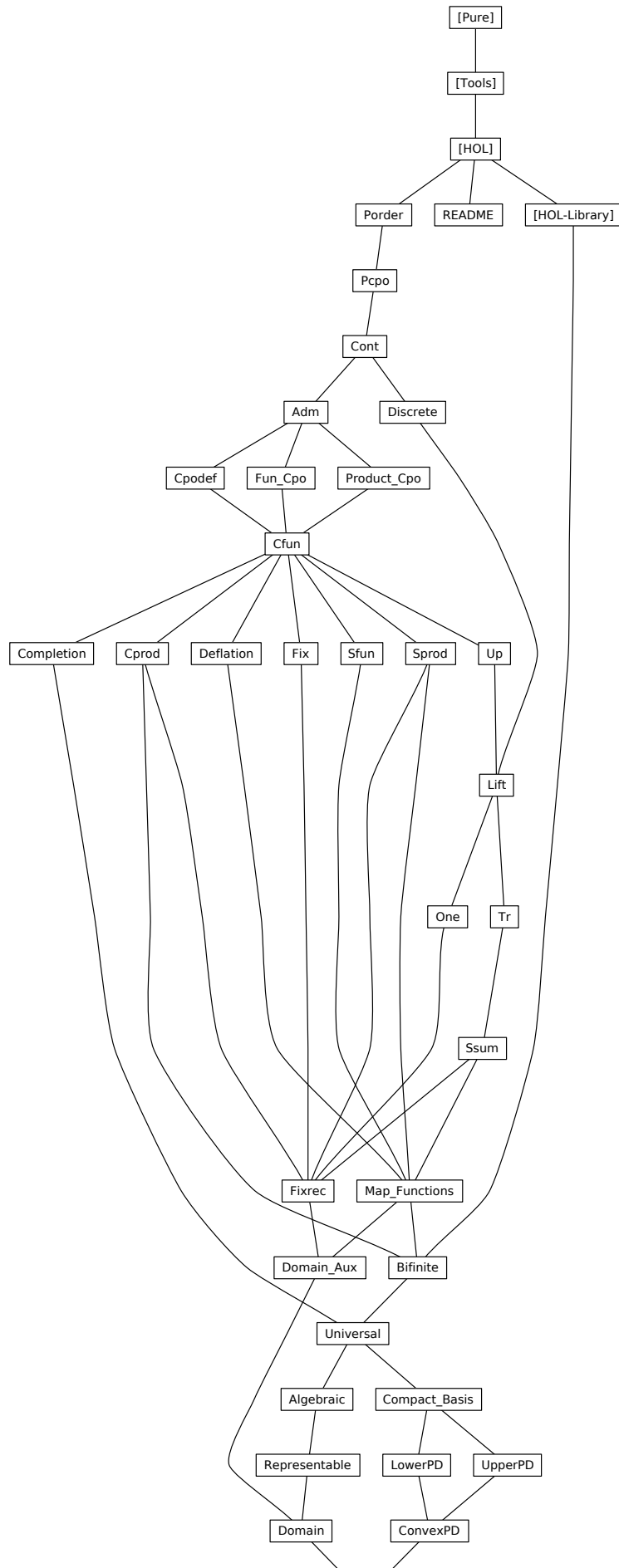
10 The type of strict products	54
10.1 Definition of strict product type	54
10.2 Definitions of constants	54
10.3 Case analysis	55
10.4 Properties of <i>spair</i>	55
10.5 Properties of <i>sfst</i> and <i>ssnd</i>	56
10.6 Compactness	57
10.7 Properties of <i>ssplit</i>	57
10.8 Strict product preserves flatness	58
11 Discrete cpo types	58
11.1 Discrete cpo class instance	58
11.2 <i>undiscr</i>	58
12 The type of lifted values	59
12.1 Definition of new type for lifting	59
12.2 Ordering on lifted cpo	59
12.3 Lifted cpo is a partial order	59
12.4 Lifted cpo is a cpo	60
12.5 Lifted cpo is pointed	61
12.6 Continuity of <i>Iup</i> and <i>Ifup</i>	61
12.7 Continuous versions of constants	62
13 Lifting types of class type to flat pcpo's	64
13.1 Lift as a datatype	64
13.2 Lift is flat	65
13.3 Continuity of <i>case-lift</i>	65
13.4 Further operations	65
14 The type of lifted booleans	66
14.1 Type definition and constructors	66
14.2 Case analysis	67
14.3 Boolean connectives	68
14.4 Rewriting of HOLCF operations to HOL functions	69
14.5 Compactness	69
15 The type of strict sums	70
15.1 Definition of strict sum type	70
15.2 Definitions of constructors	70
15.3 Properties of <i>sinl</i> and <i>sinr</i>	71
15.4 Case analysis	72
15.5 Case analysis combinator	73
15.6 Strict sum preserves flatness	73
16 The Strict Function Type	73

17 Map functions for various types	74
17.1 Map operator for continuous function space	75
17.2 Map operator for product type	77
17.3 Map function for lifted cpo	78
17.4 Map function for strict products	79
17.5 Map function for strict sums	81
17.6 Map operator for strict function space	83
18 The cpo of cartesian products	85
18.1 Continuous case function for unit type	85
18.2 Continuous version of split function	85
18.3 Convert all lemmas to the continuous versions	85
19 Profinite and bifinite cpos	85
19.1 Chains of finite deflations	86
19.2 Omega-profinite and bifinite domains	86
19.3 Building approx chains	86
19.4 Class instance proofs	88
20 Defining algebraic domains by ideal completion	91
20.1 Ideals over a preorder	91
20.2 Lemmas about least upper bounds	94
20.3 Locale for ideal completion	94
20.3.1 Principal ideals approximate all elements	95
20.4 Defining functions in terms of basis elements	97
21 A universal bifinite domain	100
21.1 Basis for universal domain	100
21.1.1 Basis datatype	100
21.1.2 Basis ordering	101
21.1.3 Generic take function	102
21.2 Defining the universal domain by ideal completion	103
21.3 Compact bases of domains	104
21.4 Universality of <i>udom</i>	105
21.4.1 Choosing a maximal element from a finite set	105
21.4.2 Compact basis take function	108
21.4.3 Rank of basis elements	109
21.4.4 Sequencing basis elements	110
21.4.5 Embedding and projection on basis elements	111
21.4.6 EP-pair from any bifinite domain into <i>udom</i>	117
21.5 Chain of approx functions for type <i>udom</i>	118

22 Algebraic deflations	120
22.1 Type constructor for finite deflations	120
22.2 Defining algebraic deflations by ideal completion	121
22.3 Applying algebraic deflations	123
22.4 Deflation combinators	124
23 Representable domains	125
23.1 Class of representable domains	126
23.2 Domains are bifinite	127
23.3 Universal domain ep-pairs	128
23.4 Type combinators	129
23.5 Class instance proofs	130
23.5.1 Universal domain	130
23.5.2 Lifted cpo	131
23.5.3 Strict function space	131
23.5.4 Continuous function space	132
23.5.5 Strict product	133
23.5.6 Cartesian product	134
23.5.7 Unit type	136
23.5.8 Discrete cpo	136
23.5.9 Strict sum	137
23.5.10 Lifted HOL type	138
24 The unit domain	139
25 Fixed point operator and admissibility	140
25.1 Iteration	140
25.2 Least fixed point operator	141
25.3 Fixed point induction	143
25.4 Fixed-points on product types	144
26 Package for defining recursive functions in HOLCF	145
26.1 Pattern-match monad	145
26.1.1 Run operator	146
26.1.2 Monad plus operator	146
26.2 Match functions for built-in types	147
26.3 Mutual recursion	149
26.4 Initializing the fixrec package	149
27 Domain package support	150
27.1 Continuous isomorphisms	150
27.2 Proofs about take functions	152
27.3 Finiteness	153
27.4 Proofs about constructor functions	155

27.5 ML setup	157
28 Domain package	157
28.1 Representations of types	157
28.2 Deflations as sets	158
28.3 Proving a subtype is representable	158
28.4 Isomorphic deflations	160
28.5 Setting up the domain package	163
29 A compact basis for powerdomains	164
29.1 A compact basis for powerdomains	164
29.2 Unit and plus constructors	165
29.3 Fold operator	166
30 Upper powerdomain	166
30.1 Basis preorder	166
30.2 Type definition	168
30.3 Monadic unit and plus	168
30.4 Induction rules	171
30.5 Monadic bind	172
30.6 Map	173
30.7 Upper powerdomain is bifinite	175
30.8 Join	176
31 Lower powerdomain	176
31.1 Basis preorder	176
31.2 Type definition	178
31.3 Monadic unit and plus	179
31.4 Induction rules	182
31.5 Monadic bind	182
31.6 Map	184
31.7 Lower powerdomain is bifinite	186
31.8 Join	186
32 Convex powerdomain	187
32.1 Basis preorder	187
32.2 Type definition	189
32.3 Monadic unit and plus	190
32.4 Induction rules	192
32.5 Monadic bind	193
32.6 Map	194
32.7 Convex powerdomain is bifinite	196
32.8 Join	196
32.9 Conversions to other powerdomains	197

33 Powerdomains	199
33.1 Universal domain embeddings	199
33.2 Deflation combinators	200
33.3 Domain class instances	200
33.4 Isomorphic deflations	202
33.5 Domain package setup for powerdomains	203



1 Partial orders

```
theory Porder
  imports Main
begin
```

```
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 ⊑ z ⇒ x ⊑ y ⇒ x ⊑ z
```

by (rule below-trans)

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

end

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

context po
begin

1.2 Upper bounds

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

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

lemma is-ubD: $\llbracket S <| u; x \in S \rrbracket \implies x \sqsubseteq u$
by (simp add: is-ub-def)

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

lemma ub-imageD: $\llbracket f ' S <| u; x \in S \rrbracket \implies f x \sqsubseteq u$
unfolding is-ub-def by fast

lemma ub-rangeI: $(\bigwedge i. S i \sqsubseteq x) \implies \text{range } S <| x$
unfolding is-ub-def by fast

lemma ub-rangeD: $\text{range } S <| x \implies S i \sqsubseteq x$
unfolding is-ub-def by fast

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

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

lemma is-ub-upward: $\llbracket S <| x; x \sqsubseteq y \rrbracket \implies S <| y$
unfolding is-ub-def by (fast intro: below-trans)

1.3 Least upper bounds

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

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

end

syntax (*ASCII*)

-*BLub* :: [pttrn, 'a set, 'b] \Rightarrow 'b ((*3LUB* -:/ -) [0,0, 10] 10)

syntax

-*BLub* :: [pttrn, 'a set, 'b] \Rightarrow 'b ((*3* \sqcup - \in -./ -) [0,0, 10] 10)

translations

LUB $x:A. t \Rightarrow CONST\ lub\ ((\lambda x. t)\ 'A)$

context *po*

begin

abbreviation *Lub* (**binder** \sqcup 10)

where $\sqcup n. t\ n \equiv lub\ (range\ t)$

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*: $\llbracket S <<| x; S <| u \rrbracket \Longrightarrow x \sqsubseteq u$

unfolding *is-lub-def* **by** *fast*

lemma *is-lubI*: $\llbracket S <| x; \bigwedge u. S <| u \rrbracket \Longrightarrow x \sqsubseteq u \rrbracket \Longrightarrow S <<| x$

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*: $M \ll\mid l \implies \text{lub } M = l$
by (*rule is-lub-unique* [*OF is-lub-lub*])

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

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

lemma *is-lub-bin*: $x \sqsubseteq y \implies \{x, y\} \ll\mid y$
by (*simp add: is-lub-def*)

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

lemma *is-lub-maximal*: $S \ll\mid x \implies x \in S \implies S \ll\mid x$
by (*erule is-lubI, erule (1) is-ubD*)

lemma *lub-maximal*: $S \ll\mid x \implies x \in S \implies \text{lub } S = x$
by (*rule is-lub-maximal* [*THEN lub-eqI*])

1.4 Countable chains

definition *chain* :: $(\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$

where — Here we use countable chains and I prefer to code them as functions!

chain $Y = (\forall i. Y\ i \sqsubseteq Y\ (\text{Suc } i))$

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

lemma *chainE*: $\text{chain } Y \implies Y\ i \sqsubseteq Y\ (\text{Suc } i)$
unfolding *chain-def* **by** *fast*

chains are monotone functions

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

lemma *chain-mono*: $\text{chain } Y \implies i \leq j \implies Y\ i \sqsubseteq Y\ j$
by (*cases i = j*) (*simp-all add: chain-mono-less*)

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

technical lemmas about (least) upper bounds of chains

lemma *is-lub-rangeD1*: $\text{range } S \ll\mid x \implies S\ i \sqsubseteq x$
by (*rule is-lubD1* [*THEN ub-rangeD*])

lemma *is-ub-range-shift*: $\text{chain } S \implies \text{range } (\lambda i. S\ (i + j)) \ll\mid x = \text{range } S \ll\mid x$
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*: $\text{chain } S \implies \text{range } (\lambda i. S (i + j)) \ll\ll x = \text{range } S \ll\ll x$
by (*simp add: is-lub-def is-ub-range-shift*)

the lub of a constant chain is the constant

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

lemma *is-lub-const*: $\text{range } (\lambda x. c) \ll\ll c$
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* :: $\text{nat} \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$
where — finite chains, needed for monotony of continuous functions
max-in-chain i $C \iff (\forall j. i \leq j \longrightarrow C i = C j)$

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

results about finite chains

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

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

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

lemma *finite-chainE*: $\llbracket \text{finite-chain } C; \bigwedge i. \llbracket \text{chain } C; \text{max-in-chain } i C \rrbracket \implies R \rrbracket$
 $\implies R$
unfolding *finite-chain-def* **by** *fast*

lemma *lub-finch1*: $\text{chain } C \implies \text{max-in-chain } i C \implies \text{range } C \ll\ll C i$
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*: $\text{finite-chain } C \implies \text{range } C \ll\ll C$ (LEAST i . $\text{max-in-chain } i$ C)

```

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

```

lemma *finch-imp-finite-range*: $\text{finite-chain } Y \implies \text{finite } (\text{range } Y)$

```

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

```

lemma *finite-range-has-max*:

```

fixes f :: nat  $\Rightarrow$  'a
and r :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool
assumes mono:  $\bigwedge i j. i \leq j \implies r (f i) (f j)$ 
assumes finite-range:  $\text{finite } (\text{range } f)$ 
shows  $\exists k. \forall i. r (f i) (f k)$ 
proof (intro exI allI)
fix i :: nat
let ?j = LEAST k. f k = f i
let ?k = Max (( $\lambda x$ . LEAST k. f k = x) ‘range f)
have ?j  $\leq$  ?k
proof (rule Max-ge)
show  $\text{finite } ((\lambda x. \text{LEAST } k. f k = x) \text{ ‘range } f)$ 
using finite-range by (rule finite-imageI)
show ?j  $\in$  ( $\lambda x. \text{LEAST } k. f k = x$ ) ‘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:  $chain\ Y \implies finite\ (range\ Y) \implies finite-chain\ Y$ 
apply (subgoal-tac  $\exists k. \forall i. Y\ i \sqsubseteq Y\ k$ )
apply (erule exE)
apply (rule finite-chainI, assumption)
apply (rule max-in-chainI)
apply (rule below-antisym)
apply (erule (1) chain-mono)
apply (erule spec)
apply (rule finite-range-has-max)
apply (erule (1) chain-mono)
apply assumption
done

```

```

lemma bin-chain:  $x \sqsubseteq y \implies chain\ (\lambda i. if\ i=0\ then\ x\ else\ y)$ 
by (rule chainI) simp

```

```

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

```

```

lemma is-lub-bin-chain:  $x \sqsubseteq y \implies range\ (\lambda i::nat. if\ i=0\ then\ x\ else\ y) \lll\ y$ 
apply (frule bin-chain)
apply (drule bin-chainmax)
apply (drule (1) lub-finchI)
apply simp
done

```

the maximal element in a chain is its lub

```

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

```

end

end

2 Classes cpo and pcpo

```

theory Pcpo
imports Porder
begin

```

2.1 Complete partial orders

The class cpo of chain complete partial orders

```

class cpo = po +
assumes cpo:  $chain\ S \implies \exists x. range\ S \lll\ x$ 
begin

```

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

lemma *cpo-lubI*: $\text{chain } S \implies \text{range } S \ll\mid (\bigsqcup i. S\ i)$
by (*fast dest: cpo elim: is-lub-lub*)

lemma *thelubE*: $\llbracket \text{chain } S; (\bigsqcup i. S\ i) = l \rrbracket \implies \text{range } S \ll\mid l$
by (*blast dest: cpo intro: is-lub-lub*)

Properties of the lub

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

lemma *is-lub-thelub*: $\llbracket \text{chain } S; \text{range } S \ll\mid x \rrbracket \implies (\bigsqcup i. S\ i) \sqsubseteq x$
by (*blast dest: cpo intro: is-lub-lub [THEN is-lubD2]*)

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

lemma *lub-below*: $\llbracket \text{chain } S; \bigwedge i. S\ i \sqsubseteq x \rrbracket \implies (\bigsqcup i. S\ i) \sqsubseteq x$
by (*simp add: lub-below-iff*)

lemma *below-lub*: $\llbracket \text{chain } S; x \sqsubseteq S\ i \rrbracket \implies x \sqsubseteq (\bigsqcup i. S\ i)$
by (*erule below-trans, erule is-ub-thelub*)

lemma *lub-range-mono*: $\llbracket \text{range } X \subseteq \text{range } Y; \text{chain } Y; \text{chain } X \rrbracket \implies (\bigsqcup i. X\ i) \sqsubseteq (\bigsqcup i. Y\ i)$
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*: $\text{chain } Y \implies (\bigsqcup i. Y\ (i + j)) = (\bigsqcup i. Y\ i)$
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*: $\text{chain } Y \implies \text{max-in-chain } i\ Y = ((\bigsqcup i. Y\ i) = Y\ i)$
apply (*rule iffI*)
apply (*fast intro!: lub-eqI lub-finch1*)
apply (*unfold max-in-chain-def*)


```

apply (safe intro!: below-antisym)
apply (fast elim!: chain-mono)
apply (drule sym)
apply (force elim!: is-ub-thelub)
done

```

the \sqsubseteq relation between two chains is preserved by their lubs

```

lemma lub-mono:  $\llbracket \text{chain } X; \text{chain } Y; \bigwedge i. X\ i \sqsubseteq Y\ i \rrbracket \implies (\bigsqcup i. X\ i) \sqsubseteq (\bigsqcup i. Y\ i)$ 
by (fast elim: lub-below below-lub)

```

the $=$ relation between two chains is preserved by their lubs

```

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

```

lemma *ch2ch-lub*:

```

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

```

lemma *diag-lub*:

```

assumes 1:  $\bigwedge j. \text{chain } (\lambda i. Y\ i\ j)$ 
assumes 2:  $\bigwedge i. \text{chain } (\lambda j. Y\ i\ j)$ 
shows  $(\bigsqcup i. \bigsqcup j. Y\ i\ j) = (\bigsqcup i. Y\ i\ i)$ 
proof (rule below-antisym)
have 3:  $\text{chain } (\lambda i. Y\ i\ i)$ 
apply (rule chainI)
apply (rule below-trans)
apply (rule chainE [OF 1])
apply (rule chainE [OF 2])
done
have 4:  $\text{chain } (\lambda i. \bigsqcup j. Y\ i\ j)$ 
by (rule ch2ch-lub [OF 1 2])
show  $(\bigsqcup i. \bigsqcup j. Y\ i\ j) \sqsubseteq (\bigsqcup i. Y\ i\ i)$ 
apply (rule lub-below [OF 4])
apply (rule lub-below [OF 2])
apply (rule below-lub [OF 3])
apply (rule below-trans)
apply (rule chain-mono [OF 1 max.cobounded1])
apply (rule chain-mono [OF 2 max.cobounded2])
done
show  $(\bigsqcup i. Y\ i\ i) \sqsubseteq (\bigsqcup i. \bigsqcup j. Y\ i\ j)$ 
apply (rule lub-mono [OF 3 4])
apply (rule is-ub-thelub [OF 2])
done

```

qed

```

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

```

end

2.2 Pointed cpos

The class pcpo of pointed cpos

```

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

```

```

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

```

```

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

```

end

Old "UU" syntax:

```

syntax UU :: logic
translations UU  $\rightarrow$  CONST bottom

```

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

```

setup  $\langle$ Reorient-Proc.add (fn Const-bottom -> => true | - => false) $\rangle$ 
simproc-setup reorient-bottom  $(\perp = x) = \langle$ K Reorient-Proc.proc $\rangle$ 

```

useful lemmas about \perp

```

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

```

```

lemma eq-bottom-iff:  $x = \perp \longleftrightarrow x \sqsubseteq \perp$ 
  by simp

```

```

lemma bottomI:  $x \sqsubseteq \perp \implies x = \perp$ 
  by (subst eq-bottom-iff)

```

lemma *lub-eq-bottom-iff*: $\text{chain } Y \implies (\bigsqcup i. Y\ i) = \perp \iff (\forall i. Y\ i = \perp)$
 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*: $\text{chain } Y \implies \exists n. \text{max-in-chain } n\ Y$
begin

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

lemma *chfin2finch*: $\text{chain } Y \implies \text{finite-chain } Y$
 by (*simp add: chfin finite-chain-def*)

end

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

subclass *chfin*
proof
fix *Y*
assume *: $\text{chain } Y$
show $\exists n. \text{max-in-chain } n\ Y$
apply (*unfold max-in-chain-def*)
apply (*cases* $\forall i. Y\ i = \perp$)
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*: $x \sqsubseteq y \iff x = \perp \vee x = y$
 by (*safe dest!: ax-flat*)

lemma *flat-eq*: $a \neq \perp \implies a \sqsubseteq b = (a = b)$
 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  $\Rightarrow$  '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  $\exists i. \text{max-in-chain } i\ S$  ..
qed

end

end

```

3 Continuity and monotonicity

```

theory Cont
  imports Pcpo
begin

```

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

```

default-sort po

```

3.1 Definitions

```

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

```

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

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

lemma $contE: cont\ f \Longrightarrow chain\ Y \Longrightarrow range\ (\lambda i. f\ (Y\ i)) \ll\ f\ (\bigsqcup i. Y\ i)$
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 $ub2ub-monofun: monofun\ f \Longrightarrow range\ Y \ll\ u \Longrightarrow range\ (\lambda i. f\ (Y\ i)) \ll\ f\ u$
apply (*rule ub-rangeI*)
apply (*erule monofunE*)
apply (*erule ub-rangeD*)
done

a lemma about binary chains

lemma $binchain-cont: cont\ f \Longrightarrow x \sqsubseteq y \Longrightarrow range\ (\lambda i::nat. f\ (if\ i = 0\ then\ x\ else\ y)) \ll\ f\ y$
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  $\implies$  chain Y  $\implies$  f ( $\sqcup$  i. Y i) = ( $\sqcup$  i. f (Y i))
apply (rule lub-eqI [symmetric])
apply (erule (1) contE)
done

```

```

lemma contI2:

```

```

  fixes f :: 'a::cpo  $\Rightarrow$  'b::cpo
  assumes mono: monofun f
  assumes below:  $\bigwedge$  Y.  $\llbracket$ chain Y; chain ( $\lambda$ i. f (Y i)) $\rrbracket \implies$  f ( $\sqcup$  i. Y i)  $\sqsubseteq$  ( $\sqcup$  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 ( $\sqcup$  i. f (Y i)) = f ( $\sqcup$  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))  $\lll$  f ( $\sqcup$  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 $f :: 'a::cpo \Rightarrow 'b::cpo \Rightarrow 'c::cpo$ **and** $t :: 'a \Rightarrow 'b$

assumes 1: *cont* ($\lambda x. t x$)

assumes 2: $\bigwedge x. \text{cont } (\lambda y. f x y)$

assumes 3: $\bigwedge y. \text{cont } (\lambda x. f x y)$

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 $Y :: \text{nat} \Rightarrow 'a$

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* $c \implies \text{cont } (\lambda x. f x) \implies \text{cont } (\lambda x. c (f x))$
by (*rule cont-apply* [*OF - - cont-const*])

Least upper bounds preserve continuity

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

assumes *chain*: $\bigwedge x. \text{chain } (\lambda i. F i x)$

and *cont*: $\bigwedge i. \text{cont } (\lambda x. F i x)$

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* $f \implies \text{cont } g \implies \text{cont } (\lambda x. \text{if } b \text{ then } f x \text{ else } g x)$

by (*induct b*) *simp-all*

3.5 Finite chains and flat pcpos

Monotone functions map finite chains to finite chains.

lemma *monofun-finch2finch*: $\text{monofun } f \implies \text{finite-chain } Y \implies \text{finite-chain } (\lambda n. f (Y n))$
by (*force simp add: finite-chain-def ch2ch-monofun max-in-chain-def*)

The same holds for continuous functions.

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

All monotone functions with chain-finite domain are continuous.

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

All strict functions with flat domain are continuous.

lemma *flatdom-strict2mono*: $f \perp = \perp \implies \text{monofun } f$
for $f :: 'a::\text{flat} \Rightarrow 'b::\text{pcpo}$
apply (*rule monofunI*)
apply (*drule ax-flat*)
apply *auto*
done

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

All functions with discrete domain are continuous.

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

end

4 Admissibility and compactness

theory *Adm*

imports *Cont*
begin

default-sort *cpo*

4.1 Definitions

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

lemma *admI*: ($\bigwedge Y. \llbracket \text{chain } Y; \forall i. P (Y i) \rrbracket \Longrightarrow P (\bigsqcup i. Y i) \Longrightarrow \text{adm } P$)
unfolding *adm-def* **by** *fast*

lemma *admD*: *adm* *P* $\Longrightarrow \text{chain } Y \Longrightarrow (\bigwedge i. P (Y i)) \Longrightarrow P (\bigsqcup i. Y i)$
unfolding *adm-def* **by** *fast*

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

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

4.2 Admissibility on chain-finite types

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

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

4.3 Admissibility of special formulae and propagation

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

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

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

lemma *adm-ball* [*simp*]: ($\bigwedge y. y \in A \Longrightarrow \text{adm } (\lambda x. P x y)$) $\Longrightarrow \text{adm } (\lambda x. \forall y \in A. P x y)$
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  $adm-disj-lemma2: \forall n::nat. P n \vee Q n \implies (\forall i. \exists j \geq i. P j) \vee (\forall i. \exists j \geq i. Q j)$ 
  apply ( $erule\ contrapos-pp$ )
  apply ( $clarsimp, rename-tac\ a\ b$ )
  apply ( $rule-tac\ x=max\ a\ b\ in\ exI$ )
  apply  $simp$ 
done

```

```

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

```

```

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

```

admissibility and continuity

lemma *adm-below* [*simp*]: $\text{cont } (\lambda x. u x) \Longrightarrow \text{cont } (\lambda x. v x) \Longrightarrow \text{adm } (\lambda x. u x \sqsubseteq v x)$

by (*simp add: adm-def cont2contlubE lub-mono ch2ch-cont*)

lemma *adm-eq* [*simp*]: $\text{cont } (\lambda x. u x) \Longrightarrow \text{cont } (\lambda x. v x) \Longrightarrow \text{adm } (\lambda x. u x = v x)$

by (*simp add: po-eq-conv*)

lemma *adm-subst*: $\text{cont } (\lambda x. t x) \Longrightarrow \text{adm } P \Longrightarrow \text{adm } (\lambda x. P (t x))$

by (*simp add: adm-def cont2contlubE ch2ch-cont*)

lemma *adm-not-below* [*simp*]: $\text{cont } (\lambda x. t x) \Longrightarrow \text{adm } (\lambda x. t x \not\sqsubseteq u)$

by (*rule admI*) (*simp add: cont2contlubE ch2ch-cont lub-below-iff*)

4.4 Compactness

definition *compact* :: 'a::cpo \Rightarrow bool

where *compact* *k* = $\text{adm } (\lambda x. k \not\sqsubseteq x)$

lemma *compactI*: $\text{adm } (\lambda x. k \not\sqsubseteq x) \Longrightarrow \text{compact } k$

unfolding *compact-def* .

lemma *compactD*: $\text{compact } k \Longrightarrow \text{adm } (\lambda x. k \not\sqsubseteq x)$

unfolding *compact-def* .

lemma *compactI2*: $(\bigwedge Y. \llbracket \text{chain } Y; x \sqsubseteq (\bigsqcup i. Y i) \rrbracket \Longrightarrow \exists i. x \sqsubseteq Y i) \Longrightarrow \text{compact } x$

unfolding *compact-def adm-def* **by** *fast*

lemma *compactD2*: $\text{compact } x \Longrightarrow \text{chain } Y \Longrightarrow x \sqsubseteq (\bigsqcup i. Y i) \Longrightarrow \exists i. x \sqsubseteq Y i$

unfolding *compact-def adm-def* **by** *fast*

lemma *compact-below-lub-iff*: $\text{compact } x \Longrightarrow \text{chain } Y \Longrightarrow x \sqsubseteq (\bigsqcup i. Y i) \longleftrightarrow (\exists i. x \sqsubseteq Y i)$

by (*fast intro: compactD2 elim: below-lub*)

lemma *compact-chfin* [*simp*]: $\text{compact } x$

for $x :: 'a::\text{chfin}$

by (*rule compactI [OF adm-chfin]*)

lemma *compact-imp-max-in-chain*: $\text{chain } Y \Longrightarrow \text{compact } (\bigsqcup i. Y i) \Longrightarrow \exists i. \text{max-in-chain } i Y$

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-the lub]*)

done

admissibility and compactness

lemma *adm-compact-not-below* [*simp*]:
 $compact\ k \implies cont\ (\lambda x. t\ x) \implies adm\ (\lambda x. k \not\sqsubseteq t\ x)$
unfolding *compact-def* **by** (*rule adm-subst*)

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

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

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

Any upward-closed predicate is admissible.

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

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

end

5 Subtypes of pcpo

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

5.1 Proving a subtype is a partial order

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

setup $\langle Sign.add-const-constraint\ (\mathbf{const-name}\ \langle Porder.below \rangle, NONE) \rangle$

theorem *typedef-po*:
fixes $Abs :: 'a::po \Rightarrow 'b::type$
assumes $type: type-definition\ Rep\ Abs\ A$
and $below: (\sqsubseteq) \equiv \lambda x\ y. Rep\ x \sqsubseteq Rep\ y$
shows $OFCLASS('b, po-class)$
apply (*intro-classes, unfold below*)
apply (*rule below-refl*)

```

apply (erule (1) below-trans)
apply (rule type-definition.Rep-inject [OF type, THEN iffD1])
apply (erule (1) below-antisym)
done

```

```

setup ⟨Sign.add-const-constraint (const-name ⟨Porder.below⟩, SOME typ ⟨'a::below
⇒ 'a::below ⇒ bool⟩)⟩

```

5.2 Proving a subtype is finite

```

lemma typedef-finite-UNIV:
  fixes Abs :: 'a::type ⇒ 'b::type
  assumes type: type-definition Rep Abs A
  shows finite A ⇒ finite (UNIV :: 'b set)
proof –
  assume finite A
  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

```

5.3 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

```

5.4 Proving a subtype is complete

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

```

lemma typedef-is-lubI:
  assumes below: (⊆) ≡ 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 \Rightarrow 'b::po

assumes *type*: type-definition *Rep Abs A*

and *below*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$

and *adm*: adm $(\lambda x. x \in A)$

shows $\text{chain } S \Longrightarrow \text{Rep } (\text{Abs } (\bigsqcup i. \text{Rep } (S i))) = (\bigsqcup i. \text{Rep } (S i))$

apply (rule type-definition.Abs-inverse [OF type])

apply (erule admD [OF adm ch2ch-Rep [OF below]])

apply (rule type-definition.Rep [OF type])

done

theorem *typedef-is-lub*:

fixes *Abs* :: 'a::cpo \Rightarrow 'b::po

assumes *type*: type-definition *Rep Abs A*

and *below*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$

and *adm*: adm $(\lambda x. x \in A)$

assumes *S*: chain *S*

shows $\text{range } S \ll\!| \text{Abs } (\bigsqcup i. \text{Rep } (S i))$

proof –

from *S* have chain $(\lambda i. \text{Rep } (S i))$

by (rule ch2ch-Rep [OF below])

then have $\text{range } (\lambda i. \text{Rep } (S i)) \ll\!| (\bigsqcup i. \text{Rep } (S i))$

by (rule cpo-lubI)

then have $\text{range } (\lambda i. \text{Rep } (S i)) \ll\!| \text{Rep } (\text{Abs } (\bigsqcup i. \text{Rep } (S i)))$

by (simp only: Abs-inverse-lub-Rep [OF type below adm S])

then show $\text{range } S \ll\!| \text{Abs } (\bigsqcup i. \text{Rep } (S i))$

by (rule typedef-is-lubI [OF below])

qed

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

theorem *typedef-cpo*:

fixes *Abs* :: 'a::cpo \Rightarrow 'b::po

assumes *type*: type-definition *Rep Abs A*

and *below*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$

and *adm*: adm $(\lambda x. x \in A)$

shows OFCLASS('b, cpo-class)

proof

fix *S* :: nat \Rightarrow 'b

assume chain *S*

then have $\text{range } S \ll\!| \text{Abs } (\bigsqcup i. \text{Rep } (S i))$

by (rule typedef-is-lub [OF type below adm])

then show $\exists x. \text{range } S \ll\!| x ..$

qed

5.4.1 Continuity of *Rep* and *Abs*

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

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

5.5 Proving subtype elements are compact

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

5.6 Proving a subtype is pointed

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

theorem *typedef-pcpo-generic*:
fixes *Abs* :: 'a::cpo \Rightarrow 'b::cpo
assumes *type*: *type-definition* *Rep* *Abs* *A*
and *below*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
and *z-in-A*: $z \in A$
and *z-least*: $\bigwedge x. x \in A \implies z \sqsubseteq x$
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 \perp .

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

5.6.1 Strictness of *Rep* and *Abs*

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

theorem *typedef-Abs-strict*:
assumes *type*: *type-definition* *Rep* *Abs* *A*
and *below*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
and *bottom-in-A*: $\perp \in A$
shows *Abs* $\perp = \perp$
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*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
and *bottom-in-A*: $\perp \in A$
shows *Rep* $\perp = \perp$
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*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$

and *bottom-in-A*: $\perp \in A$

shows $x \in A \implies (\text{Abs } x = \perp) = (x = \perp)$

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*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$

and *bottom-in-A*: $\perp \in A$

shows $(\text{Rep } x = \perp) = (x = \perp)$

apply (*rule typedef-Rep-strict* [*OF type below bottom-in-A, THEN subst*])

apply (*simp add: type-definition.Rep-inject* [*OF type*])

done

5.7 Proving a subtype is flat

theorem *typedef-flat*:

fixes *Abs* :: '*a*::flat \Rightarrow '*b*::pcpo

assumes *type*: *type-definition Rep Abs A*

and *below*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$

and *bottom-in-A*: $\perp \in A$

shows *OFCLASS*('b, *flat-class*)

apply (*intro-classes*)

apply (*unfold below*)

apply (*simp add: type-definition.Rep-inject* [*OF type, symmetric*])

apply (*simp add: typedef-Rep-strict* [*OF type below bottom-in-A*])

apply (*simp add: ax-flat*)

done

5.8 HOLCF type definition package

ML-file $\langle \text{Tools}/\text{cpodef}.ML \rangle$

end

6 Class instances for the full function space

theory *Fun-Cpo*

imports *Adm*

begin

6.1 Full function space is a partial order

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

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

instance ..
end

instance *fun* :: (*type*, *po*) *po*

proof

fix *f* :: 'a \Rightarrow 'b

show $f \sqsubseteq f$

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

next

fix *f g* :: 'a \Rightarrow 'b

assume $f \sqsubseteq g$ **and** $g \sqsubseteq f$ **then show** $f = g$

by (*simp add: below-fun-def fun-eq-iff below-antisym*)

next

fix *f g h* :: 'a \Rightarrow 'b

assume $f \sqsubseteq g$ **and** $g \sqsubseteq h$ **then show** $f \sqsubseteq h$

unfolding *below-fun-def* **by** (*fast elim: below-trans*)

qed

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

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

lemma *fun-belowI*: $(\bigwedge x. f x \sqsubseteq g x) \Longrightarrow f \sqsubseteq g$

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

lemma *fun-belowD*: $f \sqsubseteq g \Longrightarrow f x \sqsubseteq g x$

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

6.2 Full function space is chain complete

Properties of chains of functions.

lemma *fun-chain-iff*: $\text{chain } S \longleftrightarrow (\forall x. \text{chain } (\lambda i. S i x))$

by (*auto simp: chain-def fun-below-iff*)

lemma *ch2ch-fun*: $\text{chain } S \Longrightarrow \text{chain } (\lambda i. S i x)$

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

lemma *ch2ch-lambda*: $(\bigwedge x. \text{chain } (\lambda i. S i x)) \Longrightarrow \text{chain } S$

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

Type 'a \Rightarrow 'b is chain complete

lemma *is-lub-lambda*: $(\bigwedge x. \text{range } (\lambda i. Y i x) \ll\lvert f x) \Longrightarrow \text{range } Y \ll\lvert f$

by (*simp add: is-lub-def is-ub-def below-fun-def*)

```

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

```

```

lemma lub-fun: chain S  $\implies$  ( $\bigsqcup i. S\ i$ ) = ( $\lambda x. \bigsqcup i. S\ i\ x$ )
  for S :: nat  $\Rightarrow$  'a::type  $\Rightarrow$  'b::cpo
  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  $f \sqsubseteq g \iff f = g$ 
    by (simp add: fun-below-iff fun-eq-iff)
qed

```

6.3 Full function space is pointed

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

6.4 Propagation of monotonicity and continuity

The lub of a chain of monotone functions is monotone.

```

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

```

The lub of a chain of continuous functions is continuous.

```

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

```

Function application preserves monotonicity and continuity.

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

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

lemma *cont-fun*: $\text{cont } (\lambda f. f x)$
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*: $(\bigwedge y. \text{monofun } (\lambda x. f x y)) \implies \text{monofun } f$
by (*simp add: monofun-def fun-below-iff*)

lemma *cont2cont-lambda* [*simp*]:
assumes $f: \bigwedge y. \text{cont } (\lambda x. f x y)$
shows $\text{cont } f$
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*: $(\bigwedge x. \text{chain } (\lambda i. S i x)) \implies (\lambda x. \bigsqcup i. S i x) = (\bigsqcup i. (\lambda x. S i x))$
for $S :: \text{nat} \Rightarrow 'a::\text{type} \Rightarrow 'b::\text{cpo}$
by (*simp add: lub-fun ch2ch-lambda*)

end

7 The cpo of cartesian products

theory *Product-Cpo*
imports *Adm*
begin

default-sort *cpo*

7.1 Unit type is a pcpo

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

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

instance
by *standard simp*

end

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

7.2 Product type is a partial order

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

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

instance ..

end

instance *prod* :: (*po*, *po*) *po*

proof

fix *x* :: 'a × 'b

show $x \sqsubseteq x$

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

next

fix *x y* :: 'a × 'b

assume $x \sqsubseteq y\ y \sqsubseteq x$

then show $x = y$

unfolding *below-prod-def prod-eq-iff*

by (*fast intro: below-antisym*)

next

fix *x y z* :: 'a × 'b

assume $x \sqsubseteq y\ y \sqsubseteq z$

then show $x \sqsubseteq z$

unfolding *below-prod-def*

by (*fast intro: below-trans*)

qed

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

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

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

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

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

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

fst and *snd* are monotone

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

lemma *snd-monofun*: $x \sqsubseteq y \implies snd\ x \sqsubseteq snd\ y$
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

7.4 Product type is a cpo

lemma *is-lub-Pair*: $range\ A \ll\!| x \implies range\ B \ll\!| y \implies range\ (\lambda i. (A\ i, B\ i)) \ll\!| (x, y)$
by (*simp add: is-lub-def is-ub-def below-prod-def*)

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

lemma *is-lub-prod*:

```

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

```

```

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

```

```

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

```

```

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

```

7.5 Product type is pointed

```

lemma  $\text{minimal-prod: } (\perp, \perp) \sqsubseteq p$ 
by  $(\text{simp add: below-prod-def})$ 

```

```

instance  $\text{prod} :: (\text{pcpo}, \text{pcpo}) \text{ pcpo}$ 
by  $\text{intro-classes (fast intro: minimal-prod)}$ 

```

```

lemma  $\text{inst-prod-pcpo: } \perp = (\perp, \perp)$ 
by  $(\text{rule minimal-prod [THEN bottomI, symmetric]})$ 

```

```

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

```

```

lemma  $\text{fst-strict [simp]: } \text{fst } \perp = \perp$ 
unfolding  $\text{inst-prod-pcpo by (rule fst-conv)}$ 

```

```

lemma  $\text{snd-strict [simp]: } \text{snd } \perp = \perp$ 
unfolding  $\text{inst-prod-pcpo by (rule snd-conv)}$ 

```

```

lemma  $\text{Pair-strict [simp]: } (\perp, \perp) = \perp$ 
by  $\text{simp}$ 

```

```

lemma  $\text{split-strict [simp]: } \text{case-prod } f\ \perp = f\ \perp\ \perp$ 

```

by (*simp add: split-def*)

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

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

lemma *cont-pair2*: *cont* ($\lambda y. (x, y)$)
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* ($\lambda x. f x$)
assumes *g*: *cont* ($\lambda x. g x$)
shows *cont* ($\lambda x. (f x, g x)$)
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*: $\bigwedge a b. \text{cont } (\lambda x. f x a b)$
assumes *f2*: $\bigwedge x b. \text{cont } (\lambda a. f x a b)$
assumes *f3*: $\bigwedge x a. \text{cont } (\lambda b. f x a b)$
assumes *g*: *cont* ($\lambda x. g x$)
shows *cont* ($\lambda x. \text{case } g x \text{ of } (a, b) \Rightarrow f x a b$)


```

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:  $\bigwedge y. \text{cont } (\lambda x. f (x, y))$ 
assumes f2:  $\bigwedge x. \text{cont } (\lambda y. f (x, y))$ 
shows cont f
proof –
have cont  $(\lambda(x, y). f (x, y))$ 
by (intro cont2cont-case-prod f1 f2 cont2cont)
then show cont f
by (simp only: case-prod-eta)
qed

```

```

lemma prod-cont-iff:  $\text{cont } f \longleftrightarrow (\forall y. \text{cont } (\lambda x. f (x, y))) \wedge (\forall x. \text{cont } (\lambda y. f (x, y)))$ 
apply safe
apply (erule cont-compose [OF - cont-pair1])
apply (erule cont-compose [OF - cont-pair2])
apply (simp only: prod-contI)
done

```

```

lemma cont2cont-case-prod' [simp, cont2cont]:
assumes f:  $\text{cont } (\lambda p. f (fst p) (fst (snd p)) (snd (snd p)))$ 
assumes g:  $\text{cont } (\lambda x. g x)$ 
shows  $\text{cont } (\lambda x. \text{case-prod } (f x) (g x))$ 
using assms by (simp add: cont2cont-case-prod prod-cont-iff)

```

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

```

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

```

Admissibility of predicates on product types.

```

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

```

7.7 Compactness and chain-finiteness

```

lemma fst-below-iff:  $\text{fst } x \sqsubseteq y \longleftrightarrow x \sqsubseteq (y, \text{snd } x)$ 

```

```

for  $x :: 'a \times 'b$ 
by (simp add: below-prod-def)

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

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

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

lemma compact-Pair:  $compact\ x \implies compact\ y \implies compact\ (x, y)$ 
by (rule compactI) (simp add: below-prod-def)

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

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

end

```

8 The type of continuous functions

```

theory Cfun
imports Cpodef Fun-Cpo Product-Cpo
begin

```

```

default-sort cpo

```

8.1 Definition of continuous function type

```

definition cfun =  $\{f :: 'a \Rightarrow 'b. cont\ f\}$ 

```

```

cpodef ( $'a, 'b$ ) cfun (( $- \rightarrow / -$ ) [ $1, 0$ ]  $0$ ) =  $cfun :: ('a \Rightarrow 'b)\ set$ 
by (auto simp: cfun-def intro: cont-const adm-cont)

```

```

type-notation (ASCII)
cfun (infixr  $\rightarrow$ )

```

```

notation (ASCII)

```

Rep-cfun ((-\$/-) [999,1000] 999)

notation

Rep-cfun ((-./-) [999,1000] 999)

8.2 Syntax for continuous lambda abstraction

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

parse-translation <

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

print-translation <

[(***const-syntax*** <*Abs-cfun*>, *fn* - => *fn* [*Abs abs*] =>
 let val (*x*, *t*) = *Syntax-Trans.atomic-abs-tr'* *abs*
 in *Syntax.const* ***syntax-const*** <*-cabs*> \$ *x* \$ *t* end)]
 > — To avoid eta-contraction of body

Syntax for nested abstractions

syntax (*ASCII*)

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

syntax

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

parse-ast-translation <

(* *rewrite* (*LAM* *x* *y* *z*. *t*) => (*-cabs* *x* (*-cabs* *y* (*-cabs* *z* *t*))) *)
 (* *cf.* *Syntax.lambda-ast-tr* from *src/Pure/Syntax/syn-trans.ML* *)
let
 fun *Lambda-ast-tr* [*pats*, *body*] =
 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

```

 $\Lambda \cdot. t \rightarrow \text{CONST } \text{Abs-cfun } (\lambda \cdot. t)$ 

```

8.3 Continuous function space is pointed

lemma *bottom-cfun*: $\perp \in \text{cfun}$
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*]: $\perp \cdot x = \perp$
by (*simp add: Rep-cfun-strict*)

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

for compatibility with old HOLCF-Version

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

8.4 Basic properties of continuous functions

Beta-equality for continuous functions

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

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

8.4.1 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 = #2 (Thm.dest-comb (#2 (Thm.dest-comb 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`: $f = g \iff (\forall x. f \cdot x = g \cdot x)$
by (`simp` add: `Rep-cfun-inject` [`symmetric`] `fun-eq-iff`)

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

Extensionality wrt. ordering for continuous functions

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

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

Congruence for continuous function application

lemma `cfun-cong`: $f = g \implies x = y \implies f \cdot x = g \cdot y$
by `simp`

lemma `cfun-fun-cong`: $f = g \implies f \cdot x = g \cdot x$
by `simp`

lemma `cfun-arg-cong`: $x = y \implies f \cdot x = f \cdot y$
by `simp`

8.5 Continuity of application

lemma *cont-Rep-cfun1*: $\text{cont } (\lambda f. f \cdot x)$
by (*rule cont-Rep-cfun [OF cont-id, THEN cont2cont-fun]*)

lemma *cont-Rep-cfun2*: $\text{cont } (\lambda x. f \cdot x)$
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*: $\text{chain } Y \implies f \cdot (\bigsqcup i. Y i) = (\bigsqcup i. f \cdot (Y i))$
by (*rule cont-Rep-cfun2 [THEN cont2contlubE]*)

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

monotonicity of application

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

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

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

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

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

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

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

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

lemma *ch2ch-LAM [simp]*:
 $(\bigwedge x. \text{chain } (\lambda i. S i x)) \implies (\bigwedge i. \text{cont } (\lambda x. S i x)) \implies \text{chain } (\lambda i. \bigwedge x. S i x)$
by (*simp add: chain-def cfun-below-iff*)

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

lemma *lub-APP*: $\text{chain } F \implies \text{chain } Y \implies (\bigsqcup i. F i \cdot (Y i)) = (\bigsqcup i. F i) \cdot (\bigsqcup i. Y i)$

by (*simp add: contlub-cfun-fun contlub-cfun-arg diag-lub*)

lemma *lub-LAM*:

assumes $\bigwedge x. \text{chain } (\lambda i. F i x)$
 and $\bigwedge i. \text{cont } (\lambda x. F i x)$
 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-distrib = lub-APP lub-LAM*

strictness

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

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

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

8.6 Continuity simplification procedure

cont2cont lemma for *Rep-cfun*

lemma *cont2cont-APP* [*simp, cont2cont*]:

assumes $f: \text{cont } (\lambda x. f x)$
 assumes $t: \text{cont } (\lambda x. t x)$
 shows $\text{cont } (\lambda x. (f x) \cdot (t x))$

proof –

from *cont-Rep-cfun1* f have $\text{cont } (\lambda x. (f x) \cdot y)$ for y
 by (*rule cont-compose*)
 with t *cont-Rep-cfun2* show $\text{cont } (\lambda x. (f x) \cdot (t x))$
 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*]: $\text{cont } f \implies \text{cont } g \implies \text{cont } (\lambda x. ((f x) \cdot (g x)) s)$
 by (*rule cont2cont-APP [THEN cont2cont-fun]*)

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

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

lemma *cont2mono-LAM*:

$\llbracket \bigwedge x. \text{cont } (\lambda y. f x y); \bigwedge y. \text{monofun } (\lambda x. f x y) \rrbracket$
 $\implies \text{monofun } (\lambda x. \Lambda y. f x y)$

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*: $\bigwedge x. \text{cont } (\lambda y. f x y)$

assumes *f2*: $\bigwedge y. \text{cont } (\lambda x. f x y)$

shows *cont* $(\lambda x. \Lambda y. f x y)$

proof (*rule cont-Abs-cfun*)

from *f1* **show** $f x \in \text{cfun}$ **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* :: $'a::\text{cpo} \Rightarrow 'b::\text{cpo} \Rightarrow 'c::\text{cpo}$

assumes *f*: *cont* $(\lambda p. f (\text{fst } p) (\text{snd } p))$

shows *cont* $(\lambda x. \Lambda y. f x y)$

using *assms* **by** (*simp add: cont2cont-LAM prod-cont-iff*)

lemma *cont2cont-LAM-discrete* [*simp, cont2cont*]:

$(\bigwedge y::'a::\text{discrete-cpo}. \text{cont } (\lambda x. f x y)) \Longrightarrow \text{cont } (\lambda x. \Lambda y. f x y)$

by (*simp add: cont2cont-LAM*)

8.7 Miscellaneous

Monotonicity of *Abs-cfun*

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

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

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

lemma *chfin-Rep-cfunR*: *chain Y* $\Longrightarrow \forall s. \exists n. (\text{LUB } i. Y i) \cdot s = Y n \cdot s$

for *Y* :: $\text{nat} \Rightarrow 'a::\text{cpo} \rightarrow 'b::\text{chfin}$

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* $(\lambda(u::'a::\text{cpo} \rightarrow 'b::\text{chfin}). P(u \cdot s))$

by (*rule adm-subst, simp, rule adm-chfin*)

8.8 Continuous injection-retraction pairs

Continuous retractions are strict.

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

a result about functions with flat codomain

```

lemma flat-eqI:  $x \sqsubseteq y \implies x \neq \perp \implies x = y$ 
  for  $x\ y :: 'a::\text{flat}$ 
  by (drule ax-flat) simp

```

```

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

```

done

8.9 Identity and composition

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

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

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

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

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

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

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

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

Show that interpretation of $(pcpo, \rightarrow)$ is a category.

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

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

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

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

8.10 Strictified functions

default-sort $pcpo$

definition $seq :: 'a \rightarrow 'b \rightarrow 'b$
where $seq = (\Lambda x. \text{if } x = \perp \text{ then } \perp \text{ else } ID)$

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

lemma $seq\text{-conv}\text{-if}$: $seq.x = (\text{if } x = \perp \text{ then } \perp \text{ else } ID)$
by ($simp\ add: seq\text{-def}$)

lemma $seq\text{-simps}$ [$simp$]:
 $seq.\perp = \perp$
 $seq.x.\perp = \perp$
 $x \neq \perp \implies seq.x = ID$
by ($simp\text{-all}\ add: seq\text{-conv}\text{-if}$)

definition $strictify :: ('a \rightarrow 'b) \rightarrow 'a \rightarrow 'b$
where $strictify = (\Lambda f x. seq.x.(f.x))$

lemma $strictify\text{-conv}\text{-if}$: $strictify.f.x = (\text{if } x = \perp \text{ then } \perp \text{ else } f.x)$
by ($simp\ add: strictify\text{-def}$)

lemma $strictify1$ [$simp$]: $strictify.f.\perp = \perp$
by ($simp\ add: strictify\text{-conv}\text{-if}$)

lemma $strictify2$ [$simp$]: $x \neq \perp \implies strictify.f.x = f.x$
by ($simp\ add: strictify\text{-conv}\text{-if}$)

8.11 Continuity of let-bindings

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

lemma $cont2cont\text{-Let}'$ [$simp$, $cont2cont$]:
assumes $f: cont (\lambda x. f x)$
assumes $g: cont (\lambda p. g (fst p) (snd p))$

```

  shows cont ( $\lambda x. \text{let } y = f x \text{ in } g x y$ )
  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 ( $\lambda x. g x y$ ) 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  $\bigwedge y. \text{cont } (\lambda x. g x y)$ 
  shows cont ( $\lambda x. \text{let } y = t \text{ in } g x y$ )
  unfolding Let-def using assms .

```

end

9 Continuous deflations and ep-pairs

```

theory Deflation
  imports Cfun
begin

```

```

default-sort cpo

```

9.1 Continuous deflations

```

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

```

```

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

```

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

```

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

```

```

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

```

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

```

lemma belowI:
  assumes f:  $\bigwedge x. d \cdot x = x \implies f \cdot x = x$ 

```

```

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

```

```

lemma belowD:  $\llbracket f \sqsubseteq d; f \cdot x = x \rrbracket \implies d \cdot x = x$ 
proof (rule below-antisym)
  from below show  $d \cdot x \sqsubseteq x$  .
  assume  $f \sqsubseteq d$ 
  then have  $f \cdot x \sqsubseteq d \cdot x$  by (rule monofun-cfun-fun)
  also assume  $f \cdot x = x$ 
  finally show  $x \sqsubseteq d \cdot x$  .
qed

```

end

```

lemma deflation-strict: deflation  $d \implies d \cdot \perp = \perp$ 
  by (rule deflation.below [THEN bottomI])

```

```

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

```

```

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

```

```

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

```

```

lemma deflation-below-iff: deflation  $p \implies$  deflation  $q \implies p \sqsubseteq q \iff (\forall x. p \cdot x = x \longrightarrow q \cdot x = x)$ 
  apply safe
  apply (simp add: deflation.belowD)
  apply (simp add: deflation.belowI)
  done

```

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

```

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

```

next

interpret f : *deflation* f **by fact**
assume $f \sqsubseteq g$
then have $f \cdot x \sqsubseteq g \cdot x$ **by** (*rule monofun-cfun-fun*)
then have $f \cdot (f \cdot x) \sqsubseteq f \cdot (g \cdot x)$ **by** (*rule monofun-cfun-arg*)
also have $f \cdot (f \cdot x) = f \cdot x$ **by** (*rule f.idem*)
finally show $f \cdot x \sqsubseteq f \cdot (g \cdot x)$.

qed

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

9.2 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 $Y :: \text{nat} \Rightarrow 'a$

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-theub 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  $d \implies \text{finite } \{x. d \cdot x = x\} \implies \text{finite-deflation } d$ 
  by (intro finite-deflation.intro finite-deflation-axioms.intro)

```

```

lemma finite-deflation-imp-deflation: finite-deflation  $d \implies \text{deflation } d$ 
  by (simp add: finite-deflation-def)

```

```

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

```

9.3 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:  $e \cdot (p \cdot x) = x \implies e \cdot (p \cdot y) = y \implies p \cdot x = p \cdot y \longleftrightarrow x = y$ 
  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*: $e.x \sqsubseteq y \longleftrightarrow x \sqsubseteq p.y$

proof

assume $e.x \sqsubseteq y$

then have $p.(e.x) \sqsubseteq p.y$ by (*rule monofun-cfun-arg*)

then show $x \sqsubseteq p.y$ by *simp*

next

assume $x \sqsubseteq p.y$

then have $e.x \sqsubseteq e.(p.y)$ by (*rule monofun-cfun-arg*)

then show $e.x \sqsubseteq y$ using *e-p-below* by (*rule below-trans*)

qed

lemma *compact-e-rev*: $\text{compact } (e.x) \implies \text{compact } x$

proof –

assume $\text{compact } (e.x)$

then have $\text{adm } (\lambda y. e.x \not\sqsubseteq y)$ by (*rule compactD*)

then have $\text{adm } (\lambda y. e.x \not\sqsubseteq e.y)$ by (*rule adm-subst [OF cont-Rep-cfun2]*)

then have $\text{adm } (\lambda y. x \not\sqsubseteq y)$ by *simp*

then show $\text{compact } x$ by (*rule compactI*)

qed

lemma *compact-e*:

assumes $\text{compact } x$

shows $\text{compact } (e.x)$

proof –

from *assms* have $\text{adm } (\lambda y. x \not\sqsubseteq y)$ by (*rule compactD*)

then have $\text{adm } (\lambda y. x \not\sqsubseteq p.y)$ by (*rule adm-subst [OF cont-Rep-cfun2]*)

then have $\text{adm } (\lambda y. e.x \not\sqsubseteq y)$ by (*simp add: e-below-iff-below-p*)

then show $\text{compact } (e.x)$ by (*rule compactI*)

qed

lemma *compact-e-iff*: $\text{compact } (e.x) \longleftrightarrow \text{compact } x$

by (*rule iffI [OF compact-e-rev compact-e]*)

Deflations from ep-pairs

lemma *deflation-e-p*: $\text{deflation } (e \text{ oo } p)$

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

lemma *deflation-e-d-p*:

assumes $\text{deflation } d$

shows $\text{deflation } (e \text{ oo } d \text{ oo } p)$

proof

interpret $\text{deflation } d$ by *fact*

fix $x :: 'b$

show $(e \text{ oo } d \text{ oo } p) \cdot ((e \text{ oo } d \text{ oo } p) \cdot x) = (e \text{ oo } d \text{ oo } p) \cdot x$

by (*simp add: idem*)

show $(e \text{ oo } d \text{ oo } p) \cdot x \sqsubseteq x$

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 $(e \text{ oo } d \text{ oo } p) \cdot ((e \text{ oo } d \text{ oo } p) \cdot x) = (e \text{ oo } d \text{ oo } p) \cdot x$

by (*simp add: idem*)

show $(e \text{ oo } d \text{ oo } p) \cdot x \sqsubseteq x$

by (*simp add: e-below-iff-below-p below*)

have *finite* $((\lambda x. e \cdot x) \text{ ' } (\lambda x. d \cdot x) \text{ ' } \text{range } (\lambda x. p \cdot x))$

by (*simp add: finite-image*)

then have *finite* $(\text{range } (\lambda x. (e \text{ oo } d \text{ oo } p) \cdot x))$

by (*simp add: image-image*)

then show *finite* $\{x. (e \text{ oo } d \text{ oo } p) \cdot x = x\}$

by (*rule finite-range-imp-finite-fixes*)

qed

lemma *deflation-p-d-e*:

assumes *deflation d*

assumes $d: \bigwedge x. d \cdot x \sqsubseteq e \cdot (p \cdot x)$

shows *deflation (p oo d oo e)*

proof –

interpret $d: \text{deflation } d$ by fact

have *p-d-e-below*: $(p \text{ oo } d \text{ oo } e) \cdot x \sqsubseteq x$ for x

proof –

have $d \cdot (e \cdot x) \sqsubseteq e \cdot x$

by (*rule d.below*)

then have $p \cdot (d \cdot (e \cdot x)) \sqsubseteq p \cdot (e \cdot x)$

by (*rule monofun-cfun-arg*)

then show *?thesis* by *simp*

qed

show *?thesis*

proof

show $(p \text{ oo } d \text{ oo } e) \cdot x \sqsubseteq x$ for x

by (*rule p-d-e-below*)

show $(p \text{ oo } d \text{ oo } e) \cdot ((p \text{ oo } d \text{ oo } e) \cdot x) = (p \text{ oo } d \text{ oo } e) \cdot x$ for x

proof (*rule below-antisym*)

show $(p \text{ oo } d \text{ oo } e) \cdot ((p \text{ oo } d \text{ oo } e) \cdot x) \sqsubseteq (p \text{ oo } d \text{ oo } e) \cdot x$

by (*rule p-d-e-below*)

have $p \cdot (d \cdot (d \cdot (d \cdot (e \cdot x)))) \sqsubseteq p \cdot (d \cdot (e \cdot (p \cdot (d \cdot (e \cdot x))))))$

by (*intro monofun-cfun-arg d*)

then have $p \cdot (d \cdot (e \cdot x)) \sqsubseteq p \cdot (d \cdot (e \cdot (p \cdot (d \cdot (e \cdot x))))))$

by (*simp only: d.idem*)

then show $(p \text{ oo } d \text{ oo } e) \cdot x \sqsubseteq (p \text{ oo } d \text{ oo } e) \cdot ((p \text{ oo } d \text{ oo } e) \cdot x)$

by *simp*

qed

```

qed
qed

lemma finite-deflation-p-d-e:
  assumes finite-deflation d
  assumes d:  $\bigwedge x. d \cdot x \sqsubseteq e \cdot (p \cdot x)$ 
  shows finite-deflation (p oo d oo e)
proof -
  interpret d: finite-deflation d by fact
  show ?thesis
  proof (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 (( $\lambda x. d \cdot x$ ) ‘ range ( $\lambda x. e \cdot x$ ))
      by (rule d.finite-image)
    then have finite (( $\lambda x. p \cdot x$ ) ‘ ( $\lambda x. d \cdot x$ ) ‘ range ( $\lambda x. e \cdot x$ ))
      by (rule finite-imageI)
    then have finite (range ( $\lambda x. (p \text{ oo } d \text{ oo } e) \cdot x$ ))
      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

```

9.4 Uniqueness of ep-pairs

```

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

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

lemma ep-pair-unique-p-lemma:
  assumes 1: ep-pair e p1
  and 2: ep-pair e p2
  shows p1  $\sqsubseteq$  p2

```

```

proof (rule cfun-belowI)
  fix x
  have  $e.(p1.x) \sqsubseteq x$ 
    by (rule ep-pair.e-p-below [OF 1])
  then have  $p2.(e.(p1.x)) \sqsubseteq p2.x$ 
    by (rule monofun-cfun-arg)
  then show  $p1.x \sqsubseteq p2.x$ 
    by (simp only: ep-pair.e-inverse [OF 2])
qed

```

```

lemma ep-pair-unique-p: ep-pair e p1  $\implies$  ep-pair e p2  $\implies$  p1 = p2
  by (fast intro: below-antisym elim: ep-pair-unique-p-lemma)

```

9.5 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  $(p1 \text{ oo } p2).((e2 \text{ oo } e1).x) = x$ 
    by simp
  have  $e1.(p1.(p2.y)) \sqsubseteq p2.y$ 
    by (rule ep1.e-p-below)
  then have  $e2.(e1.(p1.(p2.y))) \sqsubseteq e2.(p2.y)$ 
    by (rule monofun-cfun-arg)
  also have  $e2.(p2.y) \sqsubseteq y$ 
    by (rule ep2.e-p-below)
  finally show  $(e2 \text{ oo } e1).((p1 \text{ oo } p2).y) \sqsubseteq y$ 
    by simp
qed

```

```

locale pcpo-ep-pair = ep-pair e p
  for e :: 'a::pcpo  $\rightarrow$  'b::pcpo
  and p :: 'b::pcpo  $\rightarrow$  'a::pcpo
begin

```

```

lemma e-strict [simp]:  $e.\perp = \perp$ 
proof –
  have  $\perp \sqsubseteq p.\perp$  by (rule minimal)
  then have  $e.\perp \sqsubseteq e.(p.\perp)$  by (rule monofun-cfun-arg)
  also have  $e.(p.\perp) \sqsubseteq \perp$  by (rule e-p-below)
  finally show  $e.\perp = \perp$  by simp
qed

```

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

lemma *e-defined*: $x \neq \perp \implies e \cdot x \neq \perp$
by *simp*

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

lemmas *stricts = e-strict p-strict*

end

end

10 The type of strict products

theory *Sprod*
imports *Cfun*
begin

default-sort *pcpo*

10.1 Definition of strict product type

definition *sprod* = $\{p :: 'a \times 'b. p = \perp \vee (fst\ p \neq \perp \wedge snd\ p \neq \perp)\}$

pcpodef (*'a*, *'b*) *sprod* ((- \otimes / -) [*21,20*] *20*) = *sprod* :: (*'a* \times *'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*)

10.2 Definitions of constants

definition *sfst* :: (*'a* ** *'b*) \rightarrow *'a*
where *sfst* = ($\Lambda p. fst\ (Rep\ sprod\ p)$)

definition *ssnd* :: (*'a* ** *'b*) \rightarrow *'b*
where *ssnd* = ($\Lambda p. snd\ (Rep\ sprod\ p)$)

definition *spair* :: *'a* \rightarrow *'b* \rightarrow (*'a* ** *'b*)
where *spair* = ($\Lambda a\ b. Abs\ sprod\ (seq \cdot b \cdot a, seq \cdot a \cdot b)$)

definition *ssplit* :: (*'a* \rightarrow *'b* \rightarrow *'c*) \rightarrow (*'a* ** *'b*) \rightarrow *'c*

where $ssplit = (\Lambda f p. seq \cdot p \cdot (f \cdot (sfst \cdot p) \cdot (ssnd \cdot p)))$

syntax $-stuple :: [logic, args] \Rightarrow logic \ ((1'(-, / -'))$

translations

$(:x, y, z:) \Leftrightarrow (:x, (:y, z:))$

$(:x, y:) \Leftrightarrow CONST \ spair \cdot x \cdot y$

translations

$\Lambda(CONST \ spair \cdot x \cdot y). t \Leftrightarrow CONST \ ssplit \cdot (\Lambda x y. t)$

10.3 Case analysis

lemma $spair-sprod: (seq \cdot b \cdot a, seq \cdot a \cdot b) \in sprod$

by ($simp \ add: \ sprod-def \ seq-conv-if$)

lemma $Rep-sprod-spair: Rep-sprod \ (:a, b:) = (seq \cdot b \cdot a, seq \cdot a \cdot b)$

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 $p = \perp \mid x \ y$ **where** $p = (:x, y:)$ **and** $x \neq \perp$ **and** $y \neq \perp$

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

lemma $sprod-induct \ [case-names \ bottom \ spair, \ induct \ type: \ sprod]:$

$\llbracket P \ \perp; \ \bigwedge x \ y. \llbracket x \neq \perp; \ y \neq \perp \rrbracket \Longrightarrow P \ (:x, y:) \rrbracket \Longrightarrow P \ x$

by ($cases \ x$) $simp-all$

10.4 Properties of *spair*

lemma $spair-strict1 \ [simp]: \ (:\perp, y:) = \perp$

by ($simp \ add: \ Rep-sprod-simps$)

lemma $spair-strict2 \ [simp]: \ (:x, \perp:) = \perp$

by ($simp \ add: \ Rep-sprod-simps$)

lemma $spair-bottom-iff \ [simp]: \ (:x, y:) = \perp \longleftrightarrow x = \perp \vee y = \perp$

by ($simp \ add: \ Rep-sprod-simps \ seq-conv-if$)

lemma $spair-below-iff: \ (:a, b:) \sqsubseteq \ (:c, d:) \longleftrightarrow a = \perp \vee b = \perp \vee (a \sqsubseteq c \wedge b \sqsubseteq d)$

by ($simp \ add: \ Rep-sprod-simps \ seq-conv-if$)

lemma $spair-eq-iff: \ (:a, b:) = \ (:c, d:) \longleftrightarrow a = c \wedge b = d \vee (a = \perp \vee b = \perp) \wedge (c = \perp \vee d = \perp)$

by ($simp \ add: \ Rep-sprod-simps \ seq-conv-if$)

lemma $spair-strict: \ x = \perp \vee y = \perp \Longrightarrow \ (:x, y:) = \perp$

by *simp*

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

lemma *spair-defined*: $\llbracket x \neq \perp; y \neq \perp \rrbracket \implies (:x, y:) \neq \perp$
by *simp*

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

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

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

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

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

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

10.5 Properties of *sfst* and *ssnd*

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

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

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

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

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

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

lemma *sfst-defined*: $p \neq \perp \implies sfst.p \neq \perp$
by *simp*

lemma *ssnd-defined*: $p \neq \perp \implies \text{ssnd} \cdot p \neq \perp$
by *simp*

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

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

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

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

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

10.6 Compactness

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

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

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

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

10.7 Properties of *ssplit*

lemma *ssplit1* [*simp*]: $\text{ssplit} \cdot f \cdot \perp = \perp$
by (*simp add: ssplit-def*)

lemma *ssplit2* [*simp*]: $x \neq \perp \implies y \neq \perp \implies \text{ssplit} \cdot f \cdot (:x, y) = f \cdot x \cdot y$
by (*simp add: ssplit-def*)

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

10.8 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

```

11 Discrete cpo types

```

theory Discrete
  imports Cont
begin

datatype 'a discr = Discr 'a :: type

11.1 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

11.2 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

```


12 The type of lifted values

```
theory Up
  imports Cfun
begin
```

```
default-sort cpo
```

12.1 Definition of new type for lifting

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

```
primrec Ifup :: ('a  $\rightarrow$  'b::pcpo)  $\Rightarrow$  'a u  $\Rightarrow$  'b
```

```
where
```

```
  Ifup f Ibottom =  $\perp$ 
| Ifup f (Iup x) = f·x
```

12.2 Ordering on lifted cpo

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

```
begin
```

```
definition below-up-def:
```

```
( $\sqsubseteq$ )  $\equiv$ 
  ( $\lambda$ x y.
    (case x of
      Ibottom  $\Rightarrow$  True
    | Iup a  $\Rightarrow$  (case y of Ibottom  $\Rightarrow$  False | Iup b  $\Rightarrow$  a  $\sqsubseteq$  b)))
```

```
instance ..
```

```
end
```

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

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

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

12.3 Lifted cpo is a partial order

```
instance u :: (cpo) po
```

```
proof
```

```
  fix x :: 'a u
```

```
  show x  $\sqsubseteq$  x
```

```
  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*: $\text{range } S \ll\ll x \implies \text{range } (\lambda i. Iup (S i)) \ll\ll Iup x$
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** $\text{range } Y \ll\ll Iup (\bigsqcup i. A i)$

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 $\text{range } A \ll\ll (\bigsqcup i. A i)$

by (*rule cpo-lubI*)

then have $\text{range } (\lambda i. Iup (A i)) \ll\ll Iup (\bigsqcup i. A i)$

by (*rule is-lub-Iup*)

then have $\text{range } (\lambda i. Y (i + k)) \ll\ll Iup (\bigsqcup i. A i)$

by (*simp only: Iup-A*)

then have $\text{range } (\lambda i. Y i) \ll\ll Iup (\bigsqcup i. A i)$

by (*simp only: is-lub-range-shift [OF Y]*)

with *Iup-A chain-A* **show** *?thesis* ..

next

case *False*

then have $\forall i. Y i = Ibottom$ **by** *simp*

then show *?thesis* ..

qed

```

instance u :: (cpo) cpo
proof
  fix S :: nat => 'a u
  assume S: chain S
  then show  $\exists x. \text{range } (\lambda i. S i) \ll\!| x$ 
  proof (rule up-chain-lemma)
    assume  $\forall i. S i = \text{Ibottom}$ 
    then have  $\text{range } (\lambda i. S i) \ll\!| \text{Ibottom}$ 
      by (simp add: is-lub-const)
    then show ?thesis ..
  next
  fix A :: nat => 'a
  assume  $\text{range } S \ll\!| \text{Iup } (\bigsqcup i. A i)$ 
  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:  $\perp = \text{Ibottom}$ 
  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 ( $\lambda f. \text{Ifup } f x$ )
  by (induct x) simp-all

```

```

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

```

```

lemma cont-Ifup2: cont ( $\lambda x. \text{Ifup } f x$ )

```

proof (*rule contI2*)
fix Y
assume Y : *chain* Y **and** Y' : *chain* $(\lambda i. \text{Ifup } f \ (Y \ i))$
from Y **show** $\text{Ifup } f \ (\bigsqcup i. Y \ i) \sqsubseteq (\bigsqcup i. \text{Ifup } f \ (Y \ i))$
proof (*rule up-chain-lemma*)
fix A **and** k
assume A : $\forall i. \text{Iup } (A \ i) = Y \ (i + k)$
assume *chain* A **and** *range* $Y \ll \text{Iup } (\bigsqcup i. A \ i)$
then have $\text{Ifup } f \ (\bigsqcup i. Y \ i) = (\bigsqcup i. \text{Ifup } f \ (\text{Iup } (A \ i)))$
by (*simp add: lub-eqI contlub-cfun-arg*)
also have $\dots = (\bigsqcup i. \text{Ifup } f \ (Y \ (i + k)))$
by (*simp add: A*)
also have $\dots = (\bigsqcup i. \text{Ifup } f \ (Y \ i))$
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 $up :: 'a \rightarrow 'a \ u$
where $up = (\Lambda x. \text{Iup } x)$

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

translations

case l of XCONST up.x $\Rightarrow t \Leftrightarrow \text{CONST } fup \cdot (\Lambda x. t) \cdot l$
case l of (XCONST up :: 'a).x $\Rightarrow t \rightarrow \text{CONST } fup \cdot (\Lambda x. t) \cdot l$
 $\Lambda(\text{XCONST } up \cdot x). t \Leftrightarrow \text{CONST } fup \cdot (\Lambda x. t)$

continuous versions of lemmas for $'a_{\perp}$

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

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

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

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

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

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

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

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

lifting preserves chain-finiteness

lemma *up-chain-cases*:

assumes *Y: chain Y*

obtains $\forall i. Y i = \perp$

| *A k* **where** $\forall i. \text{up}\cdot(A i) = Y (i + k)$ **and** *chain A* **and** $(\bigsqcup i. Y i) = \text{up}\cdot(\bigsqcup i. A i)$

by (*rule up-chain-lemma [OF Y]*) (*simp-all add: inst-up-pcpo up-def cont-Iup lub-eqI*)

lemma *compact-up*: *compact x* \implies *compact (up·x)*

apply (*rule compactI2*)

apply (*erule up-chain-cases*)

apply *simp*

apply (*drule (1) compactD2, simp*)

apply (*erule exE*)

apply (*drule-tac f=up and x=x in monofun-cfun-arg*)

apply (*simp, erule exI*)

done

lemma *compact-upD*: *compact (up·x)* \implies *compact x*

unfolding *compact-def*

by (*drule adm-subst [OF cont-Rep-cfun2 [where f=up]], simp*)

lemma *compact-up-iff* [*simp*]: *compact (up·x) = compact x*

by (*safe elim!: compact-up compact-upD*)

instance *u* :: (*chfin*) *chfin*

apply *intro-classes*

apply (*erule compact-imp-max-in-chain*)

apply (*rule-tac p= $\bigsqcup i. Y i$ in upE, simp-all*)

done

properties of fup

lemma *fup1* [*simp*]: *fup·f· $\perp = \perp$*

by (*simp add: fup-def cont-Ifup1 cont-Ifup2 inst-up-pcpo cont2cont-LAM*)

lemma *fup2* [*simp*]: *fup·f·(up·x) = f·x*

by (*simp add: up-def fup-def cont-Iup cont-Ifup1 cont-Ifup2 cont2cont-LAM*)

lemma *fup3* [*simp*]: *fup·up·x = x*

```

  by (cases x) simp-all
end

```

13 Lifting types of class type to flat pcpo’s

```

theory Lift
imports Discrete Up
begin

default-sort type

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

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

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

```

13.1 Lift as a datatype

```

lemma lift-induct:  $\llbracket P \perp; \bigwedge x. P (Def x) \rrbracket \implies P y$ 
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  $\perp :: 'a lift Def$ 
  by (erule lift-induct) (simp-all add: Def-def Abs-lift-inject inst-lift-pcpo)

 $\perp$  and Def

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

lemma lift-definedE:  $\llbracket x \neq \perp; \bigwedge a. x = Def a \implies R \rrbracket \implies R$ 
  by (cases x) simp-all

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

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

lemma DefE:  $Def x = \perp \implies R$ 

```

by *simp*

lemma *DefE2*: $\llbracket x = \text{Def } s; x = \perp \rrbracket \implies R$
by *simp*

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

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

13.2 Lift is flat

instance *lift* :: (type) flat

proof

fix *x y* :: 'a lift

assume $x \sqsubseteq y$ thus $x = \perp \vee x = y$

by (*induct x auto*)

qed

13.3 Continuity of case-lift

lemma *case-lift-eq*: $\text{case-lift } \perp f x = \text{fup} \cdot (\Lambda y. f (\text{undiscr } y)) \cdot (\text{Rep-lift } x)$

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*]:

$\llbracket \Lambda y. \text{cont } (\lambda x. f x y); \text{cont } g \rrbracket \implies \text{cont } (\lambda x. \text{case-lift } \perp (f x) (g x))$

unfolding *case-lift-eq* by (*simp add: cont-Rep-lift*)

13.4 Further operations

definition

flift1 :: ('a \Rightarrow 'b::pcpo) \Rightarrow ('a lift \rightarrow 'b) (**binder** FLIFT 10) **where**

flift1 = ($\lambda f. (\Lambda x. \text{case-lift } \perp f x)$)

translations

$\Lambda (XCONST \text{Def } x). t \Rightarrow CONST \text{flift1 } (\lambda x. t)$

$\Lambda (CONST \text{Def } x). FLIFT y. t \leq FLIFT x y. t$

$\Lambda (CONST \text{Def } x). t \leq FLIFT x. t$

definition

flift2 :: ('a \Rightarrow 'b) \Rightarrow ('a lift \rightarrow 'b lift) **where**

flift2 *f* = ($FLIFT x. \text{Def } (f x)$)

lemma *flift1-Def* [*simp*]: $\text{flift1 } f \cdot (\text{Def } x) = (f x)$

by (*simp add: flift1-def*)

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

lemma *flift1-strict* [*simp*]: $\text{flift1 } f \cdot \perp = \perp$
by (*simp add: flift1-def*)

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

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

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

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

lemma *cont2cont-flift1* [*simp, cont2cont*]:
 $\llbracket \bigwedge y. \text{cont } (\lambda x. f x y) \rrbracket \implies \text{cont } (\lambda x. \text{FLIFT } y. f x y)$
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
 $(\text{type}) \text{ tr} \leftarrow (\text{type}) \text{ bool lift}$

definition *TT* :: *tr*
where $\text{TT} = \text{Def True}$

definition *FF* :: *tr*
where $\text{FF} = \text{Def False}$

Exhaustion and Elimination for type *tr*

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

lemma *trE* [*case-names bottom TT FF, cases type: tr*]:

$\llbracket p = \perp \implies Q; p = TT \implies Q; p = FF \implies Q \rrbracket \implies Q$
by (*induct p*) (*auto simp: FF-def TT-def*)

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

distinctness for type *tr*

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

lemma *dist-eq-tr* [*simp*]: $TT \neq \perp \quad FF \neq \perp \quad TT \neq FF \quad \perp \neq TT \quad \perp \neq FF \quad FF \neq TT$
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

default-sort *pcpo*

definition *tr-case* :: $'a \rightarrow 'a \rightarrow tr \rightarrow 'a$
where *tr-case* = $(\Lambda t e (Def b). \text{if } b \text{ then } t \text{ else } e)$

abbreviation *cifte-syn* :: $[tr, 'c, 'c] \Rightarrow 'c \quad ((\text{If } (-) / \text{ then } (-) / \text{ else } (-)) [0, 0, 60]$
60)

where *If b then e1 else e2* $\equiv tr\text{-case}.e1 \cdot e2 \cdot b$

translations

$\Lambda (XCONST TT). t \Rightarrow CONST tr\text{-case}.t \cdot \perp$
 $\Lambda (XCONST FF). t \Rightarrow CONST tr\text{-case}.\perp \cdot t$

lemma *ifte-thms* [*simp*]:
If \perp *then* *e1* *else* *e2* = \perp
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 $trand :: tr \rightarrow tr \rightarrow tr$

where $andalso-def: trand = (\Lambda x y. \text{If } x \text{ then } y \text{ else } FF)$

abbreviation $andalso-syn :: tr \Rightarrow tr \Rightarrow tr$ (- $andalso$ - [36,35] 35)

where $x \text{ andalso } y \equiv trand \cdot x \cdot y$

definition $tror :: tr \rightarrow tr \rightarrow tr$

where $orelse-def: tror = (\Lambda x y. \text{If } x \text{ then } TT \text{ else } y)$

abbreviation $orelse-syn :: tr \Rightarrow tr \Rightarrow tr$ (- $orelse$ - [31,30] 30)

where $x \text{ orelse } y \equiv tror \cdot x \cdot y$

definition $neg :: tr \rightarrow tr$

where $neg = flift2 \text{ Not}$

definition $If2 :: tr \Rightarrow 'c \Rightarrow 'c \Rightarrow 'c$

where $If2 Q x y = (\text{If } Q \text{ then } x \text{ else } y)$

tactic for tr-thms with case split

lemmas $tr-defs = andalso-def \text{ orelse-def } neg-def \text{ tr-case-def } TT-def \text{ FF-def}$

lemmas about andalso, orelse, neg and if

lemma $andalso-thms [simp]:$

$(TT \text{ andalso } y) = y$

$(FF \text{ andalso } y) = FF$

$(\perp \text{ andalso } y) = \perp$

$(y \text{ andalso } TT) = y$

$(y \text{ andalso } y) = y$

apply $(unfold \text{ andalso-def}, \text{ simp-all})$

apply $(cases \text{ } y, \text{ simp-all})$

apply $(cases \text{ } y, \text{ simp-all})$

done

lemma $orelse-thms [simp]:$

$(TT \text{ orelse } y) = TT$

$(FF \text{ orelse } y) = y$

$(\perp \text{ orelse } y) = \perp$

$(y \text{ orelse } FF) = y$

$(y \text{ orelse } y) = y$

apply $(unfold \text{ orelse-def}, \text{ simp-all})$

apply $(cases \text{ } y, \text{ simp-all})$

apply $(cases \text{ } y, \text{ simp-all})$

done

lemma $neg-thms [simp]:$

$neg \cdot TT = FF$

$neg \cdot FF = TT$

$neg \cdot \perp = \perp$

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*: $P (If2\ Q\ x\ y) \longleftrightarrow ((Q = \perp \longrightarrow P\ \perp) \wedge (Q = TT \longrightarrow P\ x) \wedge (Q = FF \longrightarrow P\ y))$

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*: $t \neq \perp \implies (t\ \text{andalso}\ s) = FF \longleftrightarrow t = FF \vee s = FF$

by (*cases t*) *simp-all*

lemma *andalso-and*: $t \neq \perp \implies ((t\ \text{andalso}\ s) \neq FF) \longleftrightarrow t \neq FF \wedge s \neq FF$

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
```

```
default-sort pcpo
```

15.1 Definition of strict sum type

```
definition ssum =
```

```
{p :: tr × ('a × 'b). p = ⊥ ∨
 (fst p = TT ∧ fst (snd p) ≠ ⊥ ∧ snd (snd p) = ⊥) ∨
 (fst p = FF ∧ fst (snd p) = ⊥ ∧ snd (snd p) ≠ ⊥)}
```

```
pcpodef ('a, 'b) ssum ((- ⊕/ -) [21, 20] 20) = ssum :: (tr × 'a × 'b) set
  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 → ('a ++ 'b)
  where sinl = (λ a. Abs-ssum (seq.a.TT, a, ⊥))
```

```
definition sinr :: 'b → ('a ++ 'b)
  where sinr = (λ b. Abs-ssum (seq.b.FF, ⊥, b))
```

```
lemma sinl-ssum: (seq.a.TT, a, ⊥) ∈ ssum
  by (simp add: ssum-def seq-conv-if)
```

```
lemma sinr-ssum: (seq.b.FF, ⊥, b) ∈ ssum
  by (simp add: ssum-def seq-conv-if)
```

```
lemma Rep-ssum-sinl: Rep-ssum (sinl.a) = (seq.a.TT, a, ⊥)
  by (simp add: sinl-def cont-Abs-ssum Abs-ssum-inverse sinl-ssum)
```

```
lemma Rep-ssum-sinr: Rep-ssum (sinr.b) = (seq.b.FF, ⊥, 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*]: $\text{sinl}\cdot x \sqsubseteq \text{sinl}\cdot y \longleftrightarrow x \sqsubseteq y$
by (*simp add: Rep-ssum-simps seq-conv-if*)

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

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

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

Equality

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

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

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

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

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

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

Strictness

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

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

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

lemma *sinr-bottom-iff* [*simp*]: $\text{sinr}\cdot x = \perp \longleftrightarrow x = \perp$
using *sinr-eq [of x \perp]* **by** *simp*

lemma *sinl-defined*: $x \neq \perp \Longrightarrow \text{sinl}\cdot x \neq \perp$
by *simp*

lemma *sinr-defined*: $x \neq \perp \implies \text{sinr}\cdot x \neq \perp$
by *simp*

Compactness

lemma *compact-sinl*: $\text{compact } x \implies \text{compact } (\text{sinl}\cdot x)$
by (*rule compact-ssum*) (*simp add: Rep-ssum-sinl*)

lemma *compact-sinr*: $\text{compact } x \implies \text{compact } (\text{sinr}\cdot x)$
by (*rule compact-ssum*) (*simp add: Rep-ssum-sinr*)

lemma *compact-sinlD*: $\text{compact } (\text{sinl}\cdot x) \implies \text{compact } x$
unfolding *compact-def*
by (*drule adm-subst [OF cont-Rep-cfun2 [where f=sinl]]*, *simp*)

lemma *compact-sinrD*: $\text{compact } (\text{sinr}\cdot x) \implies \text{compact } x$
unfolding *compact-def*
by (*drule adm-subst [OF cont-Rep-cfun2 [where f=sinr]]*, *simp*)

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

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

15.4 Case analysis

lemma *ssumE* [*case-names bottom sinl sinr, cases type: ssum*]:
obtains $p = \perp$
| x **where** $p = \text{sinl}\cdot x$ **and** $x \neq \perp$
| y **where** $p = \text{sinr}\cdot y$ **and** $y \neq \perp$
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*]:
 $\llbracket P \perp;$
 $\bigwedge x. x \neq \perp \implies P (\text{sinl}\cdot x);$
 $\bigwedge y. y \neq \perp \implies P (\text{sinr}\cdot y) \rrbracket \implies P x$
by (*cases x*) *simp-all*

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

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

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

15.5 Case analysis combinator

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

translations

$\text{case } s \text{ of } XCONST \text{ sinl} \cdot x \Rightarrow t1 \mid XCONST \text{ sinr} \cdot y \Rightarrow t2 \Rightarrow CONST \text{ sscase} \cdot (\Lambda x. t1) \cdot (\Lambda y. t2) \cdot s$

$\text{case } s \text{ of } (XCONST \text{ sinl} :: 'a) \cdot x \Rightarrow t1 \mid XCONST \text{ sinr} \cdot y \Rightarrow t2 \rightarrow CONST \text{ sscase} \cdot (\Lambda x. t1) \cdot (\Lambda y. t2) \cdot s$

translations

$\Lambda(XCONST \text{ sinl} \cdot x). t \Rightarrow CONST \text{ sscase} \cdot (\Lambda x. t) \cdot \perp$
 $\Lambda(XCONST \text{ sinr} \cdot y). t \Rightarrow CONST \text{ sscase} \cdot \perp \cdot (\Lambda y. t)$

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

lemma $\text{sscase1} [\text{simp}]: \text{sscase} \cdot f \cdot g \cdot \perp = \perp$
by ($\text{simp add: beta-sscase Rep-ssum-strict}$)

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

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

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

15.6 Strict sum preserves flatness

instance $\text{ssum} :: (\text{flat}, \text{flat}) \text{ flat}$
apply ($\text{intro-classes, clarify}$)
apply ($\text{case-tac } x, \text{ simp}$)
apply ($\text{case-tac } y, \text{ simp-all add: flat-below-iff}$)
apply ($\text{case-tac } y, \text{ simp-all add: flat-below-iff}$)
done

end

16 The Strict Function Type

theory $Sfun$
imports $Cfun$
begin

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

type-notation (*ASCII*)

sfun (**infixr** $\rightarrow!$ 0)

TODO: Define nice syntax for abstraction, application.

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

definition *sfun-rep* :: ($'a \rightarrow!$ $'b$) \rightarrow $'a \rightarrow 'b$
where *sfun-rep* = (Λf . *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*. \perp = \perp
unfolding *sfun-rep-beta* **by** (*rule* *Rep-sfun-strict*)

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

lemma *strictify-cancel*: $f \cdot \perp = \perp \implies \text{strictify}.f = f$
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*: $f = g \iff \text{sfun-rep}.f = \text{sfun-rep}.g$
by (*simp* *add*: *sfun-rep-def* *cont-Rep-sfun* *Rep-sfun-inject*)

lemma *sfun-below-iff*: $f \sqsubseteq g \iff \text{sfun-rep}.f \sqsubseteq \text{sfun-rep}.g$
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

default-sort *cpo*

definition *cfun-map* :: ($'b \rightarrow 'a$) \rightarrow ($'c \rightarrow 'd$) \rightarrow ($'a \rightarrow 'c$) \rightarrow ($'b \rightarrow 'd$)
 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* ($\text{range } (\lambda x. a \cdot x)$)

assumes b : *finite* ($\text{range } (\lambda y. b \cdot y)$)

shows *finite* ($\text{range } (\lambda f. \text{cfun-map} \cdot a \cdot b \cdot f)$) (**is** *finite* ($\text{range } ?h$))

proof (*rule finite-imageD*)

let $?f = \lambda g. \text{range } (\lambda x. (a \cdot x, g \cdot x))$

show *finite* ($?f \text{ ' range } ?h$)

proof (*rule finite-subset*)

let $?B = \text{Pow } (\text{range } (\lambda x. a \cdot x) \times \text{range } (\lambda y. b \cdot y))$

show $?f \text{ ' range } ?h \subseteq ?B$

by *clarsimp*

show *finite* $?B$

by (*simp add: a b*)

qed

show *inj-on* $?f$ ($\text{range } ?h$)

proof (*rule inj-onI, rule cfun-eqI, clarsimp*)

fix $x f g$

assume $\text{range } (\lambda x. (a \cdot x, b \cdot (f \cdot (a \cdot x)))) = \text{range } (\lambda x. (a \cdot x, b \cdot (g \cdot (a \cdot x))))$

then have $\text{range } (\lambda x. (a \cdot x, b \cdot (f \cdot (a \cdot x)))) \subseteq \text{range } (\lambda x. (a \cdot x, b \cdot (g \cdot (a \cdot x))))$

by (*rule equalityD1*)

then have $(a \cdot x, b \cdot (f \cdot (a \cdot x))) \in \text{range } (\lambda x. (a \cdot x, b \cdot (g \cdot (a \cdot x))))$

by (*simp add: subset-eq*)

then obtain y where $(a \cdot x, b \cdot (f \cdot (a \cdot x))) = (a \cdot y, b \cdot (g \cdot (a \cdot y)))$

by (*rule rangeE*)

then show $b \cdot (f \cdot (a \cdot x)) = b \cdot (g \cdot (a \cdot x))$

by *clarsimp*

qed

qed

lemma *finite-deflation-cfun-map*:

assumes *finite-deflation* $d1$ **and** *finite-deflation* $d2$

shows *finite-deflation* ($\text{cfun-map} \cdot d1 \cdot d2$)

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* ($\text{cfun-map} \cdot d1 \cdot d2$)

by (*rule deflation-cfun-map*)

have *finite* ($\text{range } (\lambda f. \text{cfun-map} \cdot d1 \cdot d2 \cdot f)$)

using $d1$.*finite-range* $d2$.*finite-range*

by (*rule finite-range-cfun-map*)

then show *finite* $\{f. \text{cfun-map} \cdot d1 \cdot d2 \cdot f = f\}$

by (*rule finite-range-imp-finite-fixes*)

qed

Finite deflations are compact elements of the function space

lemma *finite-deflation-imp-compact*: *finite-deflation* $d \implies$ *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 → 'b) → ('c → 'd) → 'a × 'c → 'b × 'd
where *prod-map* = (λ f g p. (f·(fst p), g·(snd p)))

lemma *prod-map-Pair* [simp]: *prod-map*·f·g·(x, y) = (f·x, g·y)
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) ⊆ 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 ⊆ 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: ccomp1 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  $\{x. u\text{-map}\cdot d\cdot x = x\} \subseteq \text{insert } \perp ((\lambda x. up\cdot x) \cdot \{x. d\cdot x = x\})$ 
  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

default-sort *pcpo*

```

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

```

```

lemma sprod-map-strict [simp]: sprod-map·a·b·⊥ = ⊥
  by (simp add: sprod-map-def)

```

```

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

```

```

lemma sprod-map-spair':  $f\cdot \perp = \perp \implies g\cdot \perp = \perp \implies \text{sprod-map}\cdot f\cdot g\cdot (:x, y:) = (:f\cdot x,$ 
 $g\cdot y:)$ 
  by (cases x = ⊥ ∨ y = ⊥) auto

```

```

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

```

```

lemma sprod-map-map:
   $\llbracket f1\cdot \perp = \perp; g1\cdot \perp = \perp \rrbracket \implies$ 
   $\text{sprod-map}\cdot f1\cdot g1\cdot (\text{sprod-map}\cdot f2\cdot g2\cdot p) =$ 
   $\text{sprod-map}\cdot (\lambda x. f1\cdot (f2\cdot x))\cdot (\lambda x. g1\cdot (g2\cdot x))\cdot p$ 

```

```

proof (induct p)
  case bottom
  then show ?case by simp
next
  case (spair x y)
  then show ?case
    apply (cases f2·x = ⊥, simp)
    apply (cases g2·y = ⊥, 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)  $\sqsubseteq$  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 =  $\perp$ , simp, cases p2.y =  $\perp$ , 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 =  $\perp$ , simp, cases d2.y =  $\perp$ , simp)
      apply (simp add: d1.idem d2.idem)
      done
  qed
  show sprod-map.d1.d2.x  $\sqsubseteq$  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 $\{x. \text{sprod-map} \cdot d1 \cdot d2 \cdot x = x\} \subseteq$
 $\text{insert } \perp ((\lambda(x, y). (:x, y:)) ' (\{x. d1 \cdot x = x\} \times \{y. d2 \cdot y = y\}))$
by (rule subsetI, case-tac x , auto simp add: spair-eq-iff)
then show finite $\{x. \text{sprod-map} \cdot d1 \cdot d2 \cdot x = x\}$
by (rule finite-subset) (simp add: $d1$.finite-fixes $d2$.finite-fixes)
qed

17.5 Map function for strict sums

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

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

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

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

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

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

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

lemma ssum-map-map :
 $\llbracket f1 \cdot \perp = \perp; g1 \cdot \perp = \perp \rrbracket \implies$
 $\text{ssum-map} \cdot f1 \cdot g1 \cdot (\text{ssum-map} \cdot f2 \cdot g2 \cdot p) =$
 $\text{ssum-map} \cdot (\Lambda x. f1 \cdot (f2 \cdot x)) \cdot (\Lambda x. g1 \cdot (g2 \cdot x)) \cdot p$

proof (induct p)
case bottom
then show ?case **by** simp
next
case (sinl x)
then show ?case **by** (cases $f2 \cdot x = \perp$) simp-all
next
case (sinr y)
then show ?case **by** (cases $g2 \cdot y = \perp$) simp-all
qed

lemma *ep-pair-ssum-map*:

assumes *ep-pair* $e1\ p1$ **and** *ep-pair* $e2\ p2$

shows *ep-pair* $(ssum\text{-}map\cdot e1\cdot e2)\ (ssum\text{-}map\cdot p1\cdot p2)$

proof

interpret $e1p1$: *pcpo-ep-pair* $e1\ p1$ **unfolding** *pcpo-ep-pair-def* **by fact**

interpret $e2p2$: *pcpo-ep-pair* $e2\ p2$ **unfolding** *pcpo-ep-pair-def* **by fact**

show $ssum\text{-}map\cdot p1\cdot p2\cdot(ssum\text{-}map\cdot e1\cdot e2\cdot x) = x$ **for** x

by (*induct* x) *simp-all*

show $ssum\text{-}map\cdot e1\cdot e2\cdot(ssum\text{-}map\cdot p1\cdot p2\cdot y) \sqsubseteq y$ **for** y

proof (*induct* y)

case *bottom*

then show *?case* **by** *simp*

next

case (*sinl* x)

then show *?case* **by** (*cases* $p1\cdot x = \perp$) (*simp-all add: e1p1.e-p-below*)

next

case (*sinr* y)

then show *?case* **by** (*cases* $p2\cdot y = \perp$) (*simp-all add: e2p2.e-p-below*)

qed

qed

lemma *deflation-ssum-map*:

assumes *deflation* $d1$ **and** *deflation* $d2$

shows *deflation* $(ssum\text{-}map\cdot d1\cdot d2)$

proof

interpret $d1$: *deflation* $d1$ **by fact**

interpret $d2$: *deflation* $d2$ **by fact**

fix x

show $ssum\text{-}map\cdot d1\cdot d2\cdot(ssum\text{-}map\cdot d1\cdot d2\cdot x) = ssu\text{-}map\cdot d1\cdot d2\cdot x$

proof (*induct* x)

case *bottom*

then show *?case* **by** *simp*

next

case (*sinl* x)

then show *?case* **by** (*cases* $d1\cdot x = \perp$) (*simp-all add: d1.idem*)

next

case (*sinr* y)

then show *?case* **by** (*cases* $d2\cdot y = \perp$) (*simp-all add: d2.idem*)

qed

show $ssum\text{-}map\cdot d1\cdot d2\cdot x \sqsubseteq x$

proof (*induct* x)

case *bottom*

then show *?case* **by** *simp*

next

case (*sinl* x)

then show *?case* **by** (*cases* $d1\cdot x = \perp$) (*simp-all add: d1.below*)

next

case (*sinr* y)

then show ?case by (cases d2.y = \perp) (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} \subseteq
($\lambda x. \text{sinl}.x$) ‘ {x. d1.x = x} \cup
($\lambda x. \text{sinr}.x$) ‘ {x. d2.x = x} \cup { \perp }
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 \rightarrow 'a) \rightarrow ('c \rightarrow 'd) \rightarrow ('a $\rightarrow!$ 'c) \rightarrow ('b $\rightarrow!$ 'd)
where sfun-map = ($\Lambda a b. \text{sfun-abs oo cfun-map}.a.b \text{ oo sfun-rep}$)

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. \perp = \perp and g2. \perp = \perp
shows sfun-map.f1.g1.(sfun-map.f2.g2.p) =
sfun-map.($\Lambda x. f2.(f1.x)$).($\Lambda x. g1.(g2.x)$).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)  $\sqsubseteq$  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 ( $\lambda f. sfun-rep.f$ )
  by (rule inj-onI) (simp add: sfun-eq-iff)
  ultimately have finite (( $\lambda f. sfun-rep.f$ ) - ‘ {f. cfun-map·d1·d2·f = f} )
  by (rule finite-vimageI)
  with  $\langle deflation d1 \rangle \langle deflation d2 \rangle$  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
```

```
default-sort cpo
```

18.1 Continuous case function for unit type

```
definition unit-when :: 'a → unit → 'a
  where unit-when = (λ a . a)
```

translations

```
Λ(). t ⇔ CONST unit-when·t
```

```
lemma unit-when [simp]: unit-when·a·u = a
  by (simp add: unit-when-def)
```

18.2 Continuous version of split function

```
definition csplit :: ('a → 'b → 'c) → ('a × 'b) → 'c
  where csplit = (λ f p. f·(fst p)·(snd p))
```

translations

```
Λ(CONST Pair x y). t ⇔ CONST csplit·(Λ x y. t)
```

```
abbreviation cfst :: 'a × 'b → 'a
  where cfst ≡ Abs-cfun fst
```

```
abbreviation csnd :: 'a × 'b → 'b
  where csnd ≡ Abs-cfun snd
```

18.3 Convert all lemmas to the continuous versions

```
lemma csplit1 [simp]: csplit·f·⊥ = f·⊥·⊥
  by (simp add: csplit-def)
```

```
lemma csplit-Pair [simp]: csplit·f·(x, y) = f·x·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
```

```
default-sort cpo
```

19.1 Chains of finite deflations

```

locale approx-chain =
  fixes approx :: nat  $\Rightarrow$  'a  $\rightarrow$  'a
  assumes chain-approx [simp]: chain ( $\lambda i.$  approx i)
  assumes lub-approx [simp]: ( $\bigsqcup i.$  approx i) = ID
  assumes finite-deflation-approx [simp]:  $\bigwedge i.$  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  $\cdot$  (approx i  $\cdot$  x) = approx i  $\cdot$  x
using deflation-approx by (rule deflation.idem)

```

```

lemma approx-below: approx i  $\cdot$  x  $\sqsubseteq$  x
using deflation-approx by (rule deflation.below)

```

```

lemma finite-range-approx: finite (range ( $\lambda x.$  approx i  $\cdot$  x))
apply (rule finite-deflation.finite-range)
apply (rule finite-deflation-approx)
done

```

```

lemma compact-approx [simp]: compact (approx n  $\cdot$  x)
apply (rule finite-deflation.compact)
apply (rule finite-deflation-approx)
done

```

```

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

```

```

end

```

19.2 Omega-profinite and bifinite domains

```

class bifinite = pcpo +
  assumes bifinite:  $\exists (a::nat \Rightarrow 'a \rightarrow 'a).$  approx-chain a

```

```

class profinite = cpo +
  assumes profinite:  $\exists (a::nat \Rightarrow 'a_{\perp} \rightarrow 'a_{\perp}).$  approx-chain a

```

19.3 Building approx chains

```

lemma approx-chain-iso:
  assumes a: approx-chain a
  assumes [simp]:  $\bigwedge x.$  f  $\cdot$  (g  $\cdot$  x) = x
  assumes [simp]:  $\bigwedge y.$  g  $\cdot$  (f  $\cdot$  y) = y
  shows approx-chain ( $\lambda i.$  f oo a i oo g)
proof –
  have 1: f oo g = ID by (simp add: cfun-eqI)

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

Approx chains for countable discrete types.

```

definition discr-approx :: nat  $\Rightarrow$  'a::countable discr u  $\rightarrow$  'a discr u
  where discr-approx = ( $\lambda i. \Lambda(up\cdot x). \text{if } to\text{-nat } (undiscr\ x) < i \text{ then } up\cdot x \text{ else } \perp$ )

```

```

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

```

```

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

```

19.4 Class instance proofs

```

instance bifinite  $\subseteq$  profinite
proof

```

```

show  $\exists (a::nat \Rightarrow 'a_{\perp} \rightarrow 'a_{\perp}). \text{approx-chain } a$ 
  using bifinite [where  $'a='a$ ]
  by (fast intro!: approx-chain-u-map)
qed

```

```

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

```

Types $'a \rightarrow 'b$ and $'a_{\perp} \rightarrow! 'b$ are isomorphic.

```

definition encode-cfun = ( $\Lambda f. \text{sfun-abs} \cdot (\text{fup} \cdot f)$ )

```

```

definition decode-cfun = ( $\Lambda g x. \text{sfun-rep} \cdot g \cdot (\text{up} \cdot x)$ )

```

```

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

```

```

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

```

```

instance cfun :: (profinite, bifinite) bifinite
proof

```

```

  obtain a ::  $nat \Rightarrow 'a_{\perp} \rightarrow 'a_{\perp}$  where a: approx-chain a
    using profinite ..
  obtain b ::  $nat \Rightarrow 'b \rightarrow 'b$  where b: approx-chain b
    using bifinite ..
  have approx-chain ( $\lambda i. \text{decode-cfun} \text{ oo } \text{sfun-map} \cdot (a \ i) \cdot (b \ i) \text{ oo } \text{encode-cfun}$ )
    using a b by (simp add: approx-chain-iso approx-chain-sfun-map)
  thus  $\exists (a::nat \Rightarrow ('a \rightarrow 'b) \rightarrow ('a \rightarrow 'b)). \text{approx-chain } a$ 
    by - (rule exI)
qed

```

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

```

definition encode-prod-u = ( $\Lambda (\text{up} \cdot (x, y)). (: \text{up} \cdot x, \text{up} \cdot y)$ )

```

```

definition decode-prod-u = ( $\Lambda (: \text{up} \cdot x, \text{up} \cdot y). \text{up} \cdot (x, y)$ )

```

```

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⊥ → 'a⊥ where a: approx-chain a
    using profinite ..
  obtain b :: nat  $\Rightarrow$  'b⊥ → 'b⊥ where b: approx-chain b
    using profinite ..
  have approx-chain (λi. decode-prod-u oo sprod-map.(a i).(b i) oo encode-prod-u)
    using a b by (simp add: approx-chain-iso approx-chain-sprod-map)
  thus ∃(a::nat  $\Rightarrow$  ('a × 'b)⊥ → ('a × 'b)⊥). approx-chain a
    by - (rule exI)
qed

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma approx-chain-unit: approx-chain (⊥ :: nat  $\Rightarrow$  unit → unit)

```


by (*simp add: approx-chain-def cfun-eq-iff finite-deflation-bottom*)

instance *unit* :: *bifinite*

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

instance *discr* :: (*countable*) *profinite*

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

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

proof

note [*simp*] = *cont-Abs-lift cont-Rep-lift Rep-lift-inverse Abs-lift-inverse*

obtain *a* :: *nat* \Rightarrow (*'a discr*)_⊥ \rightarrow (*'a discr*)_⊥ **where** *a*: *approx-chain a*
 using *profinite ..*

hence *approx-chain* ($\lambda i. (\Lambda y. \text{Abs-lift } y) \text{ oo } a \text{ i oo } (\Lambda x. \text{Rep-lift } x)$)

by (*rule approx-chain-iso*) *simp-all*

thus $\exists (a::\text{nat} \Rightarrow 'a \text{ lift} \rightarrow 'a \text{ lift}). \text{approx-chain } a$

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* \Rightarrow *'a* \Rightarrow *bool* (**infix** \preceq 50)

assumes *r-refl*: $x \preceq x$

assumes *r-trans*: $\llbracket x \preceq y; y \preceq z \rrbracket \Longrightarrow x \preceq z$

begin

definition

ideal :: *'a set* \Rightarrow *bool* **where**

ideal A = $((\exists x. x \in A) \wedge (\forall x \in A. \forall y \in A. \exists z \in A. x \preceq z \wedge y \preceq z) \wedge$
 $(\forall x y. x \preceq y \longrightarrow y \in A \longrightarrow x \in A))$

lemma *idealI*:

assumes $\exists x. x \in A$

assumes $\bigwedge x y. \llbracket x \in A; y \in A \rrbracket \Longrightarrow \exists z \in A. x \preceq z \wedge y \preceq z$

assumes $\bigwedge x y. \llbracket x \preceq y; y \in A \rrbracket \Longrightarrow x \in A$

shows *ideal A*

unfolding *ideal-def* **using** *assms* **by** *fast*

lemma *idealD1*:

ideal A $\Longrightarrow \exists x. x \in A$

unfolding *ideal-def* **by** *fast*

lemma *idealD2*:

$\llbracket \text{ideal } A; x \in A; y \in A \rrbracket \implies \exists z \in A. x \preceq z \wedge y \preceq z$

unfolding *ideal-def* **by** *fast*

lemma *idealD3*:

$\llbracket \text{ideal } A; x \preceq y; y \in A \rrbracket \implies x \in A$

unfolding *ideal-def* **by** *fast*

lemma *ideal-principal*: *ideal* $\{x. x \preceq z\}$

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*: $\exists A. A \in \{A. \text{ideal } A\}$

by (*fast intro: ideal-principal*)

The set of ideals is a cpo

lemma *ideal-UN*:

fixes $A :: \text{nat} \Rightarrow 'a \text{ set}$

assumes *ideal-A*: $\bigwedge i. \text{ideal } (A \ i)$

assumes *chain-A*: $\bigwedge i \ j. i \leq j \implies A \ i \subseteq A \ j$

shows *ideal* $(\bigcup i. A \ i)$

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 $Abs :: 'a \text{ set} \Rightarrow 'b::\text{below}$

assumes *type*: *type-definition* *Rep* $Abs \ \{S. \text{ideal } S\}$

assumes *below*: $\bigwedge x \ y. x \sqsubseteq y \iff \text{Rep } x \subseteq \text{Rep } y$

shows *OFCLASS*('b, *po-class*)

apply (*intro-classes, unfold below*)

apply (*rule subset-refl*)

```

apply (erule (1) subset-trans)
apply (rule type-definition.Rep-inject [OF type, THEN iffD1])
apply (erule (1) subset-antisym)
done

```

lemma

```

fixes Abs :: 'a set  $\Rightarrow$  'b::po
assumes type: type-definition Rep Abs {S. ideal S}
assumes below:  $\bigwedge x y. x \sqsubseteq y \longleftrightarrow \text{Rep } x \subseteq \text{Rep } y$ 
assumes S: chain S
shows typedef-ideal-lub:  $\text{range } S \ll\!| \text{Abs } (\bigcup i. \text{Rep } (S i))$ 
and typedef-ideal-rep-lub:  $\text{Rep } (\bigsqcup i. S i) = (\bigcup i. \text{Rep } (S i))$ 
proof –
have 1: ideal  $(\bigcup i. \text{Rep } (S i))$ 
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:  $\text{Rep } (\text{Abs } (\bigcup i. \text{Rep } (S i))) = (\bigcup i. \text{Rep } (S i))$ 
by (simp add: type-definition.Abs-inverse [OF type])
show 3:  $\text{range } S \ll\!| \text{Abs } (\bigcup i. \text{Rep } (S i))$ 
apply (rule is-lubI)
apply (rule is-ubI)
apply (simp add: below 2, fast)
apply (simp add: below 2 is-ub-def, fast)
done
hence 4:  $(\bigsqcup i. S i) = \text{Abs } (\bigcup i. \text{Rep } (S i))$ 
by (rule lub-eqI)
show 5:  $\text{Rep } (\bigsqcup i. S i) = (\bigcup i. \text{Rep } (S i))$ 
by (simp add: 4 2)
qed

```

lemma typedef-ideal-cpo:

```

fixes Abs :: 'a set  $\Rightarrow$  'b::po
assumes type: type-definition Rep Abs {S. ideal S}
assumes below:  $\bigwedge x y. x \sqsubseteq y \longleftrightarrow \text{Rep } x \subseteq \text{Rep } y$ 
shows OFCLASS('b, cpo-class)
by standard (rule exI, erule typedef-ideal-lub [OF type below])

```

end

```

interpretation below: preorder below :: 'a::po  $\Rightarrow$  'a  $\Rightarrow$  bool
apply unfold-locales
apply (rule below-refl)
apply (erule (1) below-trans)
done

```

20.2 Lemmas about least upper bounds

lemma *is-ub-the-lub-ex*: $[\exists u. S \ll\!| u; x \in S] \implies x \sqsubseteq \text{lub } S$
apply (*erule exE*, *drule is-lub-lub*)
apply (*drule is-lubD1*)
apply (*erule (1) is-ubD*)
done

lemma *is-lub-the-lub-ex*: $[\exists u. S \ll\!| u; S \ll\!| x] \implies \text{lub } S \sqsubseteq x$
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::cpo
fixes *rep* :: 'b::cpo \Rightarrow 'a::type set
assumes *ideal-rep*: $\bigwedge x. \text{ideal } (\text{rep } x)$
assumes *rep-lub*: $\bigwedge Y. \text{chain } Y \implies \text{rep } (\bigsqcup i. Y i) = (\bigcup i. \text{rep } (Y i))$
assumes *rep-principal*: $\bigwedge a. \text{rep } (\text{principal } a) = \{b. b \preceq a\}$
assumes *belowI*: $\bigwedge x y. \text{rep } x \subseteq \text{rep } y \implies x \sqsubseteq y$
assumes *countable*: $\exists f::'a \Rightarrow \text{nat. inj } f$
begin

lemma *rep-mono*: $x \sqsubseteq y \implies \text{rep } x \subseteq \text{rep } y$
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*: $x \sqsubseteq y \longleftrightarrow \text{rep } x \subseteq \text{rep } y$
by (*rule iffI [OF rep-mono belowI]*)

lemma *principal-below-iff-mem-rep*: $\text{principal } a \sqsubseteq x \longleftrightarrow a \in \text{rep } x$
unfolding *below-def rep-principal*
by (*auto intro: r-refl elim: idealD3 [OF ideal-rep]*)

lemma *principal-below-iff [simp]*: $\text{principal } a \sqsubseteq \text{principal } b \longleftrightarrow a \preceq b$
by (*simp add: principal-below-iff-mem-rep rep-principal*)

lemma *principal-eq-iff*: $\text{principal } a = \text{principal } b \longleftrightarrow a \preceq b \wedge b \preceq a$
unfolding *po-eq-conv [where 'a='b] principal-below-iff ..*

lemma *eq-iff*: $x = y \longleftrightarrow \text{rep } x = \text{rep } y$
unfolding *po-eq-conv below-def by auto*

lemma *principal-mono*: $a \preceq b \implies \text{principal } a \sqsubseteq \text{principal } b$
by (*simp only: principal-below-iff*)

lemma *ch2ch-principal* [*simp*]:
 $\forall i. Y\ i \preceq Y\ (Suc\ i) \implies chain\ (\lambda i. principal\ (Y\ i))$
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** $\forall i. Y\ i \preceq Y\ (Suc\ i)$ **and** $x = (\bigsqcup i. principal\ (Y\ i))$

proof –

obtain *count* :: '*a* \Rightarrow *nat* **where** *inj*: *inj* *count*
using *countable* ..
define *enum* **where** *enum* *i* = (*THE* *a*. *count* *a* = *i*) **for** *i*
have *enum-count* [*simp*]: $\bigwedge x. enum\ (count\ x) = x$
unfolding *enum-def* **by** (*simp* *add*: *inj-eq* [*OF* *inj*])
define *a* **where** *a* = (*LEAST* *i*. *enum* *i* \in *rep* *x*)
define *b* **where** *b* *i* = (*LEAST* *j*. *enum* *j* \in *rep* *x* \wedge \neg *enum* *j* \preceq *enum* *i*) **for** *i*
define *c* **where** *c* *i* *j* = (*LEAST* *k*. *enum* *k* \in *rep* *x* \wedge *enum* *i* \preceq *enum* *k* \wedge *enum* *j* \preceq *enum* *k*) **for** *i* *j*
define *P* **where** *P* *i* \longleftrightarrow ($\exists j. enum\ j \in rep\ x \wedge \neg enum\ j \preceq enum\ i$) **for** *i*
define *X* **where** *X* = *rec-nat* *a* ($\lambda n\ i. if\ P\ i\ then\ c\ i\ (b\ i)\ else\ i$)
have *X-0*: *X* 0 = *a* **unfolding** *X-def* **by** *simp*
have *X-Suc*: $\bigwedge n. X\ (Suc\ n) = (if\ P\ (X\ n)\ then\ c\ (X\ n)\ (b\ (X\ n))\ else\ X\ n)$
unfolding *X-def* **by** *simp*
have *a-mem*: *enum* *a* \in *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*: $\bigwedge i. P\ i \implies enum\ i \in rep\ x$
 $\implies enum\ (b\ i) \in rep\ x \wedge \neg enum\ (b\ i) \preceq enum\ i$
unfolding *P-def* *b-def* **by** (*erule* *LeastI2-ex*, *simp*)
have *c*: $\bigwedge i\ j. enum\ i \in rep\ x \implies enum\ j \in rep\ x$
 $\implies enum\ (c\ i\ j) \in rep\ x \wedge enum\ i \preceq enum\ (c\ i\ j) \wedge enum\ j \preceq enum\ (c\ i\ j)$
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*) \in *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:  $\bigwedge n. \text{enum } (X n) \preceq \text{enum } (X (Suc n))$ 
  apply (clarsimp simp add: X-Suc r-refl)
  apply (simp add: b c X-mem)
  done
have less-b:  $\bigwedge n i. n < b i \implies \text{enum } n \in \text{rep } x \implies \text{enum } n \preceq \text{enum } i$ 
  unfolding b-def by (drule not-less-Least, simp)
have X-covers:  $\forall k \leq n. \text{enum } k \in \text{rep } x \longrightarrow \text{enum } k \preceq \text{enum } (X n)$  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:  $\forall i. \text{enum } (X i) \preceq \text{enum } (X (Suc i))$ 
  by (simp add: X-chain)
have x = ( $\bigsqcup n. \text{principal } (\text{enum } (X n))$ )
  apply (simp add: eq-iff rep-lub 1 rep-principal)
  apply auto
  subgoal for a
    apply (subgoal-tac  $\exists i. a = \text{enum } i$ , 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:  $\bigwedge a. P$  (principal a)
  shows P x
apply (rule obtain-principal-chain [of x])
apply (simp add: admD [OF adm] P)
done

```

```

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

```

20.4 Defining functions in terms of basis elements

definition

```

extension :: ('a::type  $\Rightarrow$  'c::cpo)  $\Rightarrow$  'b  $\rightarrow$  'c where
extension = ( $\lambda f. (\bigwedge x. \text{lub } (f \text{ ` rep } x))$ )

```

lemma extension-lemma:

```

fixes f :: 'a::type  $\Rightarrow$  'c::cpo
assumes f-mono:  $\bigwedge a b. a \preceq b \implies f a \sqsubseteq f b$ 
shows  $\exists u. f \text{ ` rep } x \ll\mid u$ 
proof -
  obtain Y where Y:  $\forall i. Y i \preceq Y (\text{Suc } i)$ 
  and x:  $x = (\bigsqcup i. \text{principal } (Y i))$ 
  by (rule obtain-principal-chain [of x])
  have chain: chain ( $\lambda i. f (Y i)$ )
  by (rule chainI, simp add: f-mono Y)
  have rep-x:  $\text{rep } x = (\bigcup n. \{a. a \preceq Y n\})$ 
  by (simp add: x rep-lub Y rep-principal)
  have f ` rep x  $\ll\mid (\bigsqcup n. f (Y n))$ 
  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::cpo
  assumes f-mono:  $\bigwedge a b. a \preceq b \implies f a \sqsubseteq f b$ 
  shows extension f.x = lub (f ` rep x)
unfolding extension-def
proof (rule beta-cfun)
  have lub:  $\bigwedge x. \exists u. f ` rep x \ll u$ 
    using f-mono by (rule extension-lemma)
  show cont: cont ( $\lambda x. \text{lub } (f ` \text{rep } x)$ )
    apply (rule contI2)
    apply (rule monofunI)
    apply (rule is-lub-the-lub-ex [OF lub ub-imageI])
    apply (rule is-ub-the-lub-ex [OF lub imageI])
    apply (erule (1) subsetD [OF rep-mono])
    apply (rule is-lub-the-lub-ex [OF lub ub-imageI])
    apply (simp add: rep-lub, clarify)
    apply (erule rev-below-trans [OF is-ub-the-lub])
    apply (erule is-ub-the-lub-ex [OF lub imageI])
  done
qed

```

```

lemma extension-principal:
  fixes f :: 'a::type ⇒ 'c::cpo
  assumes f-mono:  $\bigwedge a b. a \preceq b \implies f a \sqsubseteq f b$ 
  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:  $\bigwedge a b. a \preceq b \implies f a \sqsubseteq f b$ 
  assumes g-mono:  $\bigwedge a b. a \preceq b \implies g a \sqsubseteq g b$ 
  assumes below:  $\bigwedge a. f a \sqsubseteq g a$ 
  shows extension f  $\sqsubseteq$  extension g
  apply (rule cfun-belowI)
  apply (simp only: extension-beta f-mono g-mono)
  apply (rule is-lub-the-lub-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-the lub-ex)
  apply (rule extension-lemma, erule g-mono)
  apply (erule imageI)
done
done

```

lemma *cont-extension*:

```

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

```

end

lemma (in preorder) *typedef-ideal-completion*:

```

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

```

end

21 A universal bifinite domain

theory *Universal*

imports *Bifinite Completion HOL-Library.Nat-Bijection*

begin

no-notation *binomial* (infixl choose 65)

21.1 Basis for universal domain

21.1.1 Basis datatype

type-synonym *ubasis* = *nat*

definition

node :: *nat* \Rightarrow *ubasis* \Rightarrow *ubasis set* \Rightarrow *ubasis*

where

node *i a S* = *Suc* (*prod-encode* (*i*, *prod-encode* (*a*, *set-encode* *S*)))

lemma *node-not-0* [*simp*]: *node* *i a S* \neq 0

unfolding *node-def* by *simp*

lemma *node-gt-0* [*simp*]: 0 < *node* *i a S*

unfolding *node-def* by *simp*

lemma *node-inject* [*simp*]:

\llbracket *finite* *S*; *finite* *T* \rrbracket

\implies *node* *i a S* = *node* *j b T* \longleftrightarrow *i* = *j* \wedge *a* = *b* \wedge *S* = *T*

unfolding *node-def* by (*simp* add: *prod-encode-eq set-encode-eq*)

lemma *node-gt0*: *i* < *node* *i a S*

unfolding *node-def less-Suc-eq-le*

by (*rule le-prod-encode-1*)

lemma *node-gt1*: *a* < *node* *i a S*

unfolding *node-def less-Suc-eq-le*

by (*rule order-trans* [*OF le-prod-encode-1 le-prod-encode-2*])

lemma *nat-less-power2*: *n* < $2^{\wedge}n$

by (*fact less-exp*)

lemma *node-gt2*: \llbracket *finite* *S*; *b* \in *S* $\rrbracket \implies$ *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* ((\wedge) 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*:

$\llbracket \text{fst} (\text{prod-decode } x) = a; \text{snd} (\text{prod-decode } x) = b \rrbracket \implies x = \text{prod-encode } (a, b)$
by (*erule subst*, *erule subst*, *simp*)

lemma *node-cases*:

assumes 1: $x = 0 \implies P$
assumes 2: $\bigwedge i a S. \llbracket \text{finite } S; x = \text{node } i a S \rrbracket \implies P$
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: $\bigwedge i a S. \llbracket P a; \text{finite } S; \forall b \in S. P b \rrbracket \implies P (\text{node } i a S)$
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 :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$

where

ubasis-le-refl: $\text{ubasis-le } a a$

| *ubasis-le-trans*:

$\llbracket \text{ubasis-le } a b; \text{ubasis-le } b c \rrbracket \implies \text{ubasis-le } a c$

| *ubasis-le-lower*:

$\text{finite } S \implies \text{ubasis-le } a (\text{node } i a S)$

| *ubasis-le-upper*:

$\llbracket \text{finite } S; b \in S; \text{ubasis-le } a b \rrbracket \implies \text{ubasis-le } (\text{node } i a S) b$

lemma *ubasis-le-minimal*: $\text{ubasis-le } 0 x$

apply (*induct x rule: node-induct*)

apply (*rule ubasis-le-refl*)

apply (*erule ubasis-le-trans*)

apply (*erule ubasis-le-lower*)

done

interpretation *u*dom: 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* \Rightarrow *bool*) \Rightarrow *ubasis* \Rightarrow *ubasis*
where
ubasis-until *P* 0 = 0
| *finite S* \Longrightarrow *ubasis-until* *P* (*node i a S*) =
(*if P (node i a S) then node i a S else ubasis-until P a*)
apply *clarify*
apply (*rule-tac x=b in node-cases*)
apply *simp-all*
done

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

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

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

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

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

lemma *ubasis-until-0*:
 $\forall x. x \neq 0 \longrightarrow \neg P\ x \Longrightarrow \text{ubasis-until } P\ x = 0$
by (*induct x rule: node-induct*) *simp-all*

lemma *ubasis-until-less*: *ubasis-le* (*ubasis-until* *P* *x*) *x*
apply (*induct x rule: node-induct*)
apply (*simp add: ubasis-le-refl*)
apply (*simp add: ubasis-le-refl*)
apply (*rule impI*)
apply (*erule ubasis-le-trans*)

apply (*erule ubasis-le-lower*)
done

lemma *ubasis-until-chain*:
assumes $PQ: \bigwedge x. P x \implies Q x$
shows *ubasis-le* (*ubasis-until* $P x$) (*ubasis-until* $Q x$)
apply (*induct x rule: node-induct*)
apply (*simp add: ubasis-le-refl*)
apply (*simp add: ubasis-le-refl*)
apply (*simp add: PQ*)
apply *clarify*
apply (*rule ubasis-le-trans*)
apply (*rule ubasis-until-less*)
apply (*erule ubasis-le-lower*)
done

lemma *ubasis-until-mono*:
assumes $\bigwedge i a S b. \llbracket \text{finite } S; P (\text{node } i a S); b \in S; \text{ubasis-le } a b \rrbracket \implies P b$
shows *ubasis-le* $a b \implies \text{ubasis-le} (\text{ubasis-until } P a) (\text{ubasis-until } P b)$
proof (*induct set: ubasis-le*)
case (*ubasis-le-refl a*) **show** *?case by* (*rule ubasis-le.ubasis-le-refl*)
next
case (*ubasis-le-trans a b c*) **thus** *?case by* – (*rule ubasis-le.ubasis-le-trans*)
next
case (*ubasis-le-lower S a i*) **thus** *?case*
apply (*clarsimp simp add: ubasis-le-refl*)
apply (*rule ubasis-le-trans [OF ubasis-until-less]*)
apply (*erule ubasis-le.ubasis-le-lower*)
done
next
case (*ubasis-le-upper S b a i*) **thus** *?case*
apply *clarsimp*
apply (*subst ubasis-until-same*)
apply (*erule (3) assms*)
apply (*erule (2) ubasis-le.ubasis-le-upper*)
done
qed

lemma *finite-range-ubasis-until*:
 $\text{finite } \{x. P x\} \implies \text{finite} (\text{range} (\text{ubasis-until } P))$
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 *uodom* = $\{S. \text{uodom.ideal } S\}$
by (*rule uodom.ex-ideal*)

instantiation *udom* :: *below*
begin

definition

$x \sqsubseteq y \iff \text{Rep-udom } x \subseteq \text{Rep-udom } y$

instance ..
end

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

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

definition

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

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

interpretation *udom*:

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

Universal domain is pointed

lemma *udom-minimal*: *udom-principal* 0 $\sqsubseteq x$
apply (*induct x rule: udom.principal-induct*)
apply (*simp, simp add: ubasis-le-minimal*)
done

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

lemma *inst-udom-pcpo*: $\perp = \text{udom-principal } 0$
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]:

$compact\ x \implies Rep-compact-basis\ (Abs-compact-basis\ x) = x$

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*)

definition

approximants :: 'a \Rightarrow 'a *compact-basis set* **where**

approximants = $(\lambda x. \{a. Rep-compact-basis\ a \sqsubseteq x\})$

definition

compact-bot :: 'a::pcpo *compact-basis* **where**

compact-bot = *Abs-compact-basis* \perp

lemma *Rep-compact-bot* [simp]: *Rep-compact-basis compact-bot* = \perp

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

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

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

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 \Rightarrow 'a::bifinite \rightarrow 'a

begin

21.4.1 Choosing a maximal element from a finite set

lemma *finite-has-maximal*:

fixes *A* :: 'a *compact-basis set*

shows $\llbracket finite\ A; A \neq \{\} \rrbracket \implies \exists x \in A. \forall y \in A. x \sqsubseteq y \longrightarrow x = y$

proof (*induct rule: finite-ne-induct*)

case (*singleton x*)

```

  show ?case by simp
next
case (insert a A)
from ⟨ $\exists x \in A. \forall y \in A. x \sqsubseteq y \longrightarrow x = y$ ⟩
obtain x where x:  $x \in A$ 
  and x-eq:  $\bigwedge y. \llbracket y \in A; x \sqsubseteq y \rrbracket \Longrightarrow x = y$  by fast
show ?case
proof (intro bexI ballI impI)
  fix y
  assume  $y \in \text{insert } a \ A$  and (if  $x \sqsubseteq a$  then  $a$  else  $x$ )  $\sqsubseteq y$ 
  thus (if  $x \sqsubseteq a$  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  $x \sqsubseteq a$  then  $a$  else  $x$ )  $\in \text{insert } a \ A$ 
  by (simp add: x)
qed
qed

```

definition

choose :: 'a compact-basis set \Rightarrow 'a compact-basis

where

choose $A = (\text{SOME } x. x \in \{x \in A. \forall y \in A. x \sqsubseteq y \longrightarrow x = y\})$

lemma *choose-lemma*:

$\llbracket \text{finite } A; A \neq \{\} \rrbracket \Longrightarrow \text{choose } A \in \{x \in A. \forall y \in A. x \sqsubseteq y \longrightarrow x = y\}$

unfolding *choose-def*

apply (rule someI-ex)

apply (frule (1) finite-has-maximal, fast)

done

lemma *maximal-choose*:

$\llbracket \text{finite } A; y \in A; \text{choose } A \sqsubseteq y \rrbracket \Longrightarrow \text{choose } A = y$

apply (cases $A = \{\}$, simp)

apply (frule (1) choose-lemma, simp)

done

lemma *choose-in*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \Longrightarrow \text{choose } A \in A$

by (frule (1) choose-lemma, simp)

function

choose-pos :: 'a compact-basis set \Rightarrow 'a compact-basis \Rightarrow 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  $y = \text{choose } A$ )
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 ::  $\text{nat} \Rightarrow 'a \text{ compact-basis} \Rightarrow 'a \text{ compact-basis}$  where
cb-take 0 = ( $\lambda x. \text{compact-bot}$ )
| cb-take (Suc  $n$ ) = ( $\lambda a. \text{Abs-compact-basis} (\text{approx } n \cdot (\text{Rep-compact-basis } a))$ )

```

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

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

lemma *Rep-cb-take*:

```

 $\text{Rep-compact-basis} (\text{cb-take} (\text{Suc } n) a) = \text{approx } n \cdot (\text{Rep-compact-basis } a)$ 
by (simp add: cb-take.simps(2))

```

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

lemma *cb-take-covers*: $\exists n. \text{cb-take } n x = x$
apply (*subgoal-tac* $\exists n. \text{cb-take} (\text{Suc } n) x = x$, *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*: $\text{cb-take } n x \sqsubseteq x$
unfolding *compact-le-def*
by (*cases* n , *simp*, *simp add: Rep-cb-take approx-below*)

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

lemma *cb-take-mono*: $x \sqsubseteq y \Longrightarrow \text{cb-take } n x \sqsubseteq \text{cb-take } n y$
unfolding *compact-le-def*
by (*cases* n , *simp*, *simp add: Rep-cb-take monofun-cfun-arg*)

lemma *cb-take-chain-le*: $m \leq n \Longrightarrow \text{cb-take } m x \sqsubseteq \text{cb-take } n x$
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 (λx. 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

$rank :: 'a \text{ compact-basis} \Rightarrow nat$

where

$rank\ x = (LEAST\ n.\ cb\text{-take}\ n\ x = x)$

```

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  $\implies$  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  $\implies$  rank x ≤ n
unfolding rank-def by (rule Least-le)

```

```

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

```

definition

$rank\text{-le} :: 'a \text{ compact-basis} \Rightarrow 'a \text{ compact-basis set}$

where

$rank\text{-le}\ x = \{y.\ rank\ y \leq rank\ x\}$

definition

rank-lt :: 'a compact-basis \Rightarrow 'a compact-basis set

where

rank-lt $x = \{y. \text{rank } y < \text{rank } x\}$

definition

rank-eq :: 'a compact-basis \Rightarrow 'a compact-basis set

where

rank-eq $x = \{y. \text{rank } y = \text{rank } x\}$

lemma *rank-eq-cong*: $\text{rank } x = \text{rank } y \Longrightarrow \text{rank-eq } x = \text{rank-eq } y$

unfolding *rank-eq-def* **by** *simp*

lemma *rank-lt-cong*: $\text{rank } x = \text{rank } y \Longrightarrow \text{rank-lt } x = \text{rank-lt } y$

unfolding *rank-lt-def* **by** *simp*

lemma *rank-eq-subset*: $\text{rank-eq } x \subseteq \text{rank-le } x$

unfolding *rank-eq-def* *rank-le-def* **by** *auto*

lemma *rank-lt-subset*: $\text{rank-lt } x \subseteq \text{rank-le } x$

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** $B = \text{range } (\text{cb-take } (\text{rank } x))$])

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*: $\text{rank-lt } x \cap \text{rank-eq } x = \{\}$

unfolding *rank-lt-def* *rank-eq-def* *rank-le-def* **by** *auto*

lemma *rank-lt-Un-rank-eq*: $\text{rank-lt } x \cup \text{rank-eq } x = \text{rank-le } x$

unfolding *rank-lt-def* *rank-eq-def* *rank-le-def* **by** *auto*

21.4.4 Sequencing basis elements**definition**

place :: 'a compact-basis \Rightarrow nat

where

$place\ x = card\ (rank\text{-}lt\ x) + choose\text{-}pos\ (rank\text{-}eq\ x)\ x$

lemma *place-bounded*: $place\ x < card\ (rank\text{-}le\ x)$
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*: $card\ (rank\text{-}lt\ x) \leq place\ x$
unfolding *place-def* **by** *simp*

lemma *place-rank-mono*:
fixes $x\ y :: 'a\ compact\text{-}basis$
shows $rank\ x < rank\ y \implies place\ x < place\ y$
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*: $place\ x = place\ y \implies x = y$
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

$sub :: 'a\ compact\text{-}basis \Rightarrow 'a\ compact\text{-}basis$
where

$sub\ x = (case\ rank\ x\ of\ 0 \Rightarrow compact-bot \mid Suc\ k \Rightarrow cb-take\ k\ x)$

lemma *rank-sub-less*: $x \neq compact-bot \implies rank\ (sub\ x) < rank\ x$
unfolding *sub-def*
apply (*cases rank x, simp*)
apply (*simp add: less-Suc-eq-le*)
apply (*rule rank-leI*)
apply (*rule cb-take-idem*)
done

lemma *place-sub-less*: $x \neq compact-bot \implies place\ (sub\ x) < place\ x$
apply (*rule place-rank-mono*)
apply (*erule rank-sub-less*)
done

lemma *sub-below*: $sub\ x \sqsubseteq x$
unfolding *sub-def* **by** (*cases rank x, simp-all add: cb-take-less*)

lemma *rank-less-imp-below-sub*: $\llbracket x \sqsubseteq y; rank\ x < rank\ y \rrbracket \implies x \sqsubseteq sub\ y$
unfolding *sub-def*
apply (*cases rank y, simp*)
apply (*simp add: less-Suc-eq-le*)
apply (*subgoal-tac cb-take nat x \sqsubseteq cb-take nat y*)
apply (*simp add: rank-leD*)
apply (*erule cb-take-mono*)
done

function *basis-emb* :: 'a *compact-basis* \Rightarrow *ubasis*
where *basis-emb* $x = (if\ x = compact-bot\ then\ 0\ else$
node (*place* x) (*basis-emb* (*sub* x))
(*basis-emb* ' { $y.$ *place* $y < place\ x \wedge x \sqsubseteq y$ })
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* $y < place$
 $x \wedge x \sqsubseteq y$ })
if $x \neq compact-bot$
using *that basis-emb.simps* [*of x*] **by** *simp*

lemma *basis-emb-eq-0-iff* [*simp*]:

basis-emb $x = 0 \iff x = \text{compact-bot}$
by (*cases* $x = \text{compact-bot}$) (*simp-all* *add*: *basis-emb-rec*)

lemma *fin1*: *finite* $\{y. \text{place } y < \text{place } x \wedge x \sqsubseteq y\}$
apply (*subst* *Collect-conj-eq*)
apply (*rule* *finite-Int*)
apply (*rule* *disjI1*)
apply (*subgoal-tac* *finite* (*place* $\neg \{n. n < \text{place } x\}$), *simp*)
apply (*rule* *finite-vimageI* [*OF* - *inj-place*])
apply (*simp* *add*: *lessThan-def* [*symmetric*])
done

lemma *fin2*: *finite* (*basis-emb* $\{y. \text{place } y < \text{place } x \wedge x \sqsubseteq y\}$)
by (*rule* *finite-imageI* [*OF* *fin1*])

lemma *rank-place-mono*:
 $\llbracket \text{place } x < \text{place } y; x \sqsubseteq y \rrbracket \implies \text{rank } x < \text{rank } y$
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*:
 $x \sqsubseteq y \implies \text{ubasis-le } (\text{basis-emb } x) (\text{basis-emb } y)$
proof (*induct* *max* (*place* x) (*place* y) *arbitrary*: $x \ y$ *rule*: *less-induct*)
case *less*
show *?case* **proof** (*rule* *linorder-cases*)
assume *place* $x < \text{place } y$
then have *rank* $x < \text{rank } y$
using $\langle x \sqsubseteq y \rangle$ **by** (*rule* *rank-place-mono*)
with $\langle \text{place } x < \text{place } y \rangle$ **show** *?case*
apply (*case-tac* $y = \text{compact-bot}$, *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* $\langle x \sqsubseteq y \rangle$])
done
next
assume *place* $x = \text{place } y$
hence $x = y$ **by** (*rule* *place-eqD*)
thus *?case* **by** (*simp* *add*: *ubasis-le-refl*)
next

```

assume place  $x > \textit{place } y$ 
with  $\langle x \sqsubseteq y \rangle$  show ?case
  apply (case-tac  $x = \textit{compact-bot}$ , 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  $x = \textit{basis-emb } y$ 
  then show  $x = y$ 
    by (cases  $x = \textit{compact-bot} \vee y = \textit{compact-bot}$ ) (auto simp add: basis-emb-rec
fin2 place-eqD)
qed

definition
  basis-prj :: ubasis  $\Rightarrow$  'a compact-basis
where
  basis-prj  $x = \textit{inv basis-emb}$ 
  (ubasis-until ( $\lambda x. x \in \textit{range (basis-emb :: 'a compact-basis} \Rightarrow \textit{ubasis})}$ )  $x$ )

lemma basis-prj-basis-emb:  $\bigwedge x. \textit{basis-prj (basis-emb } x) = x$ 
unfolding basis-prj-def
  apply (subst ubasis-until-same)
  apply (rule rangeI)
  apply (rule inv-f-f)
  apply (rule inj-basis-emb)
done

lemma basis-prj-node:
  [finite S; node i a S  $\notin \textit{range (basis-emb :: 'a compact-basis} \Rightarrow \textit{nat})}$ ]
   $\Longrightarrow \textit{basis-prj (node } i \textit{ a } S) = (\textit{basis-prj } a :: \textit{'a compact-basis})$ 
unfolding basis-prj-def by simp

lemma basis-prj-0: basis-prj 0 = compact-bot
apply (subst basis-emb-compact-bot [symmetric])
apply (rule basis-prj-basis-emb)
done

lemma node-eq-basis-emb-iff:
  finite S  $\Longrightarrow \textit{node } i \textit{ a } S = \textit{basis-emb } x \iff$ 

```



```

    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  $\exists x. x \in \text{approximants } w \dots$ 
next
  fix  $x y :: 'a \text{ compact-basis}$ 
  assume  $x: x \in \text{approximants } w$  and  $y: y \in \text{approximants } w$ 
  obtain  $i$  where  $i: \text{approx } i. (\text{Rep-compact-basis } x) = \text{Rep-compact-basis } x$ 
    using compact-eq-approx Rep-compact-basis' by fast
  obtain  $j$  where  $j: \text{approx } j. (\text{Rep-compact-basis } y) = \text{Rep-compact-basis } y$ 
    using compact-eq-approx Rep-compact-basis' by fast
  let  $?z = \text{Abs-compact-basis } (\text{approx } (\max i j). w)$ 
  have  $?z \in \text{approximants } w$ 
    by (simp add: approximants-def approx-below)
  moreover from  $x y$  have  $x \sqsubseteq ?z \wedge y \sqsubseteq ?z$ 
    by (simp add: approximants-def compact-le-def)
    (metis  $i j$  monofun-cfun chain-mono chain-approx max.cobounded1 max.cobounded2)
  ultimately show  $\exists z \in \text{approximants } w. x \sqsubseteq z \wedge y \sqsubseteq z \dots$ 
next
  fix  $x y :: 'a \text{ compact-basis}$ 
  assume  $x \sqsubseteq y$   $y \in \text{approximants } w$  thus  $x \in \text{approximants } w$ 
    unfolding approximants-def compact-le-def
    by (auto elim: below-trans)
qed
next
  fix  $Y :: \text{nat} \Rightarrow 'a$ 
  assume chain  $Y$ 
  thus approximants  $(\bigsqcup i. Y i) = (\bigcup i. \text{approximants } (Y i))$ 
    unfolding approximants-def
    by (auto simp add: compact-below-lub-iff)
next
  fix  $a :: 'a \text{ compact-basis}$ 
  show approximants  $(\text{Rep-compact-basis } a) = \{b. b \sqsubseteq a\}$ 
    unfolding approximants-def compact-le-def ..
next
  fix  $x y :: 'a$ 
  assume approximants  $x \subseteq \text{approximants } y$ 
  hence  $\forall z. \text{compact } z \longrightarrow z \sqsubseteq x \longrightarrow z \sqsubseteq y$ 
    by (simp add: approximants-def subset-eq)
    (metis Abs-compact-basis-inverse')
  hence  $(\bigsqcup i. \text{approx } i.x) \sqsubseteq y$ 
    by (simp add: lub-below approx-below)
  thus  $x \sqsubseteq y$ 
    by (simp add: lub-distrib)
next
  show  $\exists f :: 'a \text{ compact-basis} \Rightarrow \text{nat. inj } f$ 
    by (rule exI, rule inj-place)
qed
end

```

interpretation *compact-basis*:

ideal-completion below Rep-compact-basis

approximants :: 'a::bifinite \Rightarrow 'a *compact-basis set*

proof –

obtain $a :: \text{nat} \Rightarrow 'a \rightarrow 'a$ **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 \Rightarrow -)*

by (*rule bifinite-approx-chain.ideal-completion*)

qed

21.4.6 EP-pair from any bifinite domain into *uom*

context *bifinite-approx-chain begin*

definition

uom-emb :: 'a \rightarrow *uom*

where

uom-emb = *compact-basis.extension* ($\lambda x.$ *uom-principal* (*basis-emb x*))

definition

uom-prj :: *uom* \rightarrow 'a

where

uom-prj = *uom.extension* ($\lambda x.$ *Rep-compact-basis* (*basis-prj x*))

lemma *uom-emb-principal*:

uom-emb·(*Rep-compact-basis x*) = *uom-principal* (*basis-emb x*)

unfolding *uom-emb-def*

apply (*rule compact-basis.extension-principal*)

apply (*rule uom.principal-mono*)

apply (*erule basis-emb-mono*)

done

lemma *uom-prj-principal*:

uom-prj·(*uom-principal x*) = *Rep-compact-basis* (*basis-prj x*)

unfolding *uom-prj-def*

apply (*rule uom.extension-principal*)

apply (*rule compact-basis.principal-mono*)

apply (*erule basis-prj-mono*)

done

lemma *ep-pair-uom*: *ep-pair uom-emb uom-prj*

apply *standard*

apply (*rule compact-basis.principal-induct, simp*)

apply (*simp add: uom-emb-principal uom-prj-principal*)

apply (*simp add: basis-prj-basis-emb*)

apply (*rule uom.principal-induct, simp*)

apply (*simp add: uom-emb-principal uom-prj-principal*)

```

apply (rule basis-emb-prj-less)
done

```

```

end

```

```

abbreviation udom-emb  $\equiv$  bifinite-approx-chain.udom-emb

```

```

abbreviation udom-prj  $\equiv$  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  $\Rightarrow$  udom  $\rightarrow$  udom

```

where

```

  udom-approx i =

```

```

    udom.extension ( $\lambda x.$  udom-principal (ubasis-until ( $\lambda y.$   $y \leq i$ )  $x$ ))

```

lemma udom-approx-mono:

```

  ubasis-le a b  $\implies$ 

```

```

    udom-principal (ubasis-until ( $\lambda y.$   $y \leq i$ ) a)  $\sqsubseteq$ 

```

```

    udom-principal (ubasis-until ( $\lambda y.$   $y \leq i$ ) 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: $\llbracket \text{cont } f; \text{ finite } S \rrbracket \implies \text{adm } (\lambda x. f x \in S)$

by (erule adm-subst, induct set: finite, simp-all)

lemma udom-approx-principal:

```

  udom-approx i.(udom-principal x) =

```

```

    udom-principal (ubasis-until ( $\lambda y.$   $y \leq i$ )  $x$ )

```

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  $\sqsubseteq$  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]: (⊔ 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-distrib)
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 (⊔ i. udom-approx i) = ID
    by (rule lub-udom-approx)

```

```

qed

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

hide-const (open) node

notation binomial (infixl choose 65)

end

```

22 Algebraic deflations

```

theory Algebraic
imports Universal Map-Functions
begin

```

```

default-sort bifinite

```

22.1 Type constructor for finite deflations

```

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

```

```

instantiation fin-defl :: (bifinite) below
begin

```

```

definition below-fin-defl-def:
  below ≡ λx y. Rep-fin-defl x ⊆ Rep-fin-defl y

```

```

instance ..
end

```

```

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

```

```

lemma finite-deflation-Rep-fin-defl: finite-deflation (Rep-fin-defl d)
using Rep-fin-defl by simp

```

```

lemma deflation-Rep-fin-defl: deflation (Rep-fin-defl d)
using finite-deflation-Rep-fin-defl
by (rule finite-deflation-imp-deflation)

```

```

interpretation Rep-fin-defl: finite-deflation Rep-fin-defl d
by (rule finite-deflation-Rep-fin-defl)

```

```

lemma fin-defl-belowI:
  (λx. Rep-fin-defl a · x = x ⇒ Rep-fin-defl b · x = x) ⇒ a ⊆ b

```

unfolding *below-fin-defl-def*
by (*rule Rep-fin-defl.belowI*)

lemma *fin-defl-belowD*:

$\llbracket a \sqsubseteq b; \text{Rep-fin-defl } a \cdot x = x \rrbracket \implies \text{Rep-fin-defl } b \cdot x = x$

unfolding *below-fin-defl-def*
by (*rule Rep-fin-defl.belowD*)

lemma *fin-defl-eqI*:

$(\bigwedge x. \text{Rep-fin-defl } a \cdot x = x \longleftrightarrow \text{Rep-fin-defl } b \cdot x = x) \implies a = b$

apply (*rule below-antisym*)
apply (*rule fin-defl-belowI, simp*)
apply (*rule fin-defl-belowI, simp*)
done

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

lemma *Abs-fin-defl-mono*:

$\llbracket \text{finite-deflation } a; \text{finite-deflation } b; a \sqsubseteq b \rrbracket$
 $\implies \text{Abs-fin-defl } a \sqsubseteq \text{Abs-fin-defl } b$

unfolding *below-fin-defl-def*
by (*simp add: Abs-fin-defl-inverse*)

lemma (*in finite-deflation*) *compact-belowI*:

assumes $\bigwedge x. \text{compact } x \implies d \cdot x = x \implies f \cdot x = x$ **shows** $d \sqsubseteq f$
by (*rule belowI, rule assms, erule subst, rule compact*)

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

22.2 Defining algebraic deflations by ideal completion

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

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

definition

$x \sqsubseteq y \longleftrightarrow \text{Rep-defl } x \subseteq \text{Rep-defl } y$

instance ..
end

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

```

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

```

definition

```

defl-principal :: 'a fin-defl  $\Rightarrow$  'a defl where
defl-principal t = Abs-defl {u. u  $\sqsubseteq$  t}

```

```

lemma fin-defl-countable:  $\exists f :: 'a$  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 ( $\lambda d$ . 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  $\perp$ )  $\sqsubseteq$  x
apply (induct x rule: defl.principal-induct, simp)
apply (rule defl.principal-mono)
apply (simp add: below-fin-defl-def)
apply (simp add: Abs-fin-defl-inverse finite-deflation-bottom)

```


done

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

lemma *inst-defl-pcpo*: $\perp = \text{defl-principal } (\text{Abs-fin-defl } \perp)$
by (*rule defl-minimal [THEN bottomI, symmetric]*)

22.3 Applying algebraic deflations

definition

cast :: 'a *defl* \rightarrow 'a \rightarrow 'a

where

cast = *defl.extension Rep-fin-defl*

lemma *cast-defl-principal*:

cast.(*defl-principal* a) = *Rep-fin-defl* a

unfolding *cast-def*

apply (*rule defl.extension-principal*)

apply (*simp only: below-fin-defl-def*)

done

lemma *deflation-cast*: *deflation* (*cast*·*d*)

apply (*induct d rule: defl.principal-induct*)

apply (*rule adm-subst [OF - adm-deflation], simp*)

apply (*simp add: cast-defl-principal*)

apply (*rule finite-deflation-imp-deflation*)

apply (*rule finite-deflation-Rep-fin-defl*)

done

lemma *finite-deflation-cast*:

compact d \implies *finite-deflation* (*cast*·*d*)

apply (*drule defl.compact-imp-principal, clarify*)

apply (*simp add: cast-defl-principal*)

apply (*rule finite-deflation-Rep-fin-defl*)

done

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

by (*rule deflation-cast*)

declare *cast.idem* [*simp*]

lemma *compact-cast* [*simp*]: *compact d* \implies *compact* (*cast*·*d*)

apply (*rule finite-deflation-imp-compact*)

apply (*erule finite-deflation-cast*)

done

lemma *cast-below-cast*: *cast*·*A* \sqsubseteq *cast*·*B* \iff *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*: $\text{compact } (\text{cast} \cdot d) \longleftrightarrow \text{compact } d$
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*: $\text{cast} \cdot A \sqsubseteq \text{cast} \cdot B \implies A \sqsubseteq B$
by (*simp only*: *cast-below-cast*)

lemma *cast-eq-imp-eq*: $\text{cast} \cdot A = \text{cast} \cdot B \implies A = B$
by (*simp add*: *below-antisym cast-below-imp-below*)

lemma *cast-strict1* [*simp*]: $\text{cast} \cdot \perp = \perp$
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*]: $\text{cast} \cdot A \cdot \perp = \perp$
by (*rule cast.below [THEN bottomI]*)

22.4 Deflation combinators

definition

$$\begin{aligned} \text{defl-fun1 } e \text{ } p \text{ } f = & \\ & \text{defl.extension } (\lambda a. \\ & \text{defl-principal } (\text{Abs-fin-defl} \\ & (e \text{ oo } f \cdot (\text{Rep-fin-defl } a) \text{ oo } p))) \end{aligned}$$

definition

$$\begin{aligned} \text{defl-fun2 } e \text{ } p \text{ } f = & \\ & \text{defl.extension } (\lambda a. \\ & \text{defl.extension } (\lambda b. \\ & \text{defl-principal } (\text{Abs-fin-defl} \\ & (e \text{ oo } f \cdot (\text{Rep-fin-defl } a) \cdot (\text{Rep-fin-defl } b) \text{ oo } p)))) \end{aligned}$$

lemma *cast-defl-fun1*:

assumes *ep*: *ep-pair* e p
assumes f : $\bigwedge a. \text{finite-deflation } a \implies \text{finite-deflation } (f \cdot a)$
shows $\text{cast} \cdot (\text{defl-fun1 } e \text{ } p \text{ } f \cdot A) = e \text{ oo } f \cdot (\text{cast} \cdot A) \text{ oo } p$

proof –

have 1: $\bigwedge a. \text{finite-deflation } (e \text{ oo } f \cdot (\text{Rep-fin-defl } a) \text{ oo } p)$
apply (*rule ep-pair.finite-deflation-e-d-p [OF ep]*)

```

apply (rule f, rule finite-deflation-Rep-fin-defl)
done
show ?thesis
by (induct A rule: defl.principal-induct, simp)
    (simp only: defl-fun1-def
      defl.extension-principal
      defl.extension-mono
      defl.principal-mono
      Abs-fin-defl-mono [OF 1 1]
      monofun-cfun below-refl
      Rep-fin-defl-mono
      cast-defl-principal
      Abs-fin-defl-inverse [unfolded mem-Collect-eq, OF 1])
qed

```

```

lemma cast-defl-fun2:
  assumes ep: ep-pair e p
  assumes f:  $\bigwedge a b.$  finite-deflation a  $\implies$  finite-deflation b  $\implies$ 
    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:  $\bigwedge a b.$  finite-deflation
    (e oo f.(Rep-fin-defl a).(Rep-fin-defl b) oo p)
  apply (rule ep-pair.finite-deflation-e-d-p [OF ep])
  apply (rule f, (rule finite-deflation-Rep-fin-defl)+)
  done
show ?thesis
  apply (induct A rule: defl.principal-induct, simp)
  apply (induct B rule: defl.principal-induct, simp)
  by (simp only: defl-fun2-def
    defl.extension-principal
    defl.extension-mono
    defl.principal-mono
    Abs-fin-defl-mono [OF 1 1]
    monofun-cfun below-refl
    Rep-fin-defl-mono
    cast-defl-principal
    Abs-fin-defl-inverse [unfolded mem-Collect-eq, OF 1])
qed
end

```

23 Representable domains

```

theory Representable
imports Algebraic Map-Functions HOL-Library.Countable
begin

default-sort cpo

```

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'(-))))
translations LIFTDEFL('t) ⇒ CONST liftdefl TYPE('t)
```

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

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

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

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

```
instance domain ⊆ predomain
```

proof

```
show ep-pair liftemb (liftprj::udom⊥ → 'a⊥)
  unfolding liftemb-eq liftprj-eq
  by (intro ep-pair-u-map ep-pair-emb-prj)
show cast·LIFTDEFL('a) = liftemb oo (liftprj::udom⊥ → 'a⊥)
  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  $\exists (a::nat \Rightarrow 'a::pcpo \rightarrow 'a).$  approx-chain a
proof –
  interpret ep-pair e p by fact
  obtain Y where Y:  $\forall i. Y\ i \sqsubseteq Y\ (Suc\ i)$ 
  and t: t = ( $\bigsqcup i. defl-principal\ (Y\ i)$ )
  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 ( $\lambda i. approx\ i$ )
  unfolding approx-def by (simp add: Y)
  show ( $\bigsqcup i. approx\ i$ ) = ID
  unfolding approx-def
  by (simp add: lub-distrib Y t [symmetric] cast-t cfun-eq-iff)
  show  $\bigwedge i. finite-deflation\ (approx\ i)$ 
  unfolding approx-def
  apply (rule finite-deflation-p-d-e)
  apply (rule finite-deflation-cast)
  apply (rule defl.compact-principal)
  apply (rule below-trans [OF monofun-cfun-fun])
  apply (rule is-ub-the-lub, simp add: Y)
  apply (simp add: lub-distrib Y t [symmetric] cast-t)
  done
qed
thus  $\exists (a::nat \Rightarrow 'a \rightarrow 'a).$  approx-chain a by – (rule exI)

```

qed

instance $\text{domain} \subseteq \text{bifinite}$

by *standard* (rule *approx-chain-ep-cast* [OF *ep-pair-emb-prj cast-DEFL*])

instance $\text{predomain} \subseteq \text{profinite}$

by *standard* (rule *approx-chain-ep-cast* [OF *predomain-ep cast-liftdefl*])

23.3 Universal domain ep-pairs

definition $u\text{-emb} = \text{udom-emb } (\lambda i. u\text{-map} \cdot (\text{udom-approx } i))$

definition $u\text{-prj} = \text{udom-prj } (\lambda i. u\text{-map} \cdot (\text{udom-approx } i))$

definition $\text{prod-emb} = \text{udom-emb } (\lambda i. \text{prod-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i))$

definition $\text{prod-prj} = \text{udom-prj } (\lambda i. \text{prod-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i))$

definition $\text{sprod-emb} = \text{udom-emb } (\lambda i. \text{sprod-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i))$

definition $\text{sprod-prj} = \text{udom-prj } (\lambda i. \text{sprod-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i))$

definition $\text{ssum-emb} = \text{udom-emb } (\lambda i. \text{ssum-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i))$

definition $\text{ssum-prj} = \text{udom-prj } (\lambda i. \text{ssum-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i))$

definition $\text{sfun-emb} = \text{udom-emb } (\lambda i. \text{sfun-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i))$

definition $\text{sfun-prj} = \text{udom-prj } (\lambda i. \text{sfun-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i))$

lemma *ep-pair-u*: $\text{ep-pair } u\text{-emb } u\text{-prj}$

unfolding *u-emb-def* *u-prj-def*

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

lemma *ep-pair-prod*: $\text{ep-pair } \text{prod-emb } \text{prod-prj}$

unfolding *prod-emb-def* *prod-prj-def*

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

lemma *ep-pair-sprod*: $\text{ep-pair } \text{sprod-emb } \text{sprod-prj}$

unfolding *sprod-emb-def* *sprod-prj-def*

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

lemma *ep-pair-ssum*: $\text{ep-pair } \text{ssum-emb } \text{ssum-prj}$

unfolding *ssum-emb-def* *ssum-prj-def*

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

lemma *ep-pair-sfun*: $\text{ep-pair } \text{sfun-emb } \text{sfun-prj}$

unfolding *sfun-emb-def* *sfun-prj-def*

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

23.4 Type combinators

definition $u\text{-defl} :: \text{udom defl} \rightarrow \text{udom defl}$
where $u\text{-defl} = \text{defl-fun1 } u\text{-emb } u\text{-prj } u\text{-map}$

definition $\text{prod-defl} :: \text{udom defl} \rightarrow \text{udom defl} \rightarrow \text{udom defl}$
where $\text{prod-defl} = \text{defl-fun2 } \text{prod-emb } \text{prod-prj } \text{prod-map}$

definition $\text{sprod-defl} :: \text{udom defl} \rightarrow \text{udom defl} \rightarrow \text{udom defl}$
where $\text{sprod-defl} = \text{defl-fun2 } \text{sprod-emb } \text{sprod-prj } \text{sprod-map}$

definition $\text{ssum-defl} :: \text{udom defl} \rightarrow \text{udom defl} \rightarrow \text{udom defl}$
where $\text{ssum-defl} = \text{defl-fun2 } \text{ssum-emb } \text{ssum-prj } \text{ssum-map}$

definition $\text{sfun-defl} :: \text{udom defl} \rightarrow \text{udom defl} \rightarrow \text{udom defl}$
where $\text{sfun-defl} = \text{defl-fun2 } \text{sfun-emb } \text{sfun-prj } \text{sfun-map}$

lemma cast-u-defl :

$$\text{cast} \cdot (u\text{-defl} \cdot A) = u\text{-emb } \text{oo } u\text{-map} \cdot (\text{cast} \cdot A) \text{oo } u\text{-prj}$$

using $\text{ep-pair-u } \text{finite-deflation-u-map}$

unfolding $u\text{-defl-def}$ **by** (rule cast-defl-fun1)

lemma cast-prod-defl :

$$\text{cast} \cdot (\text{prod-defl} \cdot A \cdot B) =$$

$$\text{prod-emb } \text{oo } \text{prod-map} \cdot (\text{cast} \cdot A) \cdot (\text{cast} \cdot B) \text{oo } \text{prod-prj}$$

using $\text{ep-pair-prod } \text{finite-deflation-prod-map}$

unfolding prod-defl-def **by** (rule cast-defl-fun2)

lemma cast-sprod-defl :

$$\text{cast} \cdot (\text{sprod-defl} \cdot A \cdot B) =$$

$$\text{sprod-emb } \text{oo } \text{sprod-map} \cdot (\text{cast} \cdot A) \cdot (\text{cast} \cdot B) \text{oo } \text{sprod-prj}$$

using $\text{ep-pair-sprod } \text{finite-deflation-sprod-map}$

unfolding sprod-defl-def **by** (rule cast-defl-fun2)

lemma cast-ssum-defl :

$$\text{cast} \cdot (\text{ssum-defl} \cdot A \cdot B) =$$

$$\text{ssum-emb } \text{oo } \text{ssum-map} \cdot (\text{cast} \cdot A) \cdot (\text{cast} \cdot B) \text{oo } \text{ssum-prj}$$

using $\text{ep-pair-ssum } \text{finite-deflation-ssum-map}$

unfolding ssum-defl-def **by** (rule cast-defl-fun2)

lemma cast-sfun-defl :

$$\text{cast} \cdot (\text{sfun-defl} \cdot A \cdot B) =$$

$$\text{sfun-emb } \text{oo } \text{sfun-map} \cdot (\text{cast} \cdot A) \cdot (\text{cast} \cdot B) \text{oo } \text{sfun-prj}$$

using $\text{ep-pair-sfun } \text{finite-deflation-sfun-map}$

unfolding sfun-defl-def **by** (rule cast-defl-fun2)

Special deflation combinator for unpointed types.

definition $u\text{-liftdefl} :: \text{udom } u \text{ defl} \rightarrow \text{udom defl}$
where $u\text{-liftdefl} = \text{defl-fun1 } u\text{-emb } u\text{-prj } \text{ID}$

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 *u-dom* :: *domain*

begin

definition [*simp*]:

emb = (*ID* :: *u-dom* → *u-dom*)

definition [*simp*]:

prj = (*ID* :: *u-dom* → *u-dom*)

definition

defl (*t*::*u-dom* *itself*) = (\bigsqcup *i*. *defl-principal* (*Abs-fin-defl* (*u-dom-approx* *i*)))

definition

(*liftemb* :: *u-dom* *u* → *u-dom* *u*) = *u-map*·*emb*

definition

(*liftprj* :: *u-dom* *u* → *u-dom* *u*) = *u-map*·*prj*

definition

liftdefl (*t*::*u-dom* *itself*) = *liftdefl-of*·*DEFL*(*u-dom*)

instance proof

show *ep-pair* *emb* (*prj* :: *u-dom* → *u-dom*)

by (*simp* add: *ep-pair.intro*)

show *cast*·*DEFL*(*u-dom*) = *emb* oo (*prj* :: *u-dom* → *u-dom*)

unfolding *defl-u-dom-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-u-dom-approx*)

apply (*rule* *chainE*)

apply (*rule* *chain-u-dom-approx*)

apply (*subst* *cast-defl-principal*)

apply (*simp* add: *Abs-fin-defl-inverse finite-deflation-u-dom-approx*)

done

qed (*fact* *liftemb-u-dom-def liftprj-u-dom-def liftdefl-u-dom-def*)+

end

23.5.2 Lifted cpo

instantiation $u :: (\text{predomain}) \text{ domain}$
begin

definition

$emb = u\text{-emb} \text{ oo } \text{liftemb}$

definition

$prj = \text{liftprj} \text{ oo } u\text{-prj}$

definition

$\text{defl} (t :: 'a \ u \ \text{itself}) = u\text{-liftdefl} \cdot \text{LIFTDEFL}('a)$

definition

$(\text{liftemb} :: 'a \ u \ u \rightarrow \text{udom} \ u) = u\text{-map} \cdot \text{emb}$

definition

$(\text{liftprj} :: \text{udom} \ u \rightarrow 'a \ u \ u) = u\text{-map} \cdot \text{prj}$

definition

$\text{liftdefl} (t :: 'a \ u \ \text{itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \ u)$

instance proof

show $ep\text{-pair} \ emb \ (prj :: \text{udom} \rightarrow 'a \ u)$

unfolding $emb\text{-u-def} \ prj\text{-u-def}$

by ($\text{intro} \ ep\text{-pair-comp} \ ep\text{-pair-u} \ \text{predomain-ep}$)

show $\text{cast-DEFL}('a \ u) = emb \ \text{oo} \ (prj :: \text{udom} \rightarrow 'a \ u)$

unfolding $emb\text{-u-def} \ prj\text{-u-def} \ \text{defl-u-def}$

by ($\text{simp add: cast-u-liftdefl cast-liftdefl assoc-oo}$)

qed ($\text{fact liftemb-u-def liftprj-u-def liftdefl-u-def}$)+

end

lemma $\text{DEFL-u: DEFL}('a :: \text{predomain} \ u) = u\text{-liftdefl} \cdot \text{LIFTDEFL}('a)$

by (rule defl-u-def)

23.5.3 Strict function space

instantiation $\text{sfun} :: (\text{domain}, \text{domain}) \ \text{domain}$
begin

definition

$emb = \text{sfun-emb} \ \text{oo} \ \text{sfun-map} \cdot \text{prj} \cdot \text{emb}$

definition

$prj = \text{sfun-map} \cdot \text{emb} \cdot \text{prj} \ \text{oo} \ \text{sfun-prj}$

definition

$$\text{defl } (t :: ('a \rightarrow! 'b) \text{ itself}) = \text{sfun-defl} \cdot \text{DEFL}('a) \cdot \text{DEFL}('b)$$
definition

$$(\text{liftemb} :: ('a \rightarrow! 'b) u \rightarrow \text{udom } u) = u\text{-map} \cdot \text{emb}$$
definition

$$(\text{liftprj} :: \text{udom } u \rightarrow ('a \rightarrow! 'b) u) = u\text{-map} \cdot \text{prj}$$
definition

$$\text{liftdefl } (t :: ('a \rightarrow! 'b) \text{ itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \rightarrow! 'b)$$
instance proof

$$\text{show } \text{ep-pair } \text{emb } (\text{prj} :: \text{udom} \rightarrow 'a \rightarrow! 'b)$$

$$\text{unfolding } \text{emb-sfun-def } \text{prj-sfun-def}$$

$$\text{by } (\text{intro } \text{ep-pair-comp } \text{ep-pair-sfun } \text{ep-pair-sfun-map } \text{ep-pair-emb-prj})$$

$$\text{show } \text{cast} \cdot \text{DEFL}('a \rightarrow! 'b) = \text{emb } \text{oo } (\text{prj} :: \text{udom} \rightarrow 'a \rightarrow! 'b)$$

$$\text{unfolding } \text{emb-sfun-def } \text{prj-sfun-def } \text{defl-sfun-def } \text{cast-sfun-defl}$$

$$\text{by } (\text{simp } \text{add: cast-DEFL } \text{oo-def } \text{sfun-eq-iff } \text{sfun-map-map})$$

$$\text{qed } (\text{fact } \text{liftemb-sfun-def } \text{liftprj-sfun-def } \text{liftdefl-sfun-def})+$$

end

lemma *DEFL-sfun*:
$$\text{DEFL}('a :: \text{domain} \rightarrow! 'b :: \text{domain}) = \text{sfun-defl} \cdot \text{DEFL}('a) \cdot \text{DEFL}('b)$$

$$\text{by } (\text{rule } \text{defl-sfun-def})$$
23.5.4 Continuous function space

$$\text{instantiation } \text{cfun} :: (\text{pre-domain}, \text{domain}) \text{ domain}$$

begin

definition

$$\text{emb} = \text{emb } \text{oo } \text{encode-cfun}$$
definition

$$\text{prj} = \text{decode-cfun } \text{oo } \text{prj}$$
definition

$$\text{defl } (t :: ('a \rightarrow 'b) \text{ itself}) = \text{DEFL}('a \rightarrow 'b)$$
definition

$$(\text{liftemb} :: ('a \rightarrow 'b) u \rightarrow \text{udom } u) = u\text{-map} \cdot \text{emb}$$
definition

$$(\text{liftprj} :: \text{udom } u \rightarrow ('a \rightarrow 'b) u) = u\text{-map} \cdot \text{prj}$$
definition

$\text{liftdefl } (t::('a \rightarrow 'b) \text{ itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \rightarrow 'b)$

instance proof

have $\text{ep-pair encode-cfun decode-cfun}$
by $(\text{rule ep-pair.intro, simp-all})$
thus $\text{ep-pair emb } (prj :: \text{udom} \rightarrow 'a \rightarrow 'b)$
unfolding $\text{emb-cfun-def prj-cfun-def}$
using $\text{ep-pair-emb-prj by } (\text{rule ep-pair-comp})$
show $\text{cast} \cdot \text{DEFL}('a \rightarrow 'b) = \text{emb oo } (prj :: \text{udom} \rightarrow 'a \rightarrow 'b)$
unfolding $\text{emb-cfun-def prj-cfun-def defl-cfun-def}$
by $(\text{simp add: cast-DEFL cfcomp1})$
qed $(\text{fact liftemb-cfun-def liftprj-cfun-def liftdefl-cfun-def})+$
end

lemma DEFL-cfun:

$\text{DEFL}('a::\text{predomain} \rightarrow 'b::\text{domain}) = \text{DEFL}('a \text{ u} \rightarrow! 'b)$
by $(\text{rule defl-cfun-def})$

23.5.5 Strict product

instantiation $\text{sprod} :: (\text{domain}, \text{domain}) \text{ domain}$
begin

definition

$\text{emb} = \text{sprod-emb oo sprod-map} \cdot \text{emb} \cdot \text{emb}$

definition

$\text{prj} = \text{sprod-map} \cdot \text{prj} \cdot \text{prj oo sprod-prj}$

definition

$\text{defl } (t::('a \otimes 'b) \text{ itself}) = \text{sprod-defl} \cdot \text{DEFL}('a) \cdot \text{DEFL}('b)$

definition

$(\text{liftemb} :: ('a \otimes 'b) \text{ u} \rightarrow \text{udom } \text{u}) = \text{u-map} \cdot \text{emb}$

definition

$(\text{liftprj} :: \text{udom } \text{u} \rightarrow ('a \otimes 'b) \text{ u}) = \text{u-map} \cdot \text{prj}$

definition

$\text{liftdefl } (t::('a \otimes 'b) \text{ itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \otimes 'b)$

instance proof

show $\text{ep-pair emb } (prj :: \text{udom} \rightarrow 'a \otimes 'b)$
unfolding $\text{emb-sprod-def prj-sprod-def}$
by $(\text{intro ep-pair-comp ep-pair-sprod ep-pair-sprod-map ep-pair-emb-prj})$
show $\text{cast} \cdot \text{DEFL}('a \otimes 'b) = \text{emb oo } (prj :: \text{udom} \rightarrow 'a \otimes 'b)$
unfolding $\text{emb-sprod-def prj-sprod-def defl-sprod-def cast-sprod-defl}$
by $(\text{simp add: cast-DEFL oo-def cfun-eq-iff sprod-map-map})$

qed (*fact liftemb-sprod-def liftprj-sprod-def liftdefl-sprod-def*)+

end

lemma *DEFL-sprod*:

$DEFL('a::domain \otimes 'b::domain) = sprod-defl \cdot DEFL('a) \cdot DEFL('b)$

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*:

$cast \cdot (prod-liftdefl \cdot a \cdot b) =$
(*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

$liftemb = (u-map-prod-emb \text{ oo } decode-prod-u) \text{ oo}$
(*sprod-map* · *liftemb* · *liftemb* *oo* *encode-prod-u*)

definition

$liftprj = (decode-prod-u \text{ oo } sprod-map \cdot liftprj \cdot liftprj) \text{ oo}$
(*encode-prod-u oo u-map-prod-prj*)

definition

$liftdefl (t::('a \times 'b) \text{ itself}) = prod-liftdefl \cdot LIFTDEFL('a) \cdot LIFTDEFL('b)$

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

$emb = prod-emb \circ prod-map \cdot emb \cdot emb$

definition

$prj = prod-map \cdot prj \cdot prj \circ prod-prj$

definition

$defl (t :: ('a \times 'b) itself) = prod-defl \cdot DEFL('a) \cdot DEFL('b)$

instance proof

show 1: $ep-pair \ emb (prj :: udom \rightarrow 'a \times 'b)$

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: $cast \cdot DEFL('a \times 'b) = emb \circ (prj :: udom \rightarrow 'a \times 'b)$

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: $liftemb = u-map \cdot (emb :: 'a \times 'b \rightarrow udom)$

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: $liftprj = u-map \cdot (prj :: udom \rightarrow 'a \times 'b)$

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 ($rename-tac \ y, \ case-tac \ prod-prj \cdot y, \ simp$)

done

show 5: $LIFTDEFL('a \times 'b) = liftdefl-of \cdot DEFL('a \times 'b)$

by ($rule \ cast-eq-imp-eq$)

($simp \ add: \ cast-liftdefl \ cast-liftdefl-of \ cast-DEFL \ 2 \ 3 \ 4 \ u-map-oo$)

qed

end

lemma $DEFL-prod$:

$DEFL('a :: domain \times 'b :: domain) = prod-defl \cdot DEFL('a) \cdot DEFL('b)$

by ($rule \ defl-prod-def$)

lemma $LIFTDEFL-prod$:

$LIFTDEFL('a :: predomain \times 'b :: predomain) =$

$prod-liftdefl \cdot LIFTDEFL('a) \cdot LIFTDEFL('b)$

by ($rule \ liftdefl-prod-def$)

23.5.7 Unit type

instantiation *unit* :: *domain*
begin

definition

$emb = (\perp :: unit \rightarrow udom)$

definition

$prj = (\perp :: udom \rightarrow unit)$

definition

$defl (t::unit\ itself) = \perp$

definition

$(liftemb :: unit\ u \rightarrow udom\ u) = u\text{-map}\cdot emb$

definition

$(liftprj :: udom\ u \rightarrow unit\ u) = u\text{-map}\cdot prj$

definition

$liftdefl (t::unit\ itself) = liftdefl\text{-of}\cdot DEFL(unit)$

instance proof

show *ep-pair* *emb* (*prj* :: *udom* \rightarrow *unit*)

unfolding *emb-unit-def* *prj-unit-def*

by (*simp* *add*: *ep-pair.intro*)

show *cast*·*DEFL*(*unit*) = *emb* *oo* (*prj* :: *udom* \rightarrow *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

$(liftemb :: 'a\ discr\ u \rightarrow udom\ u) = strictify\cdot up\ oo\ udom\text{-emb}\ discr\text{-approx}$

definition

$(liftprj :: udom\ u \rightarrow 'a\ discr\ u) = udom\text{-prj}\ discr\text{-approx}\ oo\ fup\cdot ID$

definition

$liftdefl (t::'a\ discr\ itself) =$

$(\bigsqcup i. defl\text{-principal}\ (Abs\text{-fin}\text{-defl}\ (liftemb\ oo\ discr\text{-approx}\ i\ oo\ (liftprj::udom\ u \rightarrow 'a\ discr\ u))))$

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-distrib)
  done
qed

end

```

23.5.9 Strict sum

instantiation $ssum :: (domain, domain) domain$
begin

definition

$$emb = ssum-emb \circ ssum-map \cdot emb \cdot emb$$

definition

$$prj = ssum-map \cdot prj \cdot prj \circ ssum-prj$$

definition

$$defl (t :: ('a \oplus 'b) itself) = ssum-defl \cdot DEFL('a) \cdot DEFL('b)$$

definition

$$(liftemb :: ('a \oplus 'b) u \rightarrow udom u) = u-map \cdot emb$$

definition

$$(liftprj :: udom u \rightarrow ('a \oplus 'b) u) = u-map \cdot prj$$

definition

$$\text{liftdefl } (t :: ('a \oplus 'b) \text{ itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \oplus 'b)$$
instance proof

show $\text{ep-pair } \text{emb } (\text{prj} :: \text{udom} \rightarrow 'a \oplus 'b)$
unfolding $\text{emb-ssum-def } \text{prj-ssum-def}$
by $(\text{intro } \text{ep-pair-comp } \text{ep-pair-ssum } \text{ep-pair-ssum-map } \text{ep-pair-emb-prj})$
show $\text{cast-DEFL}('a \oplus 'b) = \text{emb } \text{oo } (\text{prj} :: \text{udom} \rightarrow 'a \oplus 'b)$
unfolding $\text{emb-ssum-def } \text{prj-ssum-def } \text{defl-ssum-def } \text{cast-ssum-defl}$
by $(\text{simp } \text{add: } \text{cast-DEFL } \text{oo-def } \text{cfun-eq-iff } \text{ssum-map-map})$
qed $(\text{fact } \text{liftemb-ssum-def } \text{liftprj-ssum-def } \text{liftdefl-ssum-def})+$

end**lemma** DEFL-ssum :
$$\text{DEFL}('a :: \text{domain} \oplus 'b :: \text{domain}) = \text{ssum-defl} \cdot \text{DEFL}('a) \cdot \text{DEFL}('b)$$

by $(\text{rule } \text{defl-ssum-def})$

23.5.10 Lifted HOL type

instantiation $\text{lift} :: (\text{countable}) \text{ domain}$
begin

definition

$$\text{emb} = \text{emb } \text{oo } (\lambda x. \text{Rep-lift } x)$$
definition

$$\text{prj} = (\lambda y. \text{Abs-lift } y) \text{oo } \text{prj}$$
definition

$$\text{defl } (t :: 'a \text{ lift } \text{itself}) = \text{DEFL}('a \text{ discr } u)$$
definition

$$(\text{liftemb} :: 'a \text{ lift } u \rightarrow \text{udom } u) = u\text{-map} \cdot \text{emb}$$
definition

$$(\text{liftprj} :: \text{udom } u \rightarrow 'a \text{ lift } u) = u\text{-map} \cdot \text{prj}$$
definition

$$\text{liftdefl } (t :: 'a \text{ lift } \text{itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \text{ lift})$$
instance proof

note $[\text{simp}] = \text{cont-Rep-lift } \text{cont-Abs-lift } \text{Rep-lift-inverse } \text{Abs-lift-inverse}$
have $\text{ep-pair } (\lambda(x :: 'a \text{ lift}). \text{Rep-lift } x) (\lambda y. \text{Abs-lift } y)$
by $(\text{simp } \text{add: } \text{ep-pair-def})$
thus $\text{ep-pair } \text{emb } (\text{prj} :: \text{udom} \rightarrow 'a \text{ lift})$
unfolding $\text{emb-lift-def } \text{prj-lift-def}$
using ep-pair-emb-prj **by** $(\text{rule } \text{ep-pair-comp})$
show $\text{cast-DEFL}('a \text{ lift}) = \text{emb } \text{oo } (\text{prj} :: \text{udom} \rightarrow 'a \text{ lift})$


```

    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  $\leftarrow$  (type) unit lift

```

```

definition ONE :: one
  where ONE  $\equiv$  Def ()

```

Exhaustion and Elimination for type *one*

```

lemma Exh-one:  $t = \perp \vee t = ONE$ 
  by (induct t) (simp-all add: ONE-def)

```

```

lemma oneE [case-names bottom ONE]:  $\llbracket p = \perp \implies Q; p = ONE \implies Q \rrbracket \implies Q$ 
  by (induct p) (simp-all add: ONE-def)

```

```

lemma one-induct [case-names bottom ONE]:  $P \perp \implies P ONE \implies P x$ 
  by (cases x rule: oneE) simp-all

```

```

lemma dist-below-one [simp]:  $ONE \not\sqsubseteq \perp$ 
  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]:  $ONE \neq \perp$ 
  by (simp add: ONE-def)

```

```

lemma one-neq-iffs [simp]:

```

```

   $x \neq ONE \longleftrightarrow x = \perp$ 

```

```

   $ONE \neq x \longleftrightarrow x = \perp$ 

```

```

   $x \neq \perp \longleftrightarrow x = ONE$ 

```

```

   $\perp \neq x \longleftrightarrow x = ONE$ 

```

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 = (Λ a x. seq.x.a)

translations

case x of XCONST ONE \Rightarrow t \Leftrightarrow CONST one-case.t.x
case x of XCONST ONE :: 'a \Rightarrow t \rightarrow CONST one-case.t.x
 Λ (XCONST ONE). t \Leftrightarrow CONST one-case.t

lemma one-case1 [simp]: (case \perp of ONE \Rightarrow t) = \perp
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

25 Fixed point operator and admissibility

theory Fix
imports Cfun
begin

default-sort pcpo

25.1 Iteration

primrec iterate :: nat \Rightarrow ('a::cpo \rightarrow 'a) \rightarrow ('a \rightarrow '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*: $iterate (Suc\ n)\cdot F\cdot x = iterate\ n\cdot F\cdot(F\cdot x)$
by (*induct n simp-all*)

lemma *iterate-iterate*: $iterate\ m\cdot F\cdot(iterate\ n\cdot F\cdot x) = iterate\ (m + n)\cdot F\cdot x$
by (*induct m simp-all*)

The sequence of function iterations is a chain.

lemma *chain-iterate* [*simp*]: $chain\ (\lambda i.\ iterate\ i\cdot F\cdot\perp)$
by (*rule chainI, unfold iterate-Suc2, rule monofun-cfun-arg, rule minimal*)

25.2 Least fixed point operator

definition $fix :: ('a \rightarrow 'a) \rightarrow 'a$
where $fix = (\Lambda\ F.\ \bigsqcup i.\ iterate\ i\cdot F\cdot\perp)$

Binder syntax for *fix*

abbreviation $fix\text{-}syn :: ('a \Rightarrow 'a) \Rightarrow 'a$ (**binder** μ 10)
where $fix\text{-}syn\ (\lambda x.\ f\ x) \equiv fix\cdot(\Lambda\ x.\ f\ x)$

notation (*ASCII*)
 $fix\text{-}syn$ (**binder** *FIX* 10)

Properties of *fix*

direct connection between *fix* and iteration

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

lemma *iterate-below-fix*: $iterate\ n\cdot f\cdot\perp \sqsubseteq fix\cdot f$
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*: $F\cdot x \sqsubseteq x \implies fix\cdot F \sqsubseteq x$
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*: $F \cdot x = x \implies \text{fix} \cdot F \sqsubseteq x$
by (*rule fix-least-below*) *simp*

lemma *fix-eqI*:
assumes *fixed*: $F \cdot x = x$
and *least*: $\bigwedge z. F \cdot z = z \implies x \sqsubseteq z$
shows $\text{fix} \cdot F = x$
apply (*rule below-antisym*)
apply (*rule fix-least* [*OF fixed*])
apply (*rule least* [*OF fix-eq* [*symmetric*]])
done

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

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

lemma *fix-eq4*: $f = \text{fix} \cdot F \implies f = F \cdot f$
by (*erule ssubst*) (*rule fix-eq*)

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

strictness of *fix*

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

lemma *fix-strict*: $F \cdot \perp = \perp \implies \text{fix} \cdot F = \perp$
by (*simp add: fix-bottom-iff*)

lemma *fix-defined*: $F \cdot \perp \neq \perp \implies \text{fix} \cdot F \neq \perp$
by (*simp add: fix-bottom-iff*)

fix applied to identity and constant functions

lemma *fix-id*: $(\mu x. x) = \perp$
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*: $\text{adm } P \implies P \perp \implies (\bigwedge x. P x \implies P (F \cdot x)) \implies P (\text{fix} \cdot F)$
unfolding *fix-def2*
apply (*erule admD*)
apply (*rule chain-iterate*)
apply (*rule nat-induct, simp-all*)
done

lemma *cont-fix-ind*: $\text{cont } F \implies \text{adm } P \implies P \perp \implies (\bigwedge x. P x \implies P (F x)) \implies P (\text{fix} \cdot (\text{Abs-cfun } F))$
 by (*simp add: fix-ind*)

lemma *def-fix-ind*: $\llbracket f \equiv \text{fix} \cdot F; \text{adm } P; P \perp; \bigwedge x. P x \implies P (F \cdot x) \rrbracket \implies P f$
 by (*simp add: fix-ind*)

lemma *fix-ind2*:
assumes *adm*: $\text{adm } P$
assumes *0*: $P \perp$ **and** *1*: $P (F \cdot \perp)$
assumes *step*: $\bigwedge x. \llbracket P x; P (F \cdot x) \rrbracket \implies P (F \cdot (F \cdot x))$
shows $P (\text{fix} \cdot F)$
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*: $\text{adm } (\lambda x. P (\text{fst } x) (\text{snd } x))$
assumes *base*: $P \perp \perp$
assumes *step*: $\bigwedge x y. P x y \implies P (F \cdot x) (G \cdot y)$
shows $P (\text{fix} \cdot F) (\text{fix} \cdot G)$

proof –

from *adm* **have** *adm'*: $\text{adm } (\text{case-prod } P)$
unfolding *split-def* .
have $P (\text{iterate } i \cdot F \cdot \perp) (\text{iterate } i \cdot G \cdot \perp)$ **for** *i*
 by (*induct i*) (*simp add: base, simp add: step*)
then **have** $\bigwedge i. \text{case-prod } P (\text{iterate } i \cdot F \cdot \perp, \text{iterate } i \cdot G \cdot \perp)$
 by *simp*
then **have** $\text{case-prod } P (\bigsqcup i. (\text{iterate } i \cdot F \cdot \perp, \text{iterate } i \cdot G \cdot \perp))$
 by – (*rule admD [OF adm']*, *simp*, *assumption*)
then **have** $\text{case-prod } P (\bigsqcup i. \text{iterate } i \cdot F \cdot \perp, \bigsqcup i. \text{iterate } i \cdot G \cdot \perp)$

by (*simp add: lub-Pair*)
 then have $P (\bigsqcup i. \text{iterate } i \cdot F \cdot \perp) (\bigsqcup i. \text{iterate } i \cdot G \cdot \perp)$
 by *simp*
 then show $P (\text{fix} \cdot F) (\text{fix} \cdot G)$
 by (*simp add: fix-def2*)
 qed

lemma *cont-parallel-fix-ind*:
 assumes *cont F and cont G*
 assumes *adm* $(\lambda x. P (\text{fst } x) (\text{snd } x))$
 assumes $P \perp \perp$
 assumes $\bigwedge x y. P x y \implies P (F x) (G y)$
 shows $P (\text{fix} \cdot (\text{Abs-cfun } F)) (\text{fix} \cdot (\text{Abs-cfun } G))$
 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*:
 $\text{fix} \cdot (F :: 'a \times 'b \rightarrow 'a \times 'b) =$
 $(\mu x. \text{fst} (F \cdot (x, \mu y. \text{snd} (F \cdot (x, y))))),$
 $\mu y. \text{snd} (F \cdot (\mu x. \text{fst} (F \cdot (x, \mu y. \text{snd} (F \cdot (x, y))))), y))$
 (is $\text{fix} \cdot F = (?x, ?y)$)

proof (*rule fix-eq1*)
 have $*$: $\text{fst} (F \cdot (?x, ?y)) = ?x$
 by (*rule trans [symmetric, OF fix-eq], simp*)
 have $\text{snd} (F \cdot (?x, ?y)) = ?y$
 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 $F \cdot z: F \cdot z = z$
obtain $x y$ **where** $z: z = (x, y)$ **by** (*rule prod.exhaust*)
from $F \cdot z \ z$ **have** $F \cdot x: \text{fst} (F \cdot (x, y)) = x$ **by** *simp*
from $F \cdot z \ z$ **have** $F \cdot y: \text{snd} (F \cdot (x, y)) = y$ **by** *simp*
let $?y1 = \mu y. \text{snd} (F \cdot (x, y))$
have $?y1 \sqsubseteq y$
 by (*rule fix-least*) (*simp add: F-y*)
then have $\text{fst} (F \cdot (x, ?y1)) \sqsubseteq \text{fst} (F \cdot (x, y))$
 by (*simp add: fst-monofun monofun-cfun*)
with $F \cdot x$ **have** $\text{fst} (F \cdot (x, ?y1)) \sqsubseteq x$
 by *simp*
then have $*$: $?x \sqsubseteq x$
 by (*simp add: fix-least-below*)
then have $\text{snd} (F \cdot (?x, y)) \sqsubseteq \text{snd} (F \cdot (x, y))$
 by (*simp add: snd-monofun monofun-cfun*)
with $F \cdot y$ **have** $\text{snd} (F \cdot (?x, y)) \sqsubseteq y$

```

  by simp
  then have ?y  $\sqsubseteq$  y
    by (simp add: fix-least-below)
  with z * show (?x, ?y)  $\sqsubseteq$  z
    by simp
qed
end

```

26 Package for defining recursive functions in HOLCF

```

theory Fixrec
imports Cprod Sprod Ssum Up One Tr Fix
keywords fixrec :: thy-defn
begin

```

26.1 Pattern-match monad

```
default-sort cpo
```

```

pcpodef 'a match = UNIV::(one ++ 'a u) set
by simp-all

```

definition

```

fail :: 'a match where
fail = Abs-match (sinl·ONE)

```

definition

```

succeed :: 'a  $\rightarrow$  'a match where
succeed = ( $\Lambda$  x. Abs-match (sinr·(up·x)))

```

```
lemma matchE [case-names bottom fail succeed, cases type: match]:
```

```
[[p =  $\perp$   $\implies$  Q; p = fail  $\implies$  Q;  $\bigwedge$ x. p = succeed·x  $\implies$  Q]]  $\implies$  Q
```

```
unfolding fail-def succeed-def
```

```
apply (cases p, rename-tac r)
```

```
apply (rule-tac p=r in ssumE, simp add: Abs-match-strict)
```

```
apply (rule-tac p=x in oneE, simp, simp)
```

```
apply (rule-tac p=y in upE, simp, simp add: cont-Abs-match)
```

```
done
```

```
lemma succeed-defined [simp]: succeed·x  $\neq$   $\perp$ 
```

```
by (simp add: succeed-def cont-Abs-match Abs-match-bottom-iff)
```

```
lemma fail-defined [simp]: fail  $\neq$   $\perp$ 
```

```
by (simp add: fail-def Abs-match-bottom-iff)
```

```
lemma succeed-eq [simp]: (succeed·x = succeed·y) = (x = y)
```

```
by (simp add: succeed-def cont-Abs-match Abs-match-inject)
```

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

26.1.1 Run operator

definition

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

rewrite rules for run

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

lemma *run-fail* [*simp*]: $run \cdot fail = \perp$
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

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

abbreviation

$mplus\text{-}syn :: ['a \text{ match}, 'a \text{ match}] \Rightarrow 'a \text{ match} \text{ (infixr } +++ \ 65) \text{ where}$
 $m1 \ +++ \ m2 == mplus \cdot m1 \cdot m2$

rewrite rules for mplus

lemma *mplus-strict* [*simp*]: $\perp \ +++ \ m = \perp$
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*]: $succeed \cdot x \ +++ \ m = succeed \cdot x$
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

default-sort *pcpo*

definition

match-bottom :: $'a \rightarrow 'c \text{ match} \rightarrow 'c \text{ match}$

where

match-bottom = $(\Lambda x k. \text{seq}\cdot x\cdot \text{fail})$

definition

match-Pair :: $'a::\text{cpo} \times 'b::\text{cpo} \rightarrow ('a \rightarrow 'b \rightarrow 'c \text{ match}) \rightarrow 'c \text{ match}$

where

match-Pair = $(\Lambda x k. \text{csplit}\cdot k\cdot x)$

definition

match-spair :: $'a \otimes 'b \rightarrow ('a \rightarrow 'b \rightarrow 'c \text{ match}) \rightarrow 'c \text{ match}$

where

match-spair = $(\Lambda x k. \text{ssplit}\cdot k\cdot x)$

definition

match-sinl :: $'a \oplus 'b \rightarrow ('a \rightarrow 'c \text{ match}) \rightarrow 'c \text{ match}$

where

match-sinl = $(\Lambda x k. \text{ssc}\text{ase}\cdot k\cdot (\Lambda b. \text{fail})\cdot x)$

definition

match-sinr :: $'a \oplus 'b \rightarrow ('b \rightarrow 'c \text{ match}) \rightarrow 'c \text{ match}$

where

match-sinr = $(\Lambda x k. \text{ssc}\text{ase}\cdot (\Lambda a. \text{fail})\cdot k\cdot x)$

definition

match-up :: $'a::\text{cpo} \ u \rightarrow ('a \rightarrow 'c \text{ match}) \rightarrow 'c \text{ match}$

where

match-up = $(\Lambda x k. \text{fup}\cdot k\cdot x)$

definition

match-ONE :: $\text{one} \rightarrow 'c \text{ match} \rightarrow 'c \text{ match}$

where

match-ONE = $(\Lambda \text{ONE} k. k)$

definition

match-TT :: $\text{tr} \rightarrow 'c \text{ match} \rightarrow 'c \text{ match}$

where

match-TT = $(\Lambda x k. \text{If } x \text{ then } k \text{ else fail})$

definition

match-FF :: $\text{tr} \rightarrow 'c \text{ match} \rightarrow 'c \text{ match}$

where

$match\text{-}FF = (\Lambda x k. \text{If } x \text{ then fail else } k)$

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

$match\text{-}bottom \cdot x \cdot k = (\text{if } x = \perp \text{ then } \perp \text{ else fail})$

by (simp add: match-bottom-def)

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

$match\text{-}Pair \cdot (x, y) \cdot k = k \cdot x \cdot y$

by (simp-all add: match-Pair-def)

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

$\llbracket x \neq \perp; y \neq \perp \rrbracket \implies match\text{-}spair \cdot (:x, y) \cdot k = k \cdot x \cdot y$

$match\text{-}spair \cdot \perp \cdot k = \perp$

by (simp-all add: match-spair-def)

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

$x \neq \perp \implies match\text{-}sinl \cdot (sinl \cdot x) \cdot k = k \cdot x$

$y \neq \perp \implies match\text{-}sinl \cdot (sinr \cdot y) \cdot k = fail$

$match\text{-}sinl \cdot \perp \cdot k = \perp$

by (simp-all add: match-sinl-def)

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

$x \neq \perp \implies match\text{-}sinr \cdot (sinl \cdot x) \cdot k = fail$

$y \neq \perp \implies match\text{-}sinr \cdot (sinr \cdot y) \cdot k = k \cdot y$

$match\text{-}sinr \cdot \perp \cdot k = \perp$

by (simp-all add: match-sinr-def)

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

$match\text{-}up \cdot (up \cdot x) \cdot k = k \cdot x$

$match\text{-}up \cdot \perp \cdot k = \perp$

by (simp-all add: match-up-def)

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

$match\text{-}ONE \cdot ONE \cdot k = k$

$match\text{-}ONE \cdot \perp \cdot k = \perp$

by (simp-all add: match-ONE-def)

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

$match\text{-}TT \cdot TT \cdot k = k$

$match\text{-}TT \cdot FF \cdot k = fail$

$match\text{-}TT \cdot \perp \cdot k = \perp$

by (simp-all add: match-TT-def)

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

$match\text{-}FF \cdot FF \cdot k = k$

$match\text{-}FF \cdot TT \cdot k = fail$

$match\text{-}FF \cdot \perp \cdot k = \perp$

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-equal1*: $\llbracket x \equiv \text{fst } p; y \equiv \text{snd } p \rrbracket \Longrightarrow (x, y) \equiv p$
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*:
 $\llbracket f \equiv \text{fix} \cdot (\text{Abs-cfun } F); \text{cont } F \rrbracket \Longrightarrow f = F f$
by (*simp, subst fix-eq, simp*)

lemma *def-cont-fix-ind*:
 $\llbracket f \equiv \text{fix} \cdot (\text{Abs-cfun } F); \text{cont } F; \text{adm } P; P \perp; \bigwedge x. P x \Longrightarrow P (F x) \rrbracket \Longrightarrow P f$
by (*simp add: fix-ind*)

lemma for proving rewrite rules

lemma *ssubst-lhs*: $\llbracket t = s; P s = Q \rrbracket \Longrightarrow P t = Q$
by *simp*

26.4 Initializing the fixrec package

ML-file $\langle \text{Tools/holcf-library.ML} \rangle$

ML-file $\langle \text{Tools/fixrec.ML} \rangle$

method-setup *fixrec-simp* = \langle
Scan.succeed (SIMPLE-METHOD' o Fixrec.fixrec-simp-tac)
 \rangle *pattern prover for fixrec constants*

setup \langle
Fixrec.add-matchers
 $[$ (*const-name* $\langle \text{up} \rangle$, *const-name* $\langle \text{match-up} \rangle$),
(*const-name* $\langle \text{sinl} \rangle$, *const-name* $\langle \text{match-sinl} \rangle$),
(*const-name* $\langle \text{sinr} \rangle$, *const-name* $\langle \text{match-sinr} \rangle$),
(*const-name* $\langle \text{spair} \rangle$, *const-name* $\langle \text{match-spair} \rangle$),
(*const-name* $\langle \text{Pair} \rangle$, *const-name* $\langle \text{match-Pair} \rangle$),
(*const-name* $\langle \text{ONE} \rangle$, *const-name* $\langle \text{match-ONE} \rangle$),
(*const-name* $\langle \text{TT} \rangle$, *const-name* $\langle \text{match-TT} \rangle$),
(*const-name* $\langle \text{FF} \rangle$, *const-name* $\langle \text{match-FF} \rangle$),
(*const-name* $\langle \text{bottom} \rangle$, *const-name* $\langle \text{match-bottom} \rangle$)]
 \rangle

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

end

27 Domain package support

```
theory Domain-Aux
imports Map-Functions Fixrec
begin
```

27.1 Continuous isomorphisms

A locale for continuous isomorphisms

```
locale iso =
  fixes abs :: 'a → 'b
  fixes rep :: 'b → 'a
  assumes abs-iso [simp]: rep·(abs·x) = x
  assumes rep-iso [simp]: abs·(rep·y) = y
begin
```

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

```
lemma abs-below: (abs·x ⊑ abs·y) = (x ⊑ y)
proof
  assume abs·x ⊑ abs·y
  then have rep·(abs·x) ⊑ rep·(abs·y) by (rule monofun-cfun-arg)
  then show x ⊑ y by simp
next
  assume x ⊑ y
  then show abs·x ⊑ abs·y by (rule monofun-cfun-arg)
qed
```

```
lemma rep-below: (rep·x ⊑ rep·y) = (x ⊑ y)
  by (rule iso.abs-below [OF swap])
```

```
lemma abs-eq: (abs·x = abs·y) = (x = y)
  by (simp add: po-eq-conv abs-below)
```

```
lemma rep-eq: (rep·x = rep·y) = (x = y)
  by (rule iso.abs-eq [OF swap])
```

```
lemma abs-strict: abs·⊥ = ⊥
proof -
  have ⊥ ⊑ rep·⊥ ..
  then have abs·⊥ ⊑ abs·(rep·⊥) by (rule monofun-cfun-arg)
  then have abs·⊥ ⊑ ⊥ by simp
  then show ?thesis by (rule bottomI)
qed
```

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

lemma *abs-defin'*: $abs.x = \perp \implies x = \perp$
proof –
have $x = rep.(abs.x)$ **by** *simp*
also assume $abs.x = \perp$
also note *rep-strict*
finally show $x = \perp$.
qed

lemma *rep-defin'*: $rep.z = \perp \implies z = \perp$
by (*rule iso.abs-defin'* [*OF swap*])

lemma *abs-defined*: $z \neq \perp \implies abs.z \neq \perp$
by (*erule contrapos-nn*, *erule abs-defin'*)

lemma *rep-defined*: $z \neq \perp \implies rep.z \neq \perp$
by (*rule iso.abs-defined* [*OF iso.swap*]) (*rule iso-axioms*)

lemma *abs-bottom-iff*: $(abs.x = \perp) = (x = \perp)$
by (*auto elim: abs-defin' intro: abs-strict*)

lemma *rep-bottom-iff*: $(rep.x = \perp) = (x = \perp)$
by (*rule iso.abs-bottom-iff* [*OF iso.swap*]) (*rule iso-axioms*)

lemma *casedist-rule*: $rep.x = \perp \vee P \implies x = \perp \vee P$
by (*simp add: rep-bottom-iff*)

lemma *compact-abs-rev*: $compact (abs.x) \implies compact x$
proof (*unfold compact-def*)
assume $adm (\lambda y. abs.x \not\sqsubseteq y)$
with *cont-Rep-cfun2*
have $adm (\lambda y. abs.x \not\sqsubseteq abs.y)$ **by** (*rule adm-subst*)
then show $adm (\lambda y. x \not\sqsubseteq y)$ **using** *abs-below* **by** *simp*
qed

lemma *compact-rep-rev*: $compact (rep.x) \implies compact x$
by (*rule iso.compact-abs-rev* [*OF iso.swap*]) (*rule iso-axioms*)

lemma *compact-abs*: $compact x \implies compact (abs.x)$
by (*rule compact-rep-rev*) *simp*

lemma *compact-rep*: $compact x \implies compact (rep.x)$
by (*rule iso.compact-abs* [*OF iso.swap*]) (*rule iso-axioms*)

lemma *iso-swap*: $(x = abs.y) = (rep.x = y)$
proof
assume $x = abs.y$

```

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* \implies *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 $d \cdot m \cdot (d \cdot n \cdot x) = d \cdot (min \ m \ n) \cdot x$
proof (*rule* *linorder-le-cases*)
assume $m \leq n$
with *chain* **have** $d \cdot m \sqsubseteq d \cdot n$ **by** (*rule* *chain-mono*)
then have $d \cdot m \cdot (d \cdot n \cdot x) = d \cdot m \cdot x$
 by (*rule* *deflation-below-comp1* [*OF* *defl* *defl*])
moreover from $\langle m \leq n \rangle$ **have** $min \ m \ n = m$ **by** *simp*
ultimately show *?thesis* **by** *simp*
next
assume $n \leq m$
with *chain* **have** $d \cdot n \sqsubseteq d \cdot m$ **by** (*rule* *chain-mono*)
then have $d \cdot m \cdot (d \cdot n \cdot x) = d \cdot n \cdot x$
 by (*rule* *deflation-below-comp2* [*OF* *defl* *defl*])
moreover from $\langle n \leq m \rangle$ **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-distrib)
  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-distrib)

```

```

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  $\langle \textit{chain } t \rangle$  have chain  $(\lambda n. t\ n\cdot x)$  by simp
  from  $\langle \textit{adm } P \rangle$  this  $\langle \bigwedge n. P\ (t\ n\cdot x) \rangle$  have  $P\ (\bigsqcup n. t\ n\cdot x)$  by (rule admD)
  with  $\langle \textit{chain } t \rangle$   $\langle (\bigsqcup n. t\ n) = ID \rangle$  show  $P\ x$  by (simp add: lub-distrib)
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 :: $(a::\textit{pcpo} \rightarrow a) \Rightarrow \textit{bool}$

where

decisive d $\longleftrightarrow (\forall x. d\cdot x = x \vee d\cdot x = \perp)$

```

lemma decisiveI:  $(\bigwedge x. d\cdot x = x \vee d\cdot x = \perp) \implies \textit{decisive } d$ 
  unfolding decisive-def by simp

```

lemma *decisive-cases*:

```

  assumes decisive d obtains  $d\cdot x = x \mid d\cdot x = \perp$ 
using assms unfolding decisive-def by auto

```

```

lemma decisive-bottom: decisive  $\perp$ 
  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  $\forall n. d n \cdot x = x \vee d n \cdot x = \perp$ 
    using decisive unfolding decisive-def by simp
  hence range ( $\lambda n. d n \cdot x$ )  $\subseteq \{x, \perp\}$ 
    by auto
  hence finite (range ( $\lambda n. d n \cdot x$ ))
    by (rule finite-subset, simp)
  with 1 have finite-chain ( $\lambda n. d n \cdot x$ )
    by (rule finite-range-imp-finch)
  then have  $\exists n. (\bigsqcup n. d n \cdot x) = d n \cdot x$ 

```


unfolding *finite-chain-def* **by** (*auto simp add: maxinch-is-thelub*)
with 2 **show** $\exists n. d\ n \cdot x = x$ **by** (*auto elim: sym*)
qed

lemma *lub-ID-finite-take-induct*:
assumes *chain d* **and** $(\bigsqcup n. d\ n) = ID$ **and** $\bigwedge n. \text{decisive } (d\ n)$
shows $(\bigwedge n. P\ (d\ n \cdot x)) \implies P\ x$
using *lub-ID-finite [OF assms]* **by** *metis*

27.4 Proofs about constructor functions

Lemmas for proving nchotomy rule:

lemma *ex-one-bottom-iff*:
 $(\exists x. P\ x \wedge x \neq \perp) = P\ ONE$
by *simp*

lemma *ex-up-bottom-iff*:
 $(\exists x. P\ x \wedge x \neq \perp) = (\exists x. P\ (up \cdot x))$
by (*safe, case-tac x, auto*)

lemma *ex-sprod-bottom-iff*:
 $(\exists y. P\ y \wedge y \neq \perp) =$
 $(\exists x\ y. (P\ (:x, y) \wedge x \neq \perp) \wedge y \neq \perp)$
by (*safe, case-tac y, auto*)

lemma *ex-sprod-up-bottom-iff*:
 $(\exists y. P\ y \wedge y \neq \perp) =$
 $(\exists x\ y. P\ (:up \cdot x, y) \wedge y \neq \perp)$
by (*safe, case-tac y, simp, case-tac x, auto*)

lemma *ex-ssum-bottom-iff*:
 $(\exists x. P\ x \wedge x \neq \perp) =$
 $((\exists x. P\ (sinl \cdot x) \wedge x \neq \perp) \vee$
 $(\exists x. P\ (sinr \cdot x) \wedge x \neq \perp))$
by (*safe, case-tac x, auto*)

lemma *exh-start*: $p = \perp \vee (\exists x. p = x \wedge x \neq \perp)$
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*: $\llbracket R; R \implies P \rrbracket \implies P$
by *auto*

lemma *exh-casedist1*: $((P \vee Q \implies R) \implies S) \equiv (\llbracket P \implies R; Q \implies R \rrbracket \implies S)$
by *rule auto*

lemma *exh-casedist2*: $(\exists x. P x \implies Q) \equiv (\bigwedge x. P x \implies Q)$
by *rule auto*

lemma *exh-casedist3*: $(P \wedge Q \implies R) \equiv (P \implies Q \implies R)$
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.ID.⊥.(sinr.x) = ⊥
sscase.ID.⊥.(sinl.x) = x
sscase.⊥.ID.(sinl.x) = ⊥
sscase.⊥.ID.(sinr.x) = x
fup.ID.(up.x) = x
by (*cases x = ⊥, simp, simp*)⁺

lemmas *sel-app-rules* =
sel-strict-rules sel-app-extra-rules
ssnd-spair sfst-spair up-defined spair-defined

lemmas *sel-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 \langle Tools/Domain/domain-take-proofs.ML \rangle

ML-file \langle Tools/cont-consts.ML \rangle

ML-file \langle Tools/cont-proc.ML \rangle

simproc-setup *cont (cont f) = \langle K ContProc.cont-proc \rangle*

ML-file \langle Tools/Domain/domain-constructors.ML \rangle

ML-file \langle Tools/Domain/domain-induction.ML \rangle

end

28 Domain package

theory *Domain*

imports *Representable Domain-Aux*

keywords

lazy unsafe and

domaindef domain :: thy-defn and

domain-isomorphism :: thy-decl

begin

default-sort *domain*

28.1 Representations of types

lemma *emb-prj: emb·((prj·x)::'a) = cast·DEFL('a)·x*
by (*simp add: cast-DEFL*)

lemma *emb-prj-emb:*

fixes *x :: 'a*

assumes *DEFL('a) \sqsubseteq DEFL('b)*

shows *emb·(prj·(emb·x) :: 'b) = 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) \sqsubseteq DEFL('b)*

shows *prj·(emb·(prj·x :: 'b)) = (prj·x :: 'a)*

apply (*rule emb-eq-iff [THEN iffD1]*)

apply (*simp only: emb-prj*)

apply (*rule deflation-below-comp1*)

apply (*rule deflation-cast*)

```

apply (rule deflation-cast)
apply (rule monofun-cfun-arg [OF assms])
done

```

Isomorphism lemmas used internally by the domain package:

```

lemma domain-abs-iso:
  fixes abs and rep
  assumes DEFL: DEFL('b) = DEFL('a)
  assumes abs-def: (abs :: 'a → 'b) ≡ prj oo emb
  assumes rep-def: (rep :: 'b → 'a) ≡ prj oo emb
  shows rep.(abs.x) = x
unfolding abs-def rep-def
by (simp add: emb-prj-emb DEFL)

```

```

lemma domain-rep-iso:
  fixes abs and rep
  assumes DEFL: DEFL('b) = DEFL('a)
  assumes abs-def: (abs :: 'a → 'b) ≡ prj oo emb
  assumes rep-def: (rep :: 'b → 'a) ≡ prj oo emb
  shows abs.(rep.x) = x
unfolding abs-def rep-def
by (simp add: emb-prj-emb DEFL)

```

28.2 Deflations as sets

```

definition defl-set :: 'a::bifinite defl ⇒ 'a set
where defl-set A = {x. cast.A.x = x}

```

```

lemma adm-defl-set: adm (λx. 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

```

28.3 Proving a subtype is representable

Temporarily relax type constraints.

```

setup <
  fold Sign.add-const-constraint
  [ (const-name <defl>, SOME typ <'a::pcpo itself ⇒ udom defl>)
    , (const-name <emb>, SOME typ <'a::pcpo → udom>)

```

```

, (const-name ⟨prj⟩, SOME typ ⟨udom → 'a::pcpo⟩)
, (const-name ⟨liftdefl⟩, SOME typ ⟨'a::pcpo itself ⇒ udom u defl⟩)
, (const-name ⟨liftemb⟩, SOME typ ⟨'a::pcpo u → udom u⟩)
, (const-name ⟨liftprj⟩, SOME typ ⟨udom u → 'a::pcpo u⟩ ]
>

```

lemma *typedef-domain-class*:

fixes *Rep* :: 'a::pcpo ⇒ udom

fixes *Abs* :: udom ⇒ 'a::pcpo

fixes *t* :: udom defl

assumes *type*: type-definition *Rep Abs* (defl-set *t*)

assumes *below*: (\sqsubseteq) ≡ $\lambda x y. Rep\ x \sqsubseteq Rep\ y$

assumes *emb*: $emb \equiv (\Lambda x. Rep\ x)$

assumes *prj*: $prj \equiv (\Lambda x. Abs\ (cast \cdot t \cdot x))$

assumes *defl*: $defl \equiv (\lambda a::'a\ itself. t)$

assumes *liftemb*: ($liftemb :: 'a\ u \rightarrow udom\ u$) ≡ *u-map.emb*

assumes *liftprj*: ($liftprj :: udom\ u \rightarrow 'a\ u$) ≡ *u-map.prj*

assumes *liftdefl*: ($liftdefl :: 'a\ itself \Rightarrow -$) ≡ ($\lambda t. liftdefl\ of \cdot DEFL('a)$)

shows *OFCLASS*('a, domain-class)

proof

have *emb-beta*: $\bigwedge x. emb \cdot x = Rep\ x$

unfolding *emb*

apply (*rule beta-cfun*)

apply (*rule typedef-cont-Rep* [*OF type below adm-defl-set cont-id*])

done

have *prj-beta*: $\bigwedge y. prj \cdot y = Abs\ (cast \cdot t \cdot y)$

unfolding *prj*

apply (*rule beta-cfun*)

apply (*rule typedef-cont-Abs* [*OF type below adm-defl-set*])

apply *simp-all*

done

have *prj-emb*: $\bigwedge x::'a. prj \cdot (emb \cdot x) = x$

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*: $\bigwedge y. emb \cdot (prj \cdot y :: 'a) = cast \cdot t \cdot y$

unfolding *prj-beta emb-beta*

by (*simp add: type-definition.Abs-inverse* [*OF type*])

show *ep-pair* (*emb* :: 'a → 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* ≡ ($\lambda a::'a::pcpo\ itself. t$)

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  $\Rightarrow$  udom defl>),
   (const-name <emb>, SOME typ <'a::domain  $\rightarrow$  udom>),
   (const-name <prj>, SOME typ <udom  $\rightarrow$  'a::domain>),
   (const-name <liftdefl>, SOME typ <'a::predomain itself  $\Rightarrow$  udom u defl>),
   (const-name <liftemb>, SOME typ <'a::predomain u  $\rightarrow$  udom u>),
   (const-name <liftprj>, SOME typ <udom u  $\rightarrow$  'a::predomain u>)]
  >
```

ML-file <*Tools/domaindef.ML*>

28.4 Isomorphic deflations

definition $isodefl :: ('a \rightarrow 'a) \Rightarrow udom\ defl \Rightarrow bool$
where $isodefl\ d\ t \iff cast \cdot t = emb\ oo\ d\ oo\ prj$

definition $isodefl' :: ('a::predomain \rightarrow 'a) \Rightarrow udom\ u\ defl \Rightarrow bool$
where $isodefl'\ d\ t \iff cast \cdot t = liftemb\ oo\ u\text{-map} \cdot d\ oo\ liftprj$

lemma $isodeflI: (\bigwedge x. cast \cdot t \cdot x = emb \cdot (d \cdot (prj \cdot x))) \implies isodefl\ d\ t$
unfolding *isodefl-def* **by** (*simp add: cfun-eqI*)

lemma $cast-isodefl: isodefl\ d\ t \implies cast \cdot t = (\bigwedge x. emb \cdot (d \cdot (prj \cdot x)))$
unfolding *isodefl-def* **by** (*simp add: cfun-eqI*)

lemma $isodefl-strict: isodefl\ d\ t \implies d \cdot \perp = \perp$
unfolding *isodefl-def*
by (*drule cfun-fun-cong [where x= \perp], simp*)

lemma *isodefl-imp-deflation*:
fixes $d :: 'a \rightarrow 'a$
assumes $isodefl\ d\ t$ **shows** *deflation d*
proof
note *assms [unfolded isodefl-def, simp]*
fix $x :: 'a$
show $d \cdot (d \cdot x) = d \cdot x$
using *cast.idem [of t emb \cdot x]* **by** *simp*
show $d \cdot x \sqsubseteq x$
using *cast.below [of t emb \cdot x]* **by** *simp*
qed

lemma $isodefl-ID-DEFL: isodefl\ (ID :: 'a \rightarrow 'a)\ DEFL('a)$
unfolding *isodefl-def* **by** (*simp add: cast-DEFL*)

lemma *isodefl-LIFTDEFL*:

isodefl' ($ID :: 'a \rightarrow 'a$) *LIFTDEFL*('a::predomain)

unfolding *isodefl'-def* **by** (*simp add: cast-liftdefl u-map-ID*)

lemma *isodefl-DEFL-imp-ID*: *isodefl* ($d :: 'a \rightarrow 'a$) *DEFL*('a) $\implies d = ID$

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* $\perp \perp$

unfolding *isodefl-def* **by** (*simp add: cfun-eq-iff*)

lemma *adm-isodefl*:

cont f \implies *cont g* \implies *adm* ($\lambda x. \text{isodefl } (f x) (g x)$)

unfolding *isodefl-def* **by** *simp*

lemma *isodefl-lub*:

assumes *chain d* **and** *chain t*

assumes $\bigwedge i. \text{isodefl } (d i) (t i)$

shows *isodefl* ($\bigsqcup i. d i$) ($\bigsqcup i. t i$)

using *assms* **unfolding** *isodefl-def*

by (*simp add: contlub-cfun-arg contlub-cfun-fun*)

lemma *isodefl-fix*:

assumes $\bigwedge d t. \text{isodefl } d t \implies \text{isodefl } (f \cdot d) (g \cdot t)$

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) = *DEFL*('a)

assumes *abs-def*: (*abs* :: $'a \rightarrow 'b$) \equiv *prj* oo *emb*

assumes *rep-def*: (*rep* :: $'b \rightarrow 'a$) \equiv *prj* oo *emb*

shows *isodefl* $d t \implies \text{isodefl } (abs \text{ oo } d \text{ oo } rep) t$

unfolding *isodefl-def*

by (*simp add: cfun-eq-iff assms prj-emb-prj emb-prj-emb*)

lemma *isodefl'-liftdefl-of*: *isodefl* $d t \implies \text{isodefl}' d (\text{liftdefl-of } t)$

unfolding *isodefl-def isodefl'-def*

by (*simp add: cast-liftdefl-of u-map-oo liftemb-eq liftprj-eq*)

lemma *isodefl-sfun*:

```

  isodefl d1 t1  $\implies$  isodefl d2 t2  $\implies$ 
    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  $\implies$  isodefl d2 t2  $\implies$ 
    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  $\implies$  isodefl d2 t2  $\implies$ 
    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  $\implies$  isodefl d2 t2  $\implies$ 
    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  $\implies$  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  $\implies$  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*:
 $encode\text{-}prod\text{-}u\text{-}(u\text{-}map\text{-}(prod\text{-}map\text{-}f\text{-}g)\text{-}(decode\text{-}prod\text{-}u\text{-}x))$
 $=\text{ }sprod\text{-}map\text{-}(u\text{-}map\text{-}f)\text{-}(u\text{-}map\text{-}g)\text{-}x$
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*:
 $encode\text{-}cfun\text{-}(cfun\text{-}map\text{-}f\text{-}g\text{-}(decode\text{-}cfun\text{-}x))$
 $=\text{ }sfun\text{-}map\text{-}(u\text{-}map\text{-}f)\text{-}g\text{-}x$
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*)

28.5 Setting up the domain package

named-theorems *domain-defl-simps* *theorems like DEFL('a t) = t-defl\$DEFL('a)*
and *domain-isodefl* *theorems like isodefl d t ==> isodefl (foo-map\$d) (foo-defl\$t)*

ML-file $\langle Tools/Domain/domain\text{-}isomorphism.ML \rangle$

ML-file $\langle Tools/Domain/domain\text{-}axioms.ML \rangle$

ML-file $\langle Tools/Domain/domain.ML \rangle$

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
```

29 A compact basis for powerdomains

```
theory Compact-Basis
imports Universal
begin
```

```
default-sort bifinite
```

29.1 A compact basis for powerdomains

```
definition pd-basis = {S::'a compact-basis set. finite S ∧ S ≠ {}}
```

```
typedef 'a pd-basis = pd-basis :: 'a compact-basis set set
  unfolding pd-basis-def
  apply (rule-tac x={-} in exI)
  apply simp
  done
```

```
lemma finite-Rep-pd-basis [simp]: finite (Rep-pd-basis u)
by (insert Rep-pd-basis [of u, unfolded pd-basis-def]) simp
```

```
lemma Rep-pd-basis-nonempty [simp]: Rep-pd-basis u ≠ {}
by (insert Rep-pd-basis [of u, unfolded pd-basis-def]) simp
```

The powerdomain basis type is countable.

```
lemma pd-basis-countable: ∃ f::'a pd-basis ⇒ nat. inj f
```

```
proof -
```

```
  obtain g :: 'a compact-basis ⇒ nat where inj g
    using compact-basis.countable ..
```

```
  hence image-g-eq: ⋀ A B. g ` A = g ` B ⟷ A = B
```

```

  by (rule inj-image-eq-iff)
  have inj ( $\lambda t. \text{set-encode } (g \text{ 'Rep-pd-basis } t)$ )
  by (simp add: inj-on-def set-encode-eq image-g-eq Rep-pd-basis-inject)
  thus ?thesis by - (rule exI)

```

qed

29.2 Unit and plus constructors

definition

```

  PDUnit :: 'a compact-basis  $\Rightarrow$  'a pd-basis where
  PDUnit = ( $\lambda x. \text{Abs-pd-basis } \{x\}$ )

```

definition

```

  PDPlus :: 'a pd-basis  $\Rightarrow$  'a pd-basis  $\Rightarrow$  'a pd-basis where
  PDPlus t u = Abs-pd-basis ( $\text{Rep-pd-basis } t \cup \text{Rep-pd-basis } u$ )

```

lemma Rep-PDUnit:

```

  Rep-pd-basis (PDUnit x) = {x}

```

unfolding PDUnit-def by (rule Abs-pd-basis-inverse) (simp add: pd-basis-def)

lemma Rep-PDPlus:

```

  Rep-pd-basis (PDPlus u v) = Rep-pd-basis u  $\cup$  Rep-pd-basis v

```

unfolding PDPlus-def by (rule Abs-pd-basis-inverse) (simp add: pd-basis-def)

lemma PDUnit-inject [simp]: ($\text{PDUnit } a = \text{PDUnit } b$) = ($a = b$)

unfolding Rep-pd-basis-inject [symmetric] Rep-PDUnit by simp

lemma PDPlus-assoc: $\text{PDPlus } (\text{PDPlus } t u) v = \text{PDPlus } t (\text{PDPlus } u v)$

unfolding Rep-pd-basis-inject [symmetric] Rep-PDPlus by (rule Un-assoc)

lemma PDPlus-commute: $\text{PDPlus } t u = \text{PDPlus } u t$

unfolding Rep-pd-basis-inject [symmetric] Rep-PDPlus by (rule Un-commute)

lemma PDPlus-absorb: $\text{PDPlus } t t = t$

unfolding Rep-pd-basis-inject [symmetric] Rep-PDPlus by (rule Un-absorb)

lemma pd-basis-induct1:

```

  assumes PDUnit:  $\bigwedge a. P (\text{PDUnit } a)$ 

```

```

  assumes PDPlus:  $\bigwedge a t. P t \Longrightarrow P (\text{PDPlus } (\text{PDUnit } a) t)$ 

```

```

  shows P x

```

apply (induct x, unfold pd-basis-def, clarify)

apply (erule (1) finite-ne-induct)

apply (cut-tac a=x in PDUnit)

apply (simp add: PDUnit-def)

apply (drule-tac a=x in PDPlus)

apply (simp add: PDUnit-def PDPlus-def)

```

  Abs-pd-basis-inverse [unfolded pd-basis-def]

```

done

```

lemma pd-basis-induct:
  assumes PDUUnit:  $\bigwedge a. P (PDUUnit a)$ 
  assumes PDPlus:  $\bigwedge t u. \llbracket P t; P u \rrbracket \implies P (PDPlus t u)$ 
  shows  $P x$ 
apply (induct x rule: pd-basis-induct1)
apply (rule PDUUnit, erule PDPlus [OF PDUUnit])
done

```

29.3 Fold operator

definition

```

fold-pd ::
  ('a compact-basis  $\Rightarrow$  'b::type)  $\Rightarrow$  ('b  $\Rightarrow$  'b  $\Rightarrow$  'b)  $\Rightarrow$  'a pd-basis  $\Rightarrow$  'b
  where fold-pd g f t = semilattice-set.F f (g ' Rep-pd-basis t)

```

lemma fold-pd-PDUUnit:

```

  assumes semilattice f
  shows fold-pd g f (PDUUnit x) = g x
proof –
  from assms interpret semilattice-set f by (rule semilattice-set.intro)
  show ?thesis by (simp add: fold-pd-def Rep-PDUUnit)
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

30 Upper powerdomain

theory UpperPD

imports Compact-Basis

begin

30.1 Basis preorder

definition

```

upper-le :: 'a pd-basis  $\Rightarrow$  'a pd-basis  $\Rightarrow$  bool (infix  $\leq_{\#}$  50) where
upper-le = ( $\lambda u v. \forall y \in \text{Rep-pd-basis } v. \exists x \in \text{Rep-pd-basis } u. x \sqsubseteq y$ )

```

lemma upper-le-refl [simp]: $t \leq_{\#} t$

unfolding upper-le-def **by** fast

lemma *upper-le-trans*: $\llbracket t \leq\# u; u \leq\# v \rrbracket \implies t \leq\# v$
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* $\leq\# t$
unfolding *upper-le-def Rep-PDUnit* **by** *simp*

lemma *PDUnit-upper-mono*: $x \sqsubseteq y \implies PDUnit\ x \leq\# PDUnit\ y$
unfolding *upper-le-def Rep-PDUnit* **by** *simp*

lemma *PDPlus-upper-mono*: $\llbracket s \leq\# t; u \leq\# v \rrbracket \implies PDPlus\ s\ u \leq\# PDPlus\ t\ v$
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *PDPlus-upper-le*: $PDPlus\ t\ u \leq\# t$
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *upper-le-PDUnit-PDUnit-iff [simp]*:
 $(PDUnit\ a \leq\# PDUnit\ b) = (a \sqsubseteq b)$
unfolding *upper-le-def Rep-PDUnit* **by** *fast*

lemma *upper-le-PDPlus-PDUnit-iff*:
 $(PDPlus\ t\ u \leq\# PDUnit\ a) = (t \leq\# PDUnit\ a \vee u \leq\# PDUnit\ a)$
unfolding *upper-le-def Rep-PDPlus Rep-PDUnit* **by** *fast*

lemma *upper-le-PDPlus-iff*: $(t \leq\# PDPlus\ u\ v) = (t \leq\# u \wedge t \leq\# v)$
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *upper-le-induct [induct set: upper-le]*:
assumes *le*: $t \leq\# u$
assumes *1*: $\bigwedge a\ b. a \sqsubseteq b \implies P\ (PDUnit\ a)\ (PDUnit\ b)$
assumes *2*: $\bigwedge t\ u\ a. P\ t\ (PDUnit\ a) \implies P\ (PDPlus\ t\ u)\ (PDUnit\ a)$
assumes *3*: $\bigwedge t\ u\ v. \llbracket P\ t\ u; P\ t\ v \rrbracket \implies P\ t\ (PDPlus\ u\ v)$
shows $P\ t\ u$
using *le* **apply** (*induct u arbitrary: t rule: pd-basis-induct*)
apply (*erule rev-mp*)
apply (*induct-tac t rule: pd-basis-induct*)
apply (*simp add: 1*)
apply (*simp add: upper-le-PDPlus-PDUnit-iff*)
apply (*simp add: 2*)
apply (*subst PDPlus-commute*)
apply (*simp add: 2*)

```

apply (simp add: upper-le-PDPlus-iff 3)
done

```

30.2 Type definition

```

typedef 'a upper-pd ((-'#)) =
  {S::'a pd-basis set. upper-le.ideal S}
by (rule upper-le.ex-ideal)

```

```

instantiation upper-pd :: (bifinite) below
begin

```

definition

$$x \sqsubseteq y \longleftrightarrow \text{Rep-upper-pd } x \subseteq \text{Rep-upper-pd } y$$

```

instance ..
end

```

```

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

```

```

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

```

definition

```

upper-principal :: 'a pd-basis  $\Rightarrow$  'a upper-pd where
upper-principal t = Abs-upper-pd {u. u  $\leq\#$  t}

```

interpretation upper-pd:

```

ideal-completion upper-le upper-principal Rep-upper-pd
using type-definition-upper-pd below-upper-pd-def
using upper-principal-def pd-basis-countable
by (rule upper-le.typedef-ideal-completion)

```

Upper powerdomain is pointed

```

lemma upper-pd-minimal: upper-principal (PDUnit compact-bot)  $\sqsubseteq$  ys
by (induct ys rule: upper-pd.principal-induct, simp, simp)

```

```

instance upper-pd :: (bifinite) pcpo
by intro-classes (fast intro: upper-pd-minimal)

```

```

lemma inst-upper-pd-pcpo:  $\perp$  = upper-principal (PDUnit compact-bot)
by (rule upper-pd-minimal [THEN bottomI, symmetric])

```

30.3 Monadic unit and plus

definition

upper-unit :: 'a \rightarrow 'a *upper-pd* **where**
upper-unit = *compact-basis.extension* ($\lambda a.$ *upper-principal* (*PDUnit* a))

definition

upper-plus :: 'a *upper-pd* \rightarrow 'a *upper-pd* \rightarrow 'a *upper-pd* **where**
upper-plus = *upper-pd.extension* ($\lambda t.$ *upper-pd.extension* ($\lambda u.$
upper-principal (*PDPlus* t u)))

abbreviation

upper-add :: 'a *upper-pd* \Rightarrow 'a *upper-pd* \Rightarrow 'a *upper-pd*
 (**infixl** $\cup\#$ 65) **where**
xs $\cup\#$ *ys* == *upper-plus*.*xs*.*ys*

syntax

-upper-pd :: *args* \Rightarrow *logic* ($\{-\}\#$)

translations

$\{x, xs\}\#$ == $\{x\}\# \cup\# \{xs\}\#$
 $\{x\}\#$ == *CONST* *upper-unit*.*x*

lemma *upper-unit-Rep-compact-basis* [*simp*]:

$\{Rep\text{-compact-basis } a\}\#$ = *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 $\cup\#$ *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* $\cup\#$ *ys*) $\cup\#$ *zs* = *xs* $\cup\#$ (*ys* $\cup\#$ *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* $\cup\#$ *ys* = *ys* $\cup\#$ *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* $\cup\#$ *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*: $xs \cup\# ys \sqsubseteq xs$
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*: $xs \cup\# ys \sqsubseteq ys$
by (*subst upper-plus-commute, rule upper-plus-below1*)

lemma *upper-plus-greatest*: $[xs \sqsubseteq ys; xs \sqsubseteq zs] \implies xs \sqsubseteq ys \cup\# zs$
apply (*subst upper-plus-absorb [of xs, symmetric]*)
apply (*erule (1) monofun-cfun [OF monofun-cfun-arg]*)
done

lemma *upper-below-plus-iff* [*simp*]:
 $xs \sqsubseteq ys \cup\# zs \longleftrightarrow xs \sqsubseteq ys \wedge xs \sqsubseteq zs$
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*]:
 $xs \cup\# ys \sqsubseteq \{z\}\# \longleftrightarrow xs \sqsubseteq \{z\}\# \vee ys \sqsubseteq \{z\}\#$
apply (*induct xs rule: upper-pd.principal-induct, simp*)
apply (*induct ys rule: upper-pd.principal-induct, simp*)
apply (*induct z rule: compact-basis.principal-induct, simp*)
apply (*simp add: upper-le-PDPlus-PDUnit-iff*)
done

lemma *upper-unit-below-iff* [*simp*]: $\{x\}\# \sqsubseteq \{y\}\# \longleftrightarrow x \sqsubseteq y$
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*]: $\{x\}\# = \{y\}\# \longleftrightarrow x = y$
unfolding *po-eq-conv* **by** *simp*

lemma *upper-unit-strict* [*simp*]: $\{\perp\}\# = \perp$
using *upper-unit-Rep-compact-basis* [of *compact-bot*]
by (*simp add: inst-upper-pd-pcpo*)

lemma *upper-plus-strict1* [*simp*]: $\perp \cup\# ys = \perp$
by (*rule bottomI, rule upper-plus-below1*)

lemma *upper-plus-strict2* [*simp*]: $xs \cup\# \perp = \perp$
by (*rule bottomI, rule upper-plus-below2*)

lemma *upper-unit-bottom-iff* [*simp*]: $\{x\}\# = \perp \longleftrightarrow x = \perp$
unfolding *upper-unit-strict* [*symmetric*] **by** (*rule upper-unit-eq-iff*)

lemma *upper-plus-bottom-iff* [*simp*]:
 $xs \cup\# ys = \perp \longleftrightarrow xs = \perp \vee ys = \perp$
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*: $\text{compact } x \implies \text{compact } \{x\}\#$
by (*auto dest!: compact-basis.compact-imp-principal*)

lemma *compact-upper-unit-iff* [*simp*]: $\text{compact } \{x\}\# \longleftrightarrow \text{compact } x$
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*]:
 $\llbracket \text{compact } xs; \text{compact } ys \rrbracket \implies \text{compact } (xs \cup\# ys)$
by (*auto dest!: upper-pd.compact-imp-principal*)

30.4 Induction rules

lemma *upper-pd-induct1*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x. P \{x\}\#$

```

assumes insert:  $\bigwedge x \text{ ys. } \llbracket P \{x\}\#; P \text{ ys} \rrbracket \implies P (\{x\}\# \cup\# \text{ ys})$ 
shows  $P (xs::'a \text{ upper-pd})$ 
apply (induct xs rule: upper-pd.principal-induct, rule P)
apply (induct-tac a rule: pd-basis-induct1)
apply (simp only: upper-unit-Rep-compact-basis [symmetric])
apply (rule unit)
apply (simp only: upper-unit-Rep-compact-basis [symmetric]
           upper-plus-principal [symmetric])
apply (erule insert [OF unit])
done

```

```

lemma upper-pd-induct
  [case-names adm upper-unit upper-plus, induct type: upper-pd]:
assumes P: adm P
assumes unit:  $\bigwedge x. P \{x\}\#$ 
assumes plus:  $\bigwedge xs \text{ ys. } \llbracket P xs; P \text{ ys} \rrbracket \implies P (xs \cup\# \text{ ys})$ 
shows  $P (xs::'a \text{ upper-pd})$ 
apply (induct xs rule: upper-pd.principal-induct, rule P)
apply (induct-tac a rule: pd-basis-induct)
apply (simp only: upper-unit-Rep-compact-basis [symmetric] unit)
apply (simp only: upper-plus-principal [symmetric] plus)
done

```

30.5 Monadic bind

definition

```

upper-bind-basis ::
  'a pd-basis  $\Rightarrow$  ('a  $\rightarrow$  'b upper-pd)  $\rightarrow$  'b upper-pd where
upper-bind-basis = fold-pd
  ( $\lambda a. \bigwedge f. f \cdot (\text{Rep-compact-basis } a)$ )
  ( $\lambda x y. \bigwedge f. x \cdot f \cup\# y \cdot f$ )

```

lemma ACI-upper-bind:

```

semilattice ( $\lambda x y. \bigwedge f. x \cdot f \cup\# y \cdot f$ )
apply unfold-locales
apply (simp add: upper-plus-assoc)
apply (simp add: upper-plus-commute)
apply (simp add: eta-cfun)
done

```

lemma upper-bind-basis-simps [simp]:

```

upper-bind-basis (PDUnt a) =
  ( $\bigwedge f. f \cdot (\text{Rep-compact-basis } a)$ )
upper-bind-basis (PDPlus t u) =
  ( $\bigwedge f. \text{upper-bind-basis } t \cdot f \cup\# \text{upper-bind-basis } u \cdot f$ )
unfolding upper-bind-basis-def
apply –
apply (rule fold-pd-PDUnt [OF ACI-upper-bind])
apply (rule fold-pd-PDPlus [OF ACI-upper-bind])

```

done

lemma *upper-bind-basis-mono*:

$t \leq_{\#} u \implies \text{upper-bind-basis } t \sqsubseteq \text{upper-bind-basis } u$

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

$\text{upper-bind} :: 'a \text{ upper-pd} \rightarrow ('a \rightarrow 'b \text{ upper-pd}) \rightarrow 'b \text{ upper-pd}$ **where**
 $\text{upper-bind} = \text{upper-pd.extension upper-bind-basis}$

syntax

$\text{-upper-bind} :: [\text{logic}, \text{logic}, \text{logic}] \Rightarrow \text{logic}$

$((\exists \bigcup_{\#} \in \cdot / \cdot) [0, 0, 10] 10)$

translations

$\bigcup_{\#} x \in xs. e == \text{CONST upper-bind} \cdot xs (\Lambda x. e)$

lemma *upper-bind-principal [simp]*:

$\text{upper-bind} \cdot (\text{upper-principal } t) = \text{upper-bind-basis } t$

unfolding *upper-bind-def*

apply (*rule upper-pd.extension-principal*)

apply (*erule upper-bind-basis-mono*)

done

lemma *upper-bind-unit [simp]*:

$\text{upper-bind} \cdot \{x\}_{\#} \cdot f = f \cdot x$

by (*induct x rule: compact-basis.principal-induct, simp, simp*)

lemma *upper-bind-plus [simp]*:

$\text{upper-bind} \cdot (xs \bigcup_{\#} ys) \cdot f = \text{upper-bind} \cdot xs \cdot f \bigcup_{\#} \text{upper-bind} \cdot ys \cdot f$

by (*induct xs rule: upper-pd.principal-induct, simp,*

induct ys rule: upper-pd.principal-induct, simp, simp)

lemma *upper-bind-strict [simp]*: $\text{upper-bind} \cdot \perp \cdot f = f \cdot \perp$

unfolding *upper-unit-strict [symmetric]* **by** (*rule upper-bind-unit*)

lemma *upper-bind-bind*:

$\text{upper-bind} \cdot (\text{upper-bind} \cdot xs \cdot f) \cdot g = \text{upper-bind} \cdot xs \cdot (\Lambda x. \text{upper-bind} \cdot (f \cdot x) \cdot g)$

by (*induct xs, simp-all*)

30.6 Map

definition

$\text{upper-map} :: ('a \rightarrow 'b) \rightarrow 'a \text{ upper-pd} \rightarrow 'b \text{ upper-pd}$ **where**

$upper-map = (\Lambda f xs. upper-bind.xs(\Lambda x. \{f.x\}\#))$

lemma *upper-map-unit* [*simp*]:

$upper-map.f.\{x\}\# = \{f.x\}\#$

unfolding *upper-map-def* **by** *simp*

lemma *upper-map-plus* [*simp*]:

$upper-map.f.(xs \cup\# ys) = upper-map.f.xs \cup\# upper-map.f.y$ s

unfolding *upper-map-def* **by** *simp*

lemma *upper-map-bottom* [*simp*]: $upper-map.f.\perp = \{f.\perp\}\#$

unfolding *upper-map-def* **by** *simp*

lemma *upper-map-ident*: $upper-map.(\Lambda x. x).xs = xs$

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*:

$upper-map.f.(upper-map.g.xs) = upper-map.(\Lambda x. f.(g.x)).xs$

by (*induct xs rule: upper-pd-induct, simp-all*)

lemma *upper-bind-map*:

$upper-bind.(upper-map.f.xs).g = upper-bind.xs(\Lambda x. g.(f.x))$

by (*simp add: upper-map-def upper-bind-bind*)

lemma *upper-map-bind*:

$upper-map.f.(upper-bind.xs.g) = upper-bind.xs(\Lambda x. upper-map.f.(g.x))$

by (*simp add: upper-map-def upper-bind-bind*)

lemma *ep-pair-upper-map*: $ep-pair\ e\ p \implies ep-pair\ (upper-map.e)\ (upper-map.p)$

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*: $deflation\ d \implies deflation\ (upper-map.d)$

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

```

30.7 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  $\exists (a::nat \Rightarrow 'a \text{ upper-pd} \rightarrow 'a \text{ upper-pd}). \text{approx-chain } a$ 
  using bifinite [where 'a='a]
  by (fast intro!: approx-chain-upper-map)
qed

```

30.8 Join

definition

```

upper-join :: 'a upper-pd upper-pd  $\rightarrow$  'a upper-pd where
upper-join = ( $\Lambda$  xss. upper-bind·xss·( $\Lambda$  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  $\cup$ # yss) = upper-join·xss  $\cup$ # upper-join·yss

```

unfolding *upper-join-def* **by** *simp*

lemma *upper-join-bottom* [*simp*]: *upper-join*· \perp = \perp

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

31 Lower powerdomain

theory *LowerPD*

imports *Compact-Basis*

begin

31.1 Basis preorder

definition

```

lower-le :: 'a pd-basis  $\Rightarrow$  'a pd-basis  $\Rightarrow$  bool (infix  $\leq$  50) where

```

```

lower-le = ( $\lambda u v. \forall x \in \text{Rep-pd-basis } u. \exists y \in \text{Rep-pd-basis } v. x \sqsubseteq y$ )

```

lemma *lower-le-refl* [*simp*]: $t \leq_b t$
unfolding *lower-le-def* **by** *fast*

lemma *lower-le-trans*: $\llbracket t \leq_b u; u \leq_b v \rrbracket \implies t \leq_b v$
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* $\leq_b t$
unfolding *lower-le-def Rep-PDUnit*
by (*simp*, *rule* *Rep-pd-basis-nonempty* [*folded ex-in-conv*])

lemma *PDUnit-lower-mono*: $x \sqsubseteq y \implies PDUnit\ x \leq_b PDUnit\ y$
unfolding *lower-le-def Rep-PDUnit* **by** *fast*

lemma *PDPlus-lower-mono*: $\llbracket s \leq_b t; u \leq_b v \rrbracket \implies PDPlus\ s\ u \leq_b PDPlus\ t\ v$
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *PDPlus-lower-le*: $t \leq_b PDPlus\ t\ u$
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *lower-le-PDUnit-PDUnit-iff* [*simp*]:
 $(PDUnit\ a \leq_b PDUnit\ b) = (a \sqsubseteq b)$
unfolding *lower-le-def Rep-PDUnit* **by** *fast*

lemma *lower-le-PDUnit-PDPlus-iff*:
 $(PDUnit\ a \leq_b PDPlus\ t\ u) = (PDUnit\ a \leq_b t \vee PDUnit\ a \leq_b u)$
unfolding *lower-le-def Rep-PDPlus Rep-PDUnit* **by** *fast*

lemma *lower-le-PDPlus-iff*: $(PDPlus\ t\ u \leq_b v) = (t \leq_b v \wedge u \leq_b v)$
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *lower-le-induct* [*induct set: lower-le*]:
assumes *le*: $t \leq_b u$
assumes 1: $\bigwedge a\ b. a \sqsubseteq b \implies P\ (PDUnit\ a)\ (PDUnit\ b)$
assumes 2: $\bigwedge t\ u\ a. P\ (PDUnit\ a)\ t \implies P\ (PDUnit\ a)\ (PDPlus\ t\ u)$
assumes 3: $\bigwedge t\ u\ v. \llbracket P\ t\ v; P\ u\ v \rrbracket \implies P\ (PDPlus\ t\ u)\ v$
shows $P\ t\ u$
using *le*
apply (*induct* *t arbitrary*: *u rule: pd-basis-induct*)
apply (*erule* *rev-mp*)
apply (*induct-tac* *u rule: pd-basis-induct*)

```

apply (simp add: 1)
apply (simp add: lower-le-PDUnit-PDPlus-iff)
apply (simp add: 2)
apply (subst PDPlus-commute)
apply (simp add: 2)
apply (simp add: lower-le-PDPlus-iff 3)
done

```

31.2 Type definition

```

typedef 'a lower-pd ((-'b)) =
  {S::'a pd-basis set. lower-le.ideal S}
by (rule lower-le.ex-ideal)

```

```

instantiation lower-pd :: (bifinite) below
begin

```

definition

$$x \sqsubseteq y \iff \text{Rep-lower-pd } x \subseteq \text{Rep-lower-pd } y$$

```

instance ..
end

```

```

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

```

```

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

```

definition

```

lower-principal :: 'a pd-basis  $\Rightarrow$  'a lower-pd where
lower-principal t = Abs-lower-pd {u. u  $\leq$  t}

```

interpretation lower-pd:

```

ideal-completion lower-le lower-principal Rep-lower-pd
using type-definition-lower-pd below-lower-pd-def
using lower-principal-def pd-basis-countable
by (rule lower-le.typedef-ideal-completion)

```

Lower powerdomain is pointed

```

lemma lower-pd-minimal: lower-principal (PDUnit compact-bot)  $\sqsubseteq$  ys
by (induct ys rule: lower-pd.principal-induct, simp, simp)

```

```

instance lower-pd :: (bifinite) pcpo
by intro-classes (fast intro: lower-pd-minimal)

```

```

lemma inst-lower-pd-pcpo:  $\perp$  = lower-principal (PDUnit compact-bot)

```


by (rule lower-pd-minimal [THEN bottomI, symmetric])

31.3 Monadic unit and plus

definition

lower-unit :: 'a \rightarrow 'a lower-pd **where**
 lower-unit = compact-basis.extension (λa . lower-principal (PDUUnit a))

definition

lower-plus :: 'a lower-pd \rightarrow 'a lower-pd \rightarrow 'a lower-pd **where**
 lower-plus = lower-pd.extension (λt . lower-pd.extension (λu .
 lower-principal (PDPlus t u)))

abbreviation

lower-add :: 'a lower-pd \Rightarrow 'a lower-pd \Rightarrow 'a lower-pd
 (infixl $\cup b$ 65) **where**
 xs $\cup b$ ys == lower-plus.xs.ys

syntax

-lower-pd :: args \Rightarrow logic ($\{-\}b$)

translations

$\{x, xs\}b$ == $\{x\}b \cup b \{xs\}b$
 $\{x\}b$ == CONST lower-unit.x

lemma lower-unit-Rep-compact-basis [simp]:

$\{Rep\text{-compact-basis } a\}b = \text{lower-principal } (PDUUnit a)$

unfolding lower-unit-def

by (simp add: compact-basis.extension-principal PDUUnit-lower-mono)

lemma lower-plus-principal [simp]:

lower-principal t $\cup b$ lower-principal u = lower-principal (PDPlus t u)

unfolding lower-plus-def

by (simp add: lower-pd.extension-principal

lower-pd.extension-mono PDPlus-lower-mono)

interpretation lower-add: semilattice lower-add **proof**

fix xs ys zs :: 'a lower-pd

show (xs $\cup b$ ys) $\cup b$ zs = xs $\cup b$ (ys $\cup b$ 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 $\cup b$ ys = ys $\cup b$ 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  $xs \cup b \ xs = xs$ 
  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  $\cup b$  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  $\cup b$  ys
by (subst lower-plus-commute, rule lower-plus-below1)

```

```

lemma lower-plus-least:  $\llbracket xs \sqsubseteq zs; ys \sqsubseteq zs \rrbracket \implies xs \cup b \ ys \sqsubseteq zs$ 
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  $\cup b$  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}b  $\sqsubseteq$  ys  $\cup b$  zs  $\longleftrightarrow$  {x}b  $\sqsubseteq$  ys  $\vee$  {x}b  $\sqsubseteq$  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*]: $\{x\}^\flat \sqsubseteq \{y\}^\flat \iff x \sqsubseteq y$
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*]: $\{x\}^\flat = \{y\}^\flat \iff x = y$
by (*simp add: po-eq-conv*)

lemma *lower-unit-strict* [*simp*]: $\{\perp\}^\flat = \perp$
using *lower-unit-Rep-compact-basis* [*of compact-bot*]
by (*simp add: inst-lower-pd-pcpo*)

lemma *lower-unit-bottom-iff* [*simp*]: $\{x\}^\flat = \perp \iff x = \perp$
unfolding *lower-unit-strict* [*symmetric*] **by** (*rule lower-unit-eq-iff*)

lemma *lower-plus-bottom-iff* [*simp*]:
 $xs \cupb ys = \perp \iff xs = \perp \wedge ys = \perp$
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*]: $\perp \cupb ys = ys$
apply (*rule below-antisym* [*OF - lower-plus-below2*])
apply (*simp add: lower-plus-least*)
done

lemma *lower-plus-strict2* [*simp*]: $xs \cupb \perp = xs$
apply (*rule below-antisym* [*OF - lower-plus-below1*])
apply (*simp add: lower-plus-least*)
done

lemma *compact-lower-unit*: $\text{compact } x \implies \text{compact } \{x\}^\flat$
by (*auto dest!: compact-basis.compact-imp-principal*)

lemma *compact-lower-unit-iff* [*simp*]: $\text{compact } \{x\}^\flat \iff \text{compact } x$
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*]:
 $\llbracket \text{compact } xs; \text{ compact } ys \rrbracket \implies \text{compact } (xs \cup b \text{ } ys)$
by (*auto dest!*: *lower-pd.compact-imp-principal*)

31.4 Induction rules

lemma *lower-pd-induct1*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x. P \{x\}b$
assumes *insert*:
 $\bigwedge x \text{ } ys. \llbracket P \{x\}b; P \text{ } ys \rrbracket \implies P (\{x\}b \cup b \text{ } ys)$
shows *P* (*xs*::'a *lower-pd*)
apply (*induct xs rule: lower-pd.principal-induct, rule P*)
apply (*induct-tac a rule: pd-basis-induct1*)
apply (*simp only: lower-unit-Rep-compact-basis [symmetric]*)
apply (*rule unit*)
apply (*simp only: lower-unit-Rep-compact-basis [symmetric]*
lower-plus-principal [symmetric])
apply (*erule insert [OF unit]*)
done

lemma *lower-pd-induct*
[*case-names adm lower-unit lower-plus, induct type: lower-pd*]:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x. P \{x\}b$
assumes *plus*: $\bigwedge xs \text{ } ys. \llbracket P \text{ } xs; P \text{ } ys \rrbracket \implies P (xs \cup b \text{ } ys)$
shows *P* (*xs*::'a *lower-pd*)
apply (*induct xs rule: lower-pd.principal-induct, rule P*)
apply (*induct-tac a rule: pd-basis-induct*)
apply (*simp only: lower-unit-Rep-compact-basis [symmetric] unit*)
apply (*simp only: lower-plus-principal [symmetric] plus*)
done

31.5 Monadic bind

definition

lower-bind-basis ::
'a *pd-basis* \Rightarrow ('a \rightarrow 'b *lower-pd*) \rightarrow 'b *lower-pd* **where**
lower-bind-basis = *fold-pd*
($\lambda a. \Lambda f. f \cdot (\text{Rep-compact-basis } a)$)
($\lambda x \text{ } y. \Lambda f. x \cdot f \cup b \text{ } y \cdot f$)

lemma *ACI-lower-bind*:
semilattice ($\lambda x \text{ } y. \Lambda f. x \cdot f \cup b \text{ } y \cdot f$)
apply *unfold-locales*
apply (*simp add: lower-plus-assoc*)
apply (*simp add: lower-plus-commute*)
apply (*simp add: eta-cfun*)
done

lemma *lower-bind-basis-simps* [simp]:
lower-bind-basis (PDUnit a) =
 ($\Lambda f. f \cdot (\text{Rep-compact-basis } a)$)
lower-bind-basis (PDPlus t u) =
 ($\Lambda f. \text{lower-bind-basis } t \cdot f \cup_b \text{lower-bind-basis } u \cdot f$)
unfolding *lower-bind-basis-def*
apply –
apply (rule *fold-pd-PDUnit* [OF ACI-lower-bind])
apply (rule *fold-pd-PDPlus* [OF ACI-lower-bind])
done

lemma *lower-bind-basis-mono*:
 $t \leq_b u \implies \text{lower-bind-basis } t \sqsubseteq \text{lower-bind-basis } u$
unfolding *cfun-below-iff*
apply (erule *lower-le-induct*, safe)
apply (simp add: *monofun-cfun*)
apply (simp add: *rev-below-trans* [OF *lower-plus-below1*])
apply *simp*
done

definition
lower-bind :: 'a lower-pd \rightarrow ('a \rightarrow 'b lower-pd) \rightarrow 'b lower-pd **where**
lower-bind = *lower-pd.extension lower-bind-basis*

syntax
 $\text{-lower-bind} :: [\text{logic}, \text{logic}, \text{logic}] \Rightarrow \text{logic}$
 $((\exists \cup_b \in \cdot / \cdot) [0, 0, 10] 10)$

translations
 $\cup_b x \in xs. e == \text{CONST } \text{lower-bind} \cdot xs \cdot (\Lambda x. e)$

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 \cup_b ys).f = *lower-bind*.xs.f \cup_b *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*.\perp.f = f.\perp
unfolding *lower-unit-strict* [symmetric] **by** (rule *lower-bind-unit*)

lemma *lower-bind-bind*:

$$\text{lower-bind} \cdot (\text{lower-bind} \cdot xs \cdot f) \cdot g = \text{lower-bind} \cdot xs \cdot (\Lambda x. \text{lower-bind} \cdot (f \cdot x) \cdot g)$$

by (*induct xs, simp-all*)

31.6 Map

definition

$$\begin{aligned} \text{lower-map} &:: ('a \rightarrow 'b) \rightarrow 'a \text{ lower-pd} \rightarrow 'b \text{ lower-pd} \textbf{ where} \\ \text{lower-map} &= (\Lambda f \ xs. \text{lower-bind} \cdot xs \cdot (\Lambda x. \{f \cdot x\}b)) \end{aligned}$$

lemma *lower-map-unit* [*simp*]:

$$\text{lower-map} \cdot f \cdot \{x\}b = \{f \cdot x\}b$$

unfolding *lower-map-def* **by** *simp*

lemma *lower-map-plus* [*simp*]:

$$\text{lower-map} \cdot f \cdot (xs \cup b \ ys) = \text{lower-map} \cdot f \cdot xs \cup b \ \text{lower-map} \cdot f \cdot ys$$

unfolding *lower-map-def* **by** *simp*

lemma *lower-map-bottom* [*simp*]: $\text{lower-map} \cdot f \cdot \perp = \{f \cdot \perp\}b$

unfolding *lower-map-def* **by** *simp*

lemma *lower-map-ident*: $\text{lower-map} \cdot (\Lambda x. x) \cdot xs = xs$

by (*induct xs rule: lower-pd-induct, simp-all*)

lemma *lower-map-ID*: $\text{lower-map} \cdot ID = ID$

by (*simp add: cfun-eq-iff ID-def lower-map-ident*)

lemma *lower-map-map*:

$$\text{lower-map} \cdot f \cdot (\text{lower-map} \cdot g \cdot xs) = \text{lower-map} \cdot (\Lambda x. f \cdot (g \cdot x)) \cdot xs$$

by (*induct xs rule: lower-pd-induct, simp-all*)

lemma *lower-bind-map*:

$$\text{lower-bind} \cdot (\text{lower-map} \cdot f \cdot xs) \cdot g = \text{lower-bind} \cdot xs \cdot (\Lambda x. g \cdot (f \cdot x))$$

by (*simp add: lower-map-def lower-bind-bind*)

lemma *lower-map-bind*:

$$\text{lower-map} \cdot f \cdot (\text{lower-bind} \cdot xs \cdot g) = \text{lower-bind} \cdot xs \cdot (\Lambda x. \text{lower-map} \cdot f \cdot (g \cdot x))$$

by (*simp add: lower-map-def lower-bind-bind*)

lemma *ep-pair-lower-map*: $\text{ep-pair } e \ p \implies \text{ep-pair } (\text{lower-map} \cdot e) \ (\text{lower-map} \cdot p)$

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*: $\text{deflation } d \implies \text{deflation } (\text{lower-map} \cdot d)$

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 ( $\lambda x. d \cdot x$ )) by (rule d.finite-range)
hence finite (Rep-compact-basis - ‘ range ( $\lambda x. d \cdot x$ )’)
  by (rule finite-vimageI, simp add: inj-on-def Rep-compact-basis-inject)
hence finite (Pow (Rep-compact-basis - ‘ range ( $\lambda x. d \cdot x$ )’)) by simp
hence finite (Rep-pd-basis - ‘ (Pow (Rep-compact-basis - ‘ range ( $\lambda x. d \cdot x$ )’)))
  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 ( $\lambda x. d \cdot x$ )’))’)) by simp
hence finite (range ( $\lambda xs. \text{lower-map } d \cdot xs$ ))
  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  $\exists b. d \cdot (\text{Rep-compact-basis } a) = \text{Rep-compact-basis } b$ )
  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  $\cdot$  xs = xs}
  by (rule finite-range-imp-finite-fixes)
qed

```

31.7 Lower powerdomain is bifinite

lemma *approx-chain-lower-map*:
assumes *approx-chain a*
shows *approx-chain* ($\lambda i. \text{lower-map}\cdot(a\ i)$)
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 $\exists (a::\text{nat} \Rightarrow 'a \text{ lower-pd} \rightarrow 'a \text{ lower-pd}). \text{approx-chain } a$
using *bifinite* [**where** *'a='a*]
by (*fast intro!: approx-chain-lower-map*)

qed

31.8 Join

definition

lower-join :: *'a lower-pd lower-pd* \rightarrow *'a lower-pd* **where**
lower-join = ($\Lambda \text{ xss}. \text{lower-bind}\cdot\text{xss}\cdot(\Lambda \text{ xs}. \text{xs})$)

lemma *lower-join-unit* [*simp*]:

lower-join·{*xs*} \flat = *xs*

unfolding *lower-join-def* **by** *simp*

lemma *lower-join-plus* [*simp*]:

lower-join·(*xss* \cup *yss*) = *lower-join*·*xss* \cup *lower-join*·*yss*

unfolding *lower-join-def* **by** *simp*

lemma *lower-join-bottom* [*simp*]: *lower-join*· \perp = \perp

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

32 Convex powerdomain

```
theory ConvexPD
imports UpperPD LowerPD
begin
```

32.1 Basis preorder

definition

```
convex-le :: 'a pd-basis  $\Rightarrow$  'a pd-basis  $\Rightarrow$  bool (infix  $\leq_{\mathfrak{h}}$  50) where
convex-le = ( $\lambda u v. u \leq_{\#} v \wedge u \leq_{\flat} v$ )
```

lemma *convex-le-refl* [simp]: $t \leq_{\mathfrak{h}} t$

unfolding *convex-le-def* **by** (*fast intro: upper-le-refl lower-le-refl*)

lemma *convex-le-trans*: $[[t \leq_{\mathfrak{h}} u; u \leq_{\mathfrak{h}} v]] \Longrightarrow t \leq_{\mathfrak{h}} v$

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* $\leq_{\mathfrak{h}} t$

unfolding *convex-le-def Rep-PDUnit* **by** *simp*

lemma *PDUnit-convex-mono*: $x \sqsubseteq y \Longrightarrow \text{PDUnit } x \leq_{\mathfrak{h}} \text{PDUnit } y$

unfolding *convex-le-def* **by** (*fast intro: PDUnit-upper-mono PDUnit-lower-mono*)

lemma *PDPlus-convex-mono*: $[[s \leq_{\mathfrak{h}} t; u \leq_{\mathfrak{h}} v]] \Longrightarrow \text{PDPlus } s \ u \leq_{\mathfrak{h}} \text{PDPlus } t \ v$

unfolding *convex-le-def* **by** (*fast intro: PDPlus-upper-mono PDPlus-lower-mono*)

lemma *convex-le-PDUnit-PDUnit-iff* [simp]:

```
(PDUnit a  $\leq_{\mathfrak{h}}$  PDUnit b) = (a  $\sqsubseteq$  b)
```

unfolding *convex-le-def upper-le-def lower-le-def Rep-PDUnit* **by** *fast*

lemma *convex-le-PDUnit-lemma1*:

```
(PDUnit a  $\leq_{\mathfrak{h}}$  t) = ( $\forall b \in \text{Rep-pd-basis } t. a \sqsubseteq b$ )
```

unfolding *convex-le-def upper-le-def lower-le-def Rep-PDUnit*

using *Rep-pd-basis-nonempty* [of *t*, *folded ex-in-conv*] **by** *fast*

lemma *convex-le-PDUnit-PDPlus-iff* [simp]:

```
(PDUnit a  $\leq_{\mathfrak{h}}$  PDPlus t u) = (PDUnit a  $\leq_{\mathfrak{h}}$  t  $\wedge$  PDUnit a  $\leq_{\mathfrak{h}}$  u)
```

unfolding *convex-le-PDUnit-lemma1 Rep-PDPlus* **by** *fast*

lemma *convex-le-PDUnit-lemma2*:

```
(t  $\leq_{\mathfrak{h}}$  PDUnit b) = ( $\forall a \in \text{Rep-pd-basis } t. a \sqsubseteq b$ )
```

unfolding *convex-le-def upper-le-def lower-le-def Rep-PDUnit*

using *Rep-pd-basis-nonempty* [of *t*, *folded ex-in-conv*] **by** *fast*

lemma *convex-le-PDPlus-PDUnit-iff* [simp]:

```
(PDPlus t u  $\leq_{\mathfrak{h}}$  PDUnit a) = (t  $\leq_{\mathfrak{h}}$  PDUnit a  $\wedge$  u  $\leq_{\mathfrak{h}}$  PDUnit a)
```

unfolding *convex-le-PDUnit-lemma2 Rep-PDPlus by fast*

lemma *convex-le-PDPlus-lemma:*

assumes $z: PDPlus\ t\ u \leq_{\mathfrak{h}} z$

shows $\exists v\ w. z = PDPlus\ v\ w \wedge t \leq_{\mathfrak{h}} v \wedge u \leq_{\mathfrak{h}} w$

proof (*intro exI conjI*)

let $?A = \{b \in Rep\text{-}pd\text{-}basis\ z. \exists a \in Rep\text{-}pd\text{-}basis\ t. a \sqsubseteq b\}$

let $?B = \{b \in Rep\text{-}pd\text{-}basis\ z. \exists a \in Rep\text{-}pd\text{-}basis\ u. a \sqsubseteq b\}$

let $?v = Abs\text{-}pd\text{-}basis\ ?A$

let $?w = Abs\text{-}pd\text{-}basis\ ?B$

have *Rep-v: Rep-pd-basis ?v = ?A*

apply (*rule Abs-pd-basis-inverse*)

apply (*rule Rep-pd-basis-nonempty [of t, folded ex-in-conv, THEN exE]*)

apply (*cut-tac z, simp only: convex-le-def lower-le-def, clarify*)

apply (*drule-tac x=x in bspec, simp add: Rep-PDPlus, erule bexE*)

apply (*simp add: pd-basis-def*)

apply *fast*

done

have *Rep-w: Rep-pd-basis ?w = ?B*

apply (*rule Abs-pd-basis-inverse*)

apply (*rule Rep-pd-basis-nonempty [of u, folded ex-in-conv, THEN exE]*)

apply (*cut-tac z, simp only: convex-le-def lower-le-def, clarify*)

apply (*drule-tac x=x in bspec, simp add: Rep-PDPlus, erule bexE*)

apply (*simp add: pd-basis-def*)

apply *fast*

done

show $z = PDPlus\ ?v\ ?w$

apply (*insert z*)

apply (*simp add: convex-le-def, erule conjE*)

apply (*simp add: Rep-pd-basis-inject [symmetric] Rep-PDPlus*)

apply (*simp add: Rep-v Rep-w*)

apply (*rule equalityI*)

apply (*rule subsetI*)

apply (*simp only: upper-le-def*)

apply (*drule (1) bspec, erule bexE*)

apply (*simp add: Rep-PDPlus*)

apply *fast*

apply *fast*

done

show $t \leq_{\mathfrak{h}} ?v\ u \leq_{\mathfrak{h}} ?w$

apply (*insert z*)

apply (*simp-all add: convex-le-def upper-le-def lower-le-def Rep-PDPlus Rep-v*

Rep-w)

apply *fast+*

done

qed

lemma *convex-le-induct [induct set: convex-le]:*

assumes $le: t \leq_{\mathfrak{h}} u$

```

assumes 2:  $\bigwedge t u v. \llbracket P t u; P u v \rrbracket \implies P t v$ 
assumes 3:  $\bigwedge a b. a \sqsubseteq b \implies P (PDUnit a) (PDUnit b)$ 
assumes 4:  $\bigwedge t u v w. \llbracket P t v; P u w \rrbracket \implies P (PDPlus t u) (PDPlus v w)$ 
shows  $P t u$ 
using le apply (induct t arbitrary: u rule: pd-basis-induct)
apply (erule rev-mp)
apply (induct-tac u rule: pd-basis-induct1)
apply (simp add: 3)
apply (simp, clarify, rename-tac a b t)
apply (subgoal-tac P (PDPlus (PDUnit a) (PDUnit a)) (PDPlus (PDUnit b) t))
apply (simp add: PDPlus-absorb)
apply (erule (1) 4 [OF 3])
apply (drule convex-le-PDPlus-lemma, clarify)
apply (simp add: 4)
done

```

32.2 Type definition

```

typedef 'a convex-pd ((-' $\sqsubseteq$ ')) =
  {S::'a pd-basis set. convex-le.ideal S}
by (rule convex-le.ex-ideal)

```

```

instantiation convex-pd :: (bifinite) below
begin

```

```

definition
   $x \sqsubseteq y \longleftrightarrow \text{Rep-convex-pd } x \subseteq \text{Rep-convex-pd } y$ 

```

```

instance ..
end

```

```

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

```

```

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

```

```

definition
  convex-principal :: 'a pd-basis  $\Rightarrow$  'a convex-pd where
  convex-principal t = Abs-convex-pd {u. u  $\leq$  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*) \sqsubseteq *ys*
by (*induct ys rule: convex-pd.principal-induct, simp, simp*)

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

lemma *inst-convex-pd-pcpo*: $\perp = \text{convex-principal } (PDUnit \text{ compact-bot})$
by (*rule convex-pd-minimal [THEN bottomI, symmetric]*)

32.3 Monadic unit and plus

definition

convex-unit :: 'a \rightarrow 'a *convex-pd* **where**
convex-unit = *compact-basis.extension* ($\lambda a.$ *convex-principal* (*PDUnit a*))

definition

convex-plus :: 'a *convex-pd* \rightarrow 'a *convex-pd* \rightarrow 'a *convex-pd* **where**
convex-plus = *convex-pd.extension* ($\lambda t.$ *convex-pd.extension* ($\lambda u.$
convex-principal (*PDPlus t u*)))

abbreviation

convex-add :: 'a *convex-pd* \Rightarrow 'a *convex-pd* \Rightarrow 'a *convex-pd*
(infixl $\cup\!\!\!\cup$ **65)** **where**
 $xs \cup\!\!\!\cup ys == \text{convex-plus} \cdot xs \cdot ys$

syntax

-convex-pd :: *args* \Rightarrow *logic* ($\{-\}\!\!\!\cup$)

translations

$\{x, xs\}\!\!\!\cup == \{x\}\!\!\!\cup \cup\!\!\!\cup \{xs\}\!\!\!\cup$
 $\{x\}\!\!\!\cup == \text{CONST } \text{convex-unit} \cdot x$

lemma *convex-unit-Rep-compact-basis* [*simp*]:

$\{\text{Rep-compact-basis } a\}\!\!\!\cup = \text{convex-principal } (PDUnit a)$

unfolding *convex-unit-def*

by (*simp add: compact-basis.extension-principal PDUnit-convex-mono*)

lemma *convex-plus-principal* [*simp*]:

$\text{convex-principal } t \cup\!\!\!\cup \text{convex-principal } u = \text{convex-principal } (PDPlus t u)$

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 $(xs \cup\!\!\!\cup ys) \cup\!\!\!\cup zs = xs \cup\!\!\!\cup (ys \cup\!\!\!\cup zs)$

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  $xs \sqcup yz = ys \sqcup xs$ 
  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  $xs \sqcup xs = xs$ 
  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]:
   $\{x\} \sqsubseteq ys \sqcup zs \iff \{x\} \sqsubseteq ys \wedge \{x\} \sqsubseteq zs$ 
  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]:
   $xs \sqcup ys \sqsubseteq \{z\} \iff xs \sqsubseteq \{z\} \wedge ys \sqsubseteq \{z\}$ 
  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]:  $\{x\} \sqsubseteq \{y\} \iff x \sqsubseteq y$ 
  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*]: $\{x\}\Downarrow = \{y\}\Downarrow \iff x = y$
unfolding *po-eq-conv* **by** *simp*

lemma *convex-unit-strict* [*simp*]: $\{\perp\}\Downarrow = \perp$
using *convex-unit-Rep-compact-basis* [*of compact-bot*]
by (*simp add: inst-convex-pd-pcpo*)

lemma *convex-unit-bottom-iff* [*simp*]: $\{x\}\Downarrow = \perp \iff x = \perp$
unfolding *convex-unit-strict* [*symmetric*] **by** (*rule convex-unit-eq-iff*)

lemma *compact-convex-unit*: $\text{compact } x \implies \text{compact } \{x\}\Downarrow$
by (*auto dest!: compact-basis.compact-imp-principal*)

lemma *compact-convex-unit-iff* [*simp*]: $\text{compact } \{x\}\Downarrow \iff \text{compact } x$
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*]:
 $\llbracket \text{compact } xs; \text{compact } ys \rrbracket \implies \text{compact } (xs \cup\Downarrow ys)$
by (*auto dest!: convex-pd.compact-imp-principal*)

32.4 Induction rules

lemma *convex-pd-induct1*:
assumes *P: adm P*
assumes *unit*: $\bigwedge x. P \{x\}\Downarrow$
assumes *insert*: $\bigwedge x ys. \llbracket P \{x\}\Downarrow; P ys \rrbracket \implies P (\{x\}\Downarrow \cup\Downarrow ys)$
shows $P (xs::'a \text{ convex-pd})$
apply (*induct xs rule: convex-pd.principal-induct, rule P*)
apply (*induct-tac a rule: pd-basis-induct1*)
apply (*simp only: convex-unit-Rep-compact-basis* [*symmetric*])
apply (*rule unit*)
apply (*simp only: convex-unit-Rep-compact-basis* [*symmetric*]
convex-plus-principal [*symmetric*])
apply (*erule insert* [*OF unit*])
done

lemma *convex-pd-induct*
[*case-names adm convex-unit convex-plus, induct type: convex-pd*]:
assumes *P: adm P*
assumes *unit*: $\bigwedge x. P \{x\}\Downarrow$
assumes *plus*: $\bigwedge xs ys. \llbracket P xs; P ys \rrbracket \implies P (xs \cup\Downarrow ys)$
shows $P (xs::'a \text{ convex-pd})$
apply (*induct xs rule: convex-pd.principal-induct, rule P*)
apply (*induct-tac a rule: pd-basis-induct*)
apply (*simp only: convex-unit-Rep-compact-basis* [*symmetric*] *unit*)
apply (*simp only: convex-plus-principal* [*symmetric*] *plus*)

done

32.5 Monadic bind

definition

convex-bind-basis ::
 'a pd-basis \Rightarrow ('a \rightarrow 'b convex-pd) \rightarrow 'b convex-pd **where**
convex-bind-basis = fold-pd
 ($\lambda a. \Lambda f. f \cdot (\text{Rep-compact-basis } a)$)
 ($\lambda x y. \Lambda f. x \cdot f \cup_{\text{h}} y \cdot f$)

lemma *ACI-convex-bind*:

semilattice ($\lambda x y. \Lambda f. x \cdot f \cup_{\text{h}} y \cdot f$)

apply *unfold-locales*

apply (*simp add: convex-plus-assoc*)

apply (*simp add: convex-plus-commute*)

apply (*simp add: eta-cfun*)

done

lemma *convex-bind-basis-simps* [*simp*]:

convex-bind-basis (PDUnit a) =
 ($\Lambda f. f \cdot (\text{Rep-compact-basis } a)$)
convex-bind-basis (PDPlus t u) =
 ($\Lambda f. \text{convex-bind-basis } t \cdot f \cup_{\text{h}} \text{convex-bind-basis } u \cdot f$)

unfolding *convex-bind-basis-def*

apply –

apply (*rule fold-pd-PDUnit [OF ACI-convex-bind]*)

apply (*rule fold-pd-PDPlus [OF ACI-convex-bind]*)

done

lemma *convex-bind-basis-mono*:

$t \leq_{\text{h}} u \implies \text{convex-bind-basis } t \sqsubseteq \text{convex-bind-basis } u$

apply (*erule convex-le-induct*)

apply (*erule (1) below-trans*)

apply (*simp add: monofun-LAM monofun-cfun*)

apply (*simp add: monofun-LAM monofun-cfun*)

done

definition

convex-bind :: 'a convex-pd \rightarrow ('a \rightarrow 'b convex-pd) \rightarrow 'b convex-pd **where**
convex-bind = *convex-pd.extension convex-bind-basis*

syntax

-convex-bind :: [*logic, logic, logic*] \Rightarrow *logic*

(($\exists \cup_{\text{h}} \in \cdot / \cdot$) [0, 0, 10] 10)

translations

$\cup_{\text{h}} x \in xs. e == \text{CONST } \text{convex-bind} \cdot xs \cdot (\Lambda x. e)$

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]:
 $convex-bind.\{x\}\dagger.f = f.x$
by (induct x rule: *compact-basis.principal-induct*, *simp*, *simp*)

lemma *convex-bind-plus* [simp]:
 $convex-bind.(xs\ \cup\ \dagger\ ys).f = convex-bind.xs.f\ \cup\ \dagger\ convex-bind.ys.f$
by (induct xs rule: *convex-pd.principal-induct*, *simp*,
induct ys rule: *convex-pd.principal-induct*, *simp*, *simp*)

lemma *convex-bind-strict* [simp]: $convex-bind.\perp.f = f.\perp$
unfolding *convex-unit-strict* [symmetric] **by** (rule *convex-bind-unit*)

lemma *convex-bind-bind*:
 $convex-bind.(convex-bind.xs.f).g =$
 $convex-bind.xs.(\Lambda\ x.\ convex-bind.(f.x).g)$
by (induct xs , *simp-all*)

32.6 Map

definition
 $convex-map :: ('a \rightarrow 'b) \rightarrow 'a\ convex-pd \rightarrow 'b\ convex-pd$ **where**
 $convex-map = (\Lambda\ f\ xs.\ convex-bind.xs.(\Lambda\ x.\ \{f.x\}\dagger))$

lemma *convex-map-unit* [simp]:
 $convex-map.f.\{x\}\dagger = \{f.x\}\dagger$
unfolding *convex-map-def* **by** *simp*

lemma *convex-map-plus* [simp]:
 $convex-map.f.(xs\ \cup\ \dagger\ ys) = convex-map.f.xs\ \cup\ \dagger\ convex-map.f.ys$
unfolding *convex-map-def* **by** *simp*

lemma *convex-map-bottom* [simp]: $convex-map.f.\perp = \{f.\perp\}\dagger$
unfolding *convex-map-def* **by** *simp*

lemma *convex-map-ident*: $convex-map.(\Lambda\ x.\ x).xs = xs$
by (induct xs rule: *convex-pd-induct*, *simp-all*)

lemma *convex-map-ID*: $convex-map.ID = ID$
by (*simp add: cfun-eq-iff ID-def convex-map-ident*)

lemma *convex-map-map*:
 $convex-map.f.(convex-map.g.xs) = convex-map.(\Lambda\ x.\ f.(g.x)).xs$

by (*induct xs rule: convex-pd-induct, simp-all*)

lemma *convex-bind-map:*

$convex-bind.(convex-map.f \cdot xs) \cdot g = convex-bind.xs(\Lambda x. g(f \cdot x))$

by (*simp add: convex-map-def convex-bind-bind*)

lemma *convex-map-bind:*

$convex-map.f.(convex-bind.xs \cdot g) = convex-bind.xs(\Lambda x. convex-map.f.(g \cdot x))$

by (*simp add: convex-map-def convex-bind-bind*)

lemma *ep-pair-convex-map:* $ep\text{-}pair\ e\ p \implies ep\text{-}pair\ (convex\text{-}map.e)\ (convex\text{-}map.p)$

apply *standard*

apply (*induct-tac x rule: convex-pd-induct, simp-all add: ep-pair.e-inverse*)

apply (*induct-tac y rule: convex-pd-induct*)

apply (*simp-all add: ep-pair.e-p-below monofun-cfun*)

done

lemma *deflation-convex-map:* $deflation\ d \implies deflation\ (convex\text{-}map.d)$

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 (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 (convex-principal ' Rep-pd-basis - ' (Pow (Rep-compact-basis - ' range (λx. d.x)))) by simp*

hence *finite (range (λxs. 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

```

32.7 Convex powerdomain is bifinite

```

lemma approx-chain-convex-map:
  assumes approx-chain a
  shows approx-chain ( $\lambda i. \text{convex-map}(a i)$ )
  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  $\exists (a::\text{nat} \Rightarrow 'a \text{ convex-pd} \rightarrow 'a \text{ convex-pd}). \text{approx-chain } a$ 
  using bifinite [where 'a='a]
  by (fast intro!: approx-chain-convex-map)

```

```

qed

```

32.8 Join

```

definition

```

```

  convex-join :: 'a convex-pd convex-pd  $\rightarrow$  'a convex-pd where
  convex-join = ( $\Lambda xss. \text{convex-bind}.xss.(\Lambda xs. xs)$ )

```

```

lemma convex-join-unit [simp]:

```

```

  convex-join.{xs}⊔ = xs

```

```

unfolding convex-join-def by simp

```

```

lemma convex-join-plus [simp]:

```

```

  convex-join.(xss  $\cup$ ⊔ yss) = convex-join.xss  $\cup$ ⊔ convex-join.yss

```

```

unfolding convex-join-def by simp

```

```

lemma convex-join-bottom [simp]: convex-join. $\perp$  =  $\perp$ 

```

```

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*)

32.9 Conversions to other powerdomains

Convex to upper

lemma *convex-le-imp-upper-le*: $t \leq_{\natural} u \implies t \leq_{\sharp} u$
unfolding *convex-le-def* **by** *simp*

definition

convex-to-upper :: 'a convex-pd \rightarrow '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*]:
 $convex-to-upper.\{x\}_{\natural} = \{x\}_{\sharp}$
by (*induct x rule: compact-basis.principal-induct, simp, simp*)

lemma *convex-to-upper-plus* [*simp*]:
 $convex-to-upper.(xs \cup_{\natural} ys) = convex-to-upper.xs \cup_{\sharp} convex-to-upper.ys$
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 cfcamp-LAM*)

lemma *convex-to-upper-join* [*simp*]:
 $convex-to-upper \cdot (convex-join \cdot xss) =$
 $upper-bind \cdot (convex-to-upper \cdot xss) \cdot convex-to-upper$
by (*simp add: convex-join-def upper-join-def cfcomp-LAM eta-cfun*)

Convex to lower

lemma *convex-le-imp-lower-le*: $t \leq_{\natural} u \implies t \leq_{\flat} u$
unfolding *convex-le-def* **by** *simp*

definition

convex-to-lower :: 'a convex-pd \rightarrow 'a lower-pd **where**
convex-to-lower = *convex-pd.extension lower-principal*

lemma *convex-to-lower-principal* [*simp*]:
 $convex-to-lower \cdot (convex-principal \ t) = lower-principal \ t$
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*]:
 $convex-to-lower \cdot \{x\}_{\natural} = \{x\}_{\flat}$
by (*induct x rule: compact-basis.principal-induct, simp, simp*)

lemma *convex-to-lower-plus* [*simp*]:
 $convex-to-lower \cdot (xs \cup_{\natural} ys) = convex-to-lower \cdot xs \cup_{\flat} convex-to-lower \cdot ys$
by (*induct xs rule: convex-pd.principal-induct, simp,*
induct ys rule: convex-pd.principal-induct, simp, simp)

lemma *convex-to-lower-bind* [*simp*]:
 $convex-to-lower \cdot (convex-bind \cdot xs \cdot f) =$
 $lower-bind \cdot (convex-to-lower \cdot xs) \cdot (convex-to-lower \ o \ f)$
by (*induct xs rule: convex-pd-induct, simp, simp, simp*)

lemma *convex-to-lower-map* [*simp*]:
 $convex-to-lower \cdot (convex-map \cdot f \cdot xs) = lower-map \cdot f \cdot (convex-to-lower \cdot xs)$
by (*simp add: convex-map-def lower-map-def cfcomp-LAM*)

lemma *convex-to-lower-join* [*simp*]:
 $convex-to-lower \cdot (convex-join \cdot xss) =$
 $lower-bind \cdot (convex-to-lower \cdot xss) \cdot convex-to-lower$
by (*simp add: convex-join-def lower-join-def cfcomp-LAM eta-cfun*)

Ordering property

lemma *convex-pd-below-iff*:
 $(xs \sqsubseteq ys) =$
 $(convex-to-upper \cdot xs \sqsubseteq convex-to-upper \cdot ys \wedge$
 $convex-to-lower \cdot xs \sqsubseteq convex-to-lower \cdot ys)$

```

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  $\cup$  ys and ys=zs  $\cup$  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

```

33 Powerdomains

```

theory Powerdomains
imports ConvexPD Domain
begin

```

33.1 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*)

33.2 Deflation combinators

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

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

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

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

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

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

33.3 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

$\text{liftdefl } (t::'a \text{ upper-pd itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \text{ upper-pd})$

instance proof

show $\text{ep-pair emb } (prj :: \text{udom} \rightarrow 'a \text{ upper-pd})$

unfolding $\text{emb-upper-pd-def } prj\text{-upper-pd-def}$

by $(\text{intro } \text{ep-pair-comp } \text{ep-pair-upper } \text{ep-pair-upper-map } \text{ep-pair-emb-prj})$

next

show $\text{cast} \cdot \text{DEFL}('a \text{ upper-pd}) = \text{emb } oo (prj :: \text{udom} \rightarrow 'a \text{ upper-pd})$

unfolding $\text{emb-upper-pd-def } prj\text{-upper-pd-def } \text{defl-upper-pd-def } \text{cast-upper-defl}$

by $(\text{simp add: } \text{cast-DEFL } oo\text{-def } \text{cfun-eq-iff } \text{upper-map-map})$

qed $(\text{fact } \text{liftemb-upper-pd-def } \text{liftprj-upper-pd-def } \text{liftdefl-upper-pd-def})+$

end

instantiation $\text{lower-pd} :: (\text{domain}) \text{ domain}$

begin

definition

$\text{emb} = \text{lower-emb } oo \text{ lower-map} \cdot \text{emb}$

definition

$\text{prj} = \text{lower-map} \cdot \text{prj } oo \text{ lower-prj}$

definition

$\text{defl } (t::'a \text{ lower-pd itself}) = \text{lower-defl} \cdot \text{DEFL}('a)$

definition

$(\text{liftemb} :: 'a \text{ lower-pd } u \rightarrow \text{udom } u) = u\text{-map} \cdot \text{emb}$

definition

$(\text{liftprj} :: \text{udom } u \rightarrow 'a \text{ lower-pd } u) = u\text{-map} \cdot \text{prj}$

definition

$\text{liftdefl } (t::'a \text{ lower-pd itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \text{ lower-pd})$

instance proof

show $\text{ep-pair emb } (prj :: \text{udom} \rightarrow 'a \text{ lower-pd})$

unfolding $\text{emb-lower-pd-def } prj\text{-lower-pd-def}$

by $(\text{intro } \text{ep-pair-comp } \text{ep-pair-lower } \text{ep-pair-lower-map } \text{ep-pair-emb-prj})$

next

show $\text{cast} \cdot \text{DEFL}('a \text{ lower-pd}) = \text{emb } oo (prj :: \text{udom} \rightarrow 'a \text{ lower-pd})$

unfolding $\text{emb-lower-pd-def } prj\text{-lower-pd-def } \text{defl-lower-pd-def } \text{cast-lower-defl}$

by $(\text{simp add: } \text{cast-DEFL } oo\text{-def } \text{cfun-eq-iff } \text{lower-map-map})$

qed $(\text{fact } \text{liftemb-lower-pd-def } \text{liftprj-lower-pd-def } \text{liftdefl-lower-pd-def})+$

end

instantiation $\text{convex-pd} :: (\text{domain}) \text{ domain}$

begin

definition

$$\text{emb} = \text{convex-emb} \text{ oo } \text{convex-map} \cdot \text{emb}$$
definition

$$\text{prj} = \text{convex-map} \cdot \text{prj} \text{ oo } \text{convex-prj}$$
definition

$$\text{defl} (t :: 'a \text{ convex-pd } \text{itself}) = \text{convex-defl} \cdot \text{DEFL}('a)$$
definition

$$(\text{liftemb} :: 'a \text{ convex-pd } u \rightarrow \text{udom } u) = u \cdot \text{map} \cdot \text{emb}$$
definition

$$(\text{liftprj} :: \text{udom } u \rightarrow 'a \text{ convex-pd } u) = u \cdot \text{map} \cdot \text{prj}$$
definition

$$\text{liftdefl} (t :: 'a \text{ convex-pd } \text{itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \text{ convex-pd})$$
instance proof

$$\text{show } \text{ep-pair } \text{emb} (\text{prj} :: \text{udom} \rightarrow 'a \text{ convex-pd})$$

$$\text{unfolding } \text{emb-convex-pd-def } \text{prj-convex-pd-def}$$

$$\text{by } (\text{intro } \text{ep-pair-comp } \text{ep-pair-convex } \text{ep-pair-convex-map } \text{ep-pair-emb-prj})$$
next

$$\text{show } \text{cast} \cdot \text{DEFL}('a \text{ convex-pd}) = \text{emb} \text{ oo } (\text{prj} :: \text{udom} \rightarrow 'a \text{ convex-pd})$$

$$\text{unfolding } \text{emb-convex-pd-def } \text{prj-convex-pd-def } \text{defl-convex-pd-def } \text{cast-convex-defl}$$

$$\text{by } (\text{simp } \text{add: } \text{cast-DEFL } \text{oo-def } \text{cfun-eq-iff } \text{convex-map-map})$$

$$\text{qed } (\text{fact } \text{liftemb-convex-pd-def } \text{liftprj-convex-pd-def } \text{liftdefl-convex-pd-def}) +$$
end

$$\text{lemma } \text{DEFL-upper: } \text{DEFL}('a :: \text{domain } \text{upper-pd}) = \text{upper-defl} \cdot \text{DEFL}('a)$$

$$\text{by } (\text{rule } \text{defl-upper-pd-def})$$

$$\text{lemma } \text{DEFL-lower: } \text{DEFL}('a :: \text{domain } \text{lower-pd}) = \text{lower-defl} \cdot \text{DEFL}('a)$$

$$\text{by } (\text{rule } \text{defl-lower-pd-def})$$

$$\text{lemma } \text{DEFL-convex: } \text{DEFL}('a :: \text{domain } \text{convex-pd}) = \text{convex-defl} \cdot \text{DEFL}('a)$$

$$\text{by } (\text{rule } \text{defl-convex-pd-def})$$

33.4 Isomorphic deflations

$$\text{lemma } \text{isodefl-upper:}$$

$$\text{isodefl } d \ t \implies \text{isodefl } (\text{upper-map} \cdot d) (\text{upper-defl} \cdot t)$$

$$\text{apply } (\text{rule } \text{isodefl})$$

$$\text{apply } (\text{simp } \text{add: } \text{cast-upper-defl } \text{cast-isodefl})$$

$$\text{apply } (\text{simp } \text{add: } \text{emb-upper-pd-def } \text{prj-upper-pd-def})$$

$$\text{apply } (\text{simp } \text{add: } \text{upper-map-map})$$

$$\text{done}$$


```

lemma isodeft-lower:
  isodeft d t  $\implies$  isodeft (lower-map.d) (lower-defl.t)
apply (rule isodeftI)
apply (simp add: cast-lower-defl cast-isodeft)
apply (simp add: emb-lower-pd-def prj-lower-pd-def)
apply (simp add: lower-map-map)
done

```

```

lemma isodeft-convex:
  isodeft d t  $\implies$  isodeft (convex-map.d) (convex-defl.t)
apply (rule isodeftI)
apply (simp add: cast-convex-defl cast-isodeft)
apply (simp add: emb-convex-pd-def prj-convex-pd-def)
apply (simp add: convex-map-map)
done

```

33.5 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-isodeft] = isodeft-upper isodeft-lower isodeft-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