

Isabelle/HOLCF — Higher-Order Logic of Computable Functions

December 17, 2025

Contents

1	Partial orders	3
1.1	Type class for partial orders	3
1.2	Upper bounds	4
1.3	Least upper bounds	5
1.4	Countable chains	6
1.5	Finite chains	7
2	Classes cpo and pcpo	8
2.1	Complete partial orders	8
2.2	Pointed cpos	10
2.3	Chain-finite and flat cpos	10
2.4	Discrete cpos	11
3	Continuity and monotonicity	11
3.1	Definitions	11
3.2	Equivalence of alternate definition	12
3.3	Collection of continuity rules	13
3.4	Continuity of basic functions	13
3.5	Finite chains and flat pcpes	13
4	Admissibility and compactness	14
4.1	Definitions	14
4.2	Admissibility on chain-finite types	15
4.3	Admissibility of special formulae and propagation	15
4.4	Compactness	16
5	Class instances for the full function space	17
5.1	Full function space is a partial order	17
5.2	Full function space is chain complete	18
5.3	Full function space is pointed	18
5.4	Propagation of monotonicity and continuity	19

6	The cpo of cartesian products	19
6.1	Unit type is a pcpo	19
6.2	Product type is a partial order	20
6.3	Monotonicity of <i>Pair</i> , <i>fst</i> , <i>snd</i>	20
6.4	Product type is a cpo	21
6.5	Product type is pointed	21
6.6	Continuity of <i>Pair</i> , <i>fst</i> , <i>snd</i>	22
6.7	Compactness and chain-finiteness	23
7	Discrete cpo types	24
7.1	Discrete cpo class instance	24
7.2	<i>undiscr</i>	24
8	Subtypes of pcpos	24
8.1	Proving a subtype is a partial order	24
8.2	Proving a subtype is finite	25
8.3	Proving a subtype is chain-finite	25
8.4	Proving a subtype is complete	25
8.4.1	Continuity of <i>Rep</i> and <i>Abs</i>	26
8.5	Proving subtype elements are compact	26
8.6	Proving a subtype is pointed	27
8.6.1	Strictness of <i>Rep</i> and <i>Abs</i>	27
8.7	Proving a subtype is flat	28
8.8	HOLCF type definition package	28
9	The type of continuous functions	28
9.1	Definition of continuous function type	28
9.2	Syntax for continuous lambda abstraction	29
9.3	Continuous function space is pointed	29
9.4	Basic properties of continuous functions	30
9.4.1	Beta-reduction simproc	30
9.5	Continuity of application	31
9.6	Continuity simplification procedure	32
9.7	Miscellaneous	33
9.8	Continuous injection-retraction pairs	33
9.9	Identity and composition	34
9.10	Strictified functions	35
9.11	Continuity of let-bindings	35
10	Continuous deflations and ep-pairs	36
10.1	Continuous deflations	36
10.2	Deflations with finite range	37
10.3	Continuous embedding-projection pairs	38
10.4	Uniqueness of ep-pairs	39

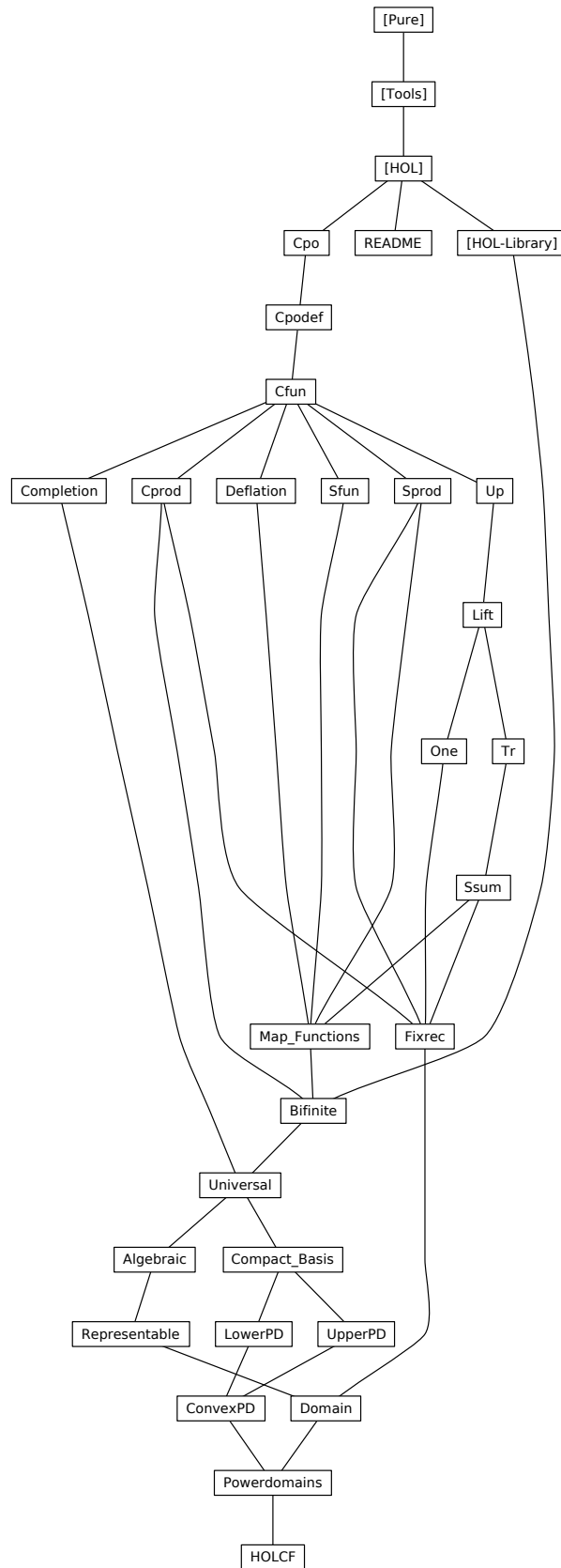
10.5 Composing ep-pairs	40
11 The type of strict products	40
11.1 Definition of strict product type	40
11.2 Definitions of constants	41
11.3 Case analysis	41
11.4 Properties of <i>spair</i>	42
11.5 Properties of <i>sfst</i> and <i>ssnd</i>	43
11.6 Compactness	43
11.7 Properties of <i>ssplit</i>	44
11.8 Strict product preserves flatness	44
12 The type of lifted values	44
12.1 Definition of new type for lifting	44
12.2 Ordering on lifted cpo	44
12.3 Lifted cpo is a partial order	45
12.4 Lifted cpo is a cpo	45
12.5 Lifted cpo is pointed	45
12.6 Continuity of <i>Iup</i> and <i>Ifup</i>	45
12.7 Continuous versions of constants	46
13 Lifting types of class type to flat pcpo's	47
13.1 Lift as a datatype	48
13.2 Lift is flat	48
13.3 Continuity of <i>case-lift</i>	48
13.4 Further operations	48
14 The type of lifted booleans	49
14.1 Type definition and constructors	49
14.2 Case analysis	50
14.3 Boolean connectives	51
14.4 Rewriting of HOLCF operations to HOL functions	52
14.5 Compactness	52
15 The type of strict sums	53
15.1 Definition of strict sum type	53
15.2 Definitions of constructors	53
15.3 Properties of <i>sinl</i> and <i>sinr</i>	54
15.4 Case analysis	55
15.5 Case analysis combinator	55
15.6 Strict sum preserves flatness	56
16 The Strict Function Type	56

17 Map functions for various types	57
17.1 Map operator for continuous function space	57
17.2 Map operator for product type	58
17.3 Map function for lifted cpo	59
17.4 Map function for strict products	59
17.5 Map function for strict sums	60
17.6 Map operator for strict function space	61
18 The cpo of cartesian products	62
18.1 Continuous case function for unit type	62
18.2 Continuous version of split function	62
18.3 Convert all lemmas to the continuous versions	62
19 Profinite and bifinite cpos	63
19.1 Chains of finite deflations	63
19.2 Omega-profinite and bifinite domains	63
19.3 Building approx chains	64
19.4 Class instance proofs	65
20 Defining algebraic domains by ideal completion	66
20.1 Ideals over a preorder	66
20.2 Lemmas about least upper bounds	68
20.3 Locale for ideal completion	68
20.3.1 Principal ideals approximate all elements	69
20.4 Defining functions in terms of basis elements	69
21 A universal bifinite domain	70
21.1 Basis for universal domain	70
21.1.1 Basis datatype	70
21.1.2 Basis ordering	71
21.1.3 Generic take function	71
21.2 Defining the universal domain by ideal completion	72
21.3 Compact bases of domains	73
21.4 Universality of <i>udom</i>	74
21.4.1 Choosing a maximal element from a finite set	74
21.4.2 Compact basis take function	75
21.4.3 Rank of basis elements	76
21.4.4 Sequencing basis elements	77
21.4.5 Embedding and projection on basis elements	78
21.4.6 EP-pair from any bifinite domain into <i>udom</i>	80
21.5 Chain of approx functions for type <i>udom</i>	81

22 Algebraic deflations	82
22.1 Type constructor for finite deflations	82
22.2 Defining algebraic deflations by ideal completion	83
22.3 Applying algebraic deflations	84
22.4 Deflation combinators	84
23 Representable domains	85
23.1 Class of representable domains	85
23.2 Domains are bifinite	86
23.3 Universal domain ep-pairs	87
23.4 Type combinators	87
23.5 Class instance proofs	88
23.5.1 Universal domain	88
23.5.2 Lifted cpo	89
23.5.3 Strict function space	89
23.5.4 Continuous function space	90
23.5.5 Strict product	91
23.5.6 Cartesian product	91
23.5.7 Unit type	92
23.5.8 Discrete cpo	93
23.5.9 Strict sum	93
23.5.10 Lifted HOL type	94
24 The unit domain	95
25 Fixed point operator and admissibility	96
25.1 Iteration	96
25.2 Least fixed point operator	97
25.3 Fixed point induction	98
25.4 Fixed-points on product types	99
26 Package for defining recursive functions in HOLCF	99
26.1 Pattern-match monad	99
26.1.1 Run operator	100
26.1.2 Monad plus operator	100
26.2 Match functions for built-in types	101
26.3 Mutual recursion	102
26.4 Initializing the fixrec package	103
27 Domain package	103
27.1 Continuous isomorphisms	103
27.2 Proofs about take functions	105
27.3 Finiteness	105
27.4 Proofs about constructor functions	107

27.5	ML setup	108
27.6	Representations of types	108
27.7	Deflations as sets	109
27.8	Proving a subtype is representable	109
27.9	Isomorphic deflations	110
27.10	Setting up the domain package	112
28	A compact basis for powerdomains	113
28.1	A compact basis for powerdomains	113
28.2	Unit and plus constructors	113
28.3	Fold operator	114
29	Upper powerdomain	115
29.1	Basis preorder	115
29.2	Type definition	116
29.3	Monadic unit and plus	116
29.4	Induction rules	118
29.5	Monadic bind	119
29.6	Map	120
29.7	Upper powerdomain is bifinite	121
29.8	Join	121
30	Lower powerdomain	122
30.1	Basis preorder	122
30.2	Type definition	123
30.3	Monadic unit and plus	124
30.4	Induction rules	126
30.5	Monadic bind	126
30.6	Map	127
30.7	Lower powerdomain is bifinite	128
30.8	Join	128
31	Convex powerdomain	129
31.1	Basis preorder	129
31.2	Type definition	130
31.3	Monadic unit and plus	131
31.4	Induction rules	132
31.5	Monadic bind	133
31.6	Map	134
31.7	Convex powerdomain is bifinite	135
31.8	Join	135
31.9	Conversions to other powerdomains	136

32 Powerdomains	138
32.1 Universal domain embeddings	138
32.2 Deflation combinators	138
32.3 Domain class instances	139
32.4 Isomorphic deflations	140
32.5 Domain package setup for powerdomains	141




```

theory Cpo
  imports Main
begin

```

1 Partial orders

```

declare [[typedef-overloaded]]

```

1.1 Type class for partial orders

```

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

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

  notation
    below (infix < $\sqsubseteq$ > 50)

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

  notation (ASCII)
    not-below (infix < $\sim$ <<> 50)

  lemma below-eq-trans: a  $\sqsubseteq$  b  $\Longrightarrow$  b = c  $\Longrightarrow$  a  $\sqsubseteq$  c
    <proof>

  lemma eq-below-trans: a = b  $\Longrightarrow$  b  $\sqsubseteq$  c  $\Longrightarrow$  a  $\sqsubseteq$  c
    <proof>

end

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

  lemma eq-imp-below: x = y  $\Longrightarrow$  x  $\sqsubseteq$  y
    <proof>

  lemma box-below: a  $\sqsubseteq$  b  $\Longrightarrow$  c  $\sqsubseteq$  a  $\Longrightarrow$  b  $\sqsubseteq$  d  $\Longrightarrow$  c  $\sqsubseteq$  d
    <proof>

  lemma po-eq-conv: x = y  $\longleftrightarrow$  x  $\sqsubseteq$  y  $\wedge$  y  $\sqsubseteq$  x

```

$\langle proof \rangle$

lemma *rev-below-trans*: $y \sqsubseteq z \implies x \sqsubseteq y \implies x \sqsubseteq z$
 $\langle proof \rangle$

lemma *not-below2not-eq*: $x \not\sqsubseteq y \implies x \neq y$
 $\langle proof \rangle$

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* :: $'a \text{ set} \Rightarrow 'a \Rightarrow \text{bool}$ (**infix** $\langle <| \rangle$ 55)
where $S <| x \longleftrightarrow (\forall y \in S. y \sqsubseteq x)$

lemma *is-ubI*: $(\bigwedge x. x \in S \implies x \sqsubseteq u) \implies S <| u$
 $\langle proof \rangle$

lemma *is-ubD*: $\llbracket S <| u; x \in S \rrbracket \implies x \sqsubseteq u$
 $\langle proof \rangle$

lemma *ub-imageI*: $(\bigwedge x. x \in S \implies f x \sqsubseteq u) \implies (\lambda x. f x) ' S <| u$
 $\langle proof \rangle$

lemma *ub-imageD*: $\llbracket f ' S <| u; x \in S \rrbracket \implies f x \sqsubseteq u$
 $\langle proof \rangle$

lemma *ub-rangeI*: $(\bigwedge i. S i \sqsubseteq x) \implies \text{range } S <| x$
 $\langle proof \rangle$

lemma *ub-rangeD*: $\text{range } S <| x \implies S i \sqsubseteq x$
 $\langle proof \rangle$

lemma *is-ub-empty* [*simp*]: $\{\} <| u$
 $\langle proof \rangle$

lemma *is-ub-insert* [*simp*]: $(\text{insert } x A) <| y = (x \sqsubseteq y \wedge A <| y)$
 $\langle proof \rangle$

lemma *is-ub-upward*: $\llbracket S <| x; x \sqsubseteq y \rrbracket \implies S <| y$

$\langle proof \rangle$

1.3 Least upper bounds

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

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

end

syntax (*ASCII*)

-*BLub* :: [*pttrn*, 'a set, 'b] \Rightarrow 'b ($\langle \langle \text{indent}=3\ \text{notation}=\langle \text{binder } LUB \rangle \rangle LUB \text{ } :-./ \text{ } - \rangle$
 \rangle [0,0, 10] 10)

syntax

-*BLub* :: [*pttrn*, 'a set, 'b] \Rightarrow 'b ($\langle \langle \text{indent}=3\ \text{notation}=\langle \text{binder } \sqcup \rangle \rangle \sqcup \text{ } -\in-./ \text{ } - \rangle$
 \rangle [0,0, 10] 10)

syntax-consts

-*BLub* \Leftarrow *lub*

translations

LUB $x:A. t \Leftarrow CONST\ lub\ ((\lambda x. t) \text{ ' } A)$

context *po*

begin

abbreviation *Lub* (**binder** $\langle \sqcup \rangle$ 10)

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

notation (*ASCII*)

Lub (**binder** $\langle LUB \rangle$ 10)

access to some definition as inference rule

lemma *is-lubD1*: $S \ll x \Longrightarrow S < x$
 $\langle proof \rangle$

lemma *is-lubD2*: $\llbracket S \ll x; S < u \rrbracket \Longrightarrow x \sqsubseteq u$
 $\langle proof \rangle$

lemma *is-lubI*: $\llbracket S < x; \bigwedge u. S < u \rrbracket \Longrightarrow x \sqsubseteq u \Longrightarrow S \ll x$
 $\langle proof \rangle$

lemma *is-lub-below-iff*: $S \ll x \Longrightarrow x \sqsubseteq u \longleftrightarrow S < u$
 $\langle proof \rangle$

lubs are unique

lemma *is-lub-unique*: $S <<| x \implies S <<| y \implies x = y$
 $\langle \text{proof} \rangle$

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

lemma *is-lub-lub*: $M <<| x \implies M <<| \text{lub } M$
 $\langle \text{proof} \rangle$

lemma *lub-eqI*: $M <<| l \implies \text{lub } M = l$
 $\langle \text{proof} \rangle$

lemma *is-lub-singleton* [simp]: $\{x\} <<| x$
 $\langle \text{proof} \rangle$

lemma *lub-singleton* [simp]: $\text{lub } \{x\} = x$
 $\langle \text{proof} \rangle$

lemma *is-lub-bin*: $x \sqsubseteq y \implies \{x, y\} <<| y$
 $\langle \text{proof} \rangle$

lemma *lub-bin*: $x \sqsubseteq y \implies \text{lub } \{x, y\} = y$
 $\langle \text{proof} \rangle$

lemma *is-lub-maximal*: $S <| x \implies x \in S \implies S <<| x$
 $\langle \text{proof} \rangle$

lemma *lub-maximal*: $S <| x \implies x \in S \implies \text{lub } S = x$
 $\langle \text{proof} \rangle$

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

lemma *chainI*: $(\bigwedge i. Y\ i \sqsubseteq Y\ (\text{Suc } i)) \implies \text{chain } Y$
 $\langle \text{proof} \rangle$

lemma *chainE*: $\text{chain } Y \implies Y\ i \sqsubseteq Y\ (\text{Suc } i)$
 $\langle \text{proof} \rangle$

chains are monotone functions

lemma *chain-mono-less*: $\text{chain } Y \implies i < j \implies Y\ i \sqsubseteq Y\ j$
 $\langle \text{proof} \rangle$

lemma *chain-mono*: $\text{chain } Y \implies i \leq j \implies Y\ i \sqsubseteq Y\ j$
 $\langle \text{proof} \rangle$

lemma *chain-shift*: $\text{chain } Y \implies \text{chain } (\lambda i. Y\ (i + j))$
 $\langle \text{proof} \rangle$

technical lemmas about (least) upper bounds of chains

lemma *is-lub-rangeD1*: $\text{range } S <<| x \implies S\ i \sqsubseteq x$
 $\langle \text{proof} \rangle$

lemma *is-ub-range-shift*: $\text{chain } S \implies \text{range } (\lambda i. S\ (i + j)) <| x = \text{range } S <| x$
 $\langle \text{proof} \rangle$

lemma *is-lub-range-shift*: $\text{chain } S \implies \text{range } (\lambda i. S\ (i + j)) <<| x = \text{range } S <<| x$
 $\langle \text{proof} \rangle$

the lub of a constant chain is the constant

lemma *chain-const* [simp]: $\text{chain } (\lambda i. c)$
 $\langle \text{proof} \rangle$

lemma *is-lub-const*: $\text{range } (\lambda x. c) <<| c$
 $\langle \text{proof} \rangle$

lemma *lub-const* [simp]: $(\bigsqcup i. c) = c$
 $\langle \text{proof} \rangle$

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
 $\text{max-in-chain } i\ C \longleftrightarrow (\forall j. i \leq j \longrightarrow C\ i = C\ j)$

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

results about finite chains

lemma *max-in-chainI*: $(\bigwedge j. i \leq j \implies Y\ i = Y\ j) \implies \text{max-in-chain } i\ Y$
 $\langle \text{proof} \rangle$

lemma *max-in-chainD*: $\text{max-in-chain } i\ Y \implies i \leq j \implies Y\ i = Y\ j$
 $\langle \text{proof} \rangle$

lemma *finite-chainI*: $\text{chain } C \implies \text{max-in-chain } i\ C \implies \text{finite-chain } C$
 $\langle \text{proof} \rangle$

lemma *finite-chainE*: $\llbracket \text{finite-chain } C; \bigwedge i. \llbracket \text{chain } C; \text{max-in-chain } i\ C \rrbracket \implies R \rrbracket$
 $\implies R$
 $\langle \text{proof} \rangle$

lemma *lub-finch1*: $\text{chain } C \implies \text{max-in-chain } i\ C \implies \text{range } C <<| C\ i$
 $\langle \text{proof} \rangle$

lemma *lub-finch2*: $\text{finite-chain } C \implies \text{range } C <<| C\ (\text{LEAST } i. \text{max-in-chain } i\ C)$

$\langle proof \rangle$

lemma *finch-imp-finite-range*: $finite-chain\ Y \implies finite\ (range\ Y)$
 $\langle proof \rangle$

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*: $finite\ (range\ f)$
shows $\exists k. \forall i. r\ (f\ i)\ (f\ k)$
 $\langle proof \rangle$

lemma *finite-range-imp-finch*: $chain\ Y \implies finite\ (range\ Y) \implies finite-chain\ Y$
 $\langle proof \rangle$

lemma *bin-chain*: $x \sqsubseteq y \implies chain\ (\lambda i. if\ i=0\ then\ x\ else\ y)$
 $\langle proof \rangle$

lemma *bin-chainmax*: $x \sqsubseteq y \implies max-in-chain\ (Suc\ 0)\ (\lambda i. if\ i=0\ then\ x\ else\ y)$
 $\langle proof \rangle$

lemma *is-lub-bin-chain*: $x \sqsubseteq y \implies range\ (\lambda i::nat. if\ i=0\ then\ x\ else\ y) <<| y$
 $\langle proof \rangle$

the maximal element in a chain is its lub

lemma *lub-chain-maxelem*: $Y\ i = c \implies \forall i. Y\ i \sqsubseteq c \implies lub\ (range\ Y) = c$
 $\langle proof \rangle$

end

2 Classes cpo and pcpo

2.1 Complete partial orders

The class cpo of chain complete partial orders

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

default-sort *cpo*

context *cpo*
begin

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

lemma *cpo-lubI*: $chain\ S \implies range\ S <<| (\bigsqcup i. S\ i)$
 $\langle proof \rangle$

lemma *thelubE*: $\llbracket \text{chain } S; (\bigsqcup i. S\ i) = l \rrbracket \implies \text{range } S <<| l$
 $\langle \text{proof} \rangle$

Properties of the lub

lemma *is-ub-thelub*: $\text{chain } S \implies S\ x \sqsubseteq (\bigsqcup i. S\ i)$
 $\langle \text{proof} \rangle$

lemma *is-lub-thelub*: $\llbracket \text{chain } S; \text{range } S <| x \rrbracket \implies (\bigsqcup i. S\ i) \sqsubseteq x$
 $\langle \text{proof} \rangle$

lemma *lub-below-iff*: $\text{chain } S \implies (\bigsqcup i. S\ i) \sqsubseteq x \longleftrightarrow (\forall i. S\ i \sqsubseteq x)$
 $\langle \text{proof} \rangle$

lemma *lub-below*: $\llbracket \text{chain } S; \bigwedge i. S\ i \sqsubseteq x \rrbracket \implies (\bigsqcup i. S\ i) \sqsubseteq x$
 $\langle \text{proof} \rangle$

lemma *below-lub*: $\llbracket \text{chain } S; x \sqsubseteq S\ i \rrbracket \implies x \sqsubseteq (\bigsqcup i. S\ i)$
 $\langle \text{proof} \rangle$

lemma *lub-range-mono*: $\llbracket \text{range } X \subseteq \text{range } Y; \text{chain } Y; \text{chain } X \rrbracket \implies (\bigsqcup i. X\ i) \sqsubseteq (\bigsqcup i. Y\ i)$
 $\langle \text{proof} \rangle$

lemma *lub-range-shift*: $\text{chain } Y \implies (\bigsqcup i. Y\ (i + j)) = (\bigsqcup i. Y\ i)$
 $\langle \text{proof} \rangle$

lemma *maxinch-is-thelub*: $\text{chain } Y \implies \text{max-in-chain } i\ Y = ((\bigsqcup i. Y\ i) = Y\ i)$
 $\langle \text{proof} \rangle$

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

lemma *lub-mono*: $\llbracket \text{chain } X; \text{chain } Y; \bigwedge i. X\ i \sqsubseteq Y\ i \rrbracket \implies (\bigsqcup i. X\ i) \sqsubseteq (\bigsqcup i. Y\ i)$
 $\langle \text{proof} \rangle$

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

lemma *lub-eq*: $(\bigwedge i. X\ i = Y\ i) \implies (\bigsqcup i. X\ i) = (\bigsqcup i. Y\ i)$
 $\langle \text{proof} \rangle$

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 $\text{chain } (\lambda i. \bigsqcup j. Y\ i\ j)$

$\langle \text{proof} \rangle$

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 $(\bigsqcup i. \bigsqcup j. Y\ i\ j) = (\bigsqcup i. Y\ i\ i)$

$\langle \text{proof} \rangle$

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 $(\bigsqcup i. \bigsqcup j. Y\ i\ j) = (\bigsqcup j. \bigsqcup i. Y\ i\ j)$
<proof>

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 ($\langle \perp \rangle$)
 where *bottom* = (*THE* $x. \forall y. x \sqsubseteq y$)

lemma *minimal* [*iff*]: $\perp \sqsubseteq x$
<proof>

end

Old "UU" syntax:

abbreviation (*input*) *UU* \equiv *bottom*

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

<ML>

useful lemmas about \perp

lemma *below-bottom-iff* [*simp*]: $x \sqsubseteq \perp \longleftrightarrow x = \perp$
<proof>

lemma *eq-bottom-iff*: $x = \perp \longleftrightarrow x \sqsubseteq \perp$
<proof>

lemma *bottomI*: $x \sqsubseteq \perp \Longrightarrow x = \perp$
<proof>

lemma *lub-eq-bottom-iff*: $\text{chain } Y \Longrightarrow (\bigsqcup i. Y\ i) = \perp \longleftrightarrow (\forall i. Y\ i = \perp)$
<proof>

2.3 Chain-finite and flat cpos

further useful classes for HOLCF domains

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


```

subclass cpo
   $\langle proof \rangle$ 

lemma chfin2finch:  $chain\ Y \implies finite-chain\ Y$ 
   $\langle proof \rangle$ 

end

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

subclass chfin
   $\langle proof \rangle$ 

lemma flat-below-iff:  $x \sqsubseteq y \longleftrightarrow x = \perp \vee x = y$ 
   $\langle proof \rangle$ 

lemma flat-eq:  $a \neq \perp \implies a \sqsubseteq b = (a = b)$ 
   $\langle proof \rangle$ 

end

```

2.4 Discrete cpos

```

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

subclass po
   $\langle proof \rangle$ 

  In a discrete cpo, every chain is constant

lemma discrete-chain-const:
  assumes S:  $chain\ S$ 
  shows  $\exists x. S = (\lambda i. x)$ 
   $\langle proof \rangle$ 

subclass chfin
   $\langle proof \rangle$ 

end

```

3 Continuity and monotonicity

3.1 Definitions

definition *monofun* :: $('a::po \Rightarrow 'b::po) \Rightarrow bool$ — monotonicity

where $\text{monofun } f \longleftrightarrow (\forall x y. x \sqsubseteq y \longrightarrow f x \sqsubseteq f y)$

definition $\text{cont} :: ('a \Rightarrow 'b) \Rightarrow \text{bool}$

where $\text{cont } f = (\forall Y. \text{chain } Y \longrightarrow \text{range } (\lambda i. f (Y i)) <<| f (\bigsqcup i. Y i))$

lemma $\text{contI}: (\bigwedge Y. \text{chain } Y \Longrightarrow \text{range } (\lambda i. f (Y i)) <<| f (\bigsqcup i. Y i)) \Longrightarrow \text{cont } f$
 $\langle \text{proof} \rangle$

lemma $\text{contE}: \text{cont } f \Longrightarrow \text{chain } Y \Longrightarrow \text{range } (\lambda i. f (Y i)) <<| f (\bigsqcup i. Y i)$
 $\langle \text{proof} \rangle$

lemma $\text{monofunI}: (\bigwedge x y. x \sqsubseteq y \Longrightarrow f x \sqsubseteq f y) \Longrightarrow \text{monofun } f$
 $\langle \text{proof} \rangle$

lemma $\text{monofunE}: \text{monofun } f \Longrightarrow x \sqsubseteq y \Longrightarrow f x \sqsubseteq f y$
 $\langle \text{proof} \rangle$

3.2 Equivalence of alternate definition

monotone functions map chains to chains

lemma $\text{ch2ch-monofun}: \text{monofun } f \Longrightarrow \text{chain } Y \Longrightarrow \text{chain } (\lambda i. f (Y i))$
 $\langle \text{proof} \rangle$

monotone functions map upper bound to upper bounds

lemma $\text{ub2ub-monofun}: \text{monofun } f \Longrightarrow \text{range } Y <| u \Longrightarrow \text{range } (\lambda i. f (Y i)) <| f u$
 $\langle \text{proof} \rangle$

a lemma about binary chains

lemma $\text{binchain-cont}: \text{cont } f \Longrightarrow x \sqsubseteq y \Longrightarrow \text{range } (\lambda i::\text{nat}. f (\text{if } i = 0 \text{ then } x \text{ else } y)) <<| f y$
 $\langle \text{proof} \rangle$

continuity implies monotonicity

lemma $\text{cont2mono}: \text{cont } f \Longrightarrow \text{monofun } f$
 $\langle \text{proof} \rangle$

lemmas $\text{cont2monofunE} = \text{cont2mono} [\text{THEN monofunE}]$

lemmas $\text{ch2ch-cont} = \text{cont2mono} [\text{THEN ch2ch-monofun}]$

continuity implies preservation of lubs

lemma $\text{cont2contlubE}: \text{cont } f \Longrightarrow \text{chain } Y \Longrightarrow f (\bigsqcup i. Y i) = (\bigsqcup i. f (Y i))$
 $\langle \text{proof} \rangle$

lemma $\text{contI2}:$

fixes $f :: 'a \Rightarrow 'b$

assumes $\text{mono}: \text{monofun } f$

assumes below: $\bigwedge Y. \llbracket \text{chain } Y; \text{chain } (\lambda i. f (Y i)) \rrbracket \implies f (\bigsqcup i. Y i) \sqsubseteq (\bigsqcup i. f (Y i))$
shows $\text{cont } f$
 $\langle \text{proof} \rangle$

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]: $\text{cont } (\lambda x. x)$
 $\langle \text{proof} \rangle$

constant functions are continuous

lemma cont-const [simp , cont2cont]: $\text{cont } (\lambda x. c)$
 $\langle \text{proof} \rangle$

application of functions is continuous

lemma cont-apply :
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c$ **and** $t :: 'a \Rightarrow 'b$
assumes 1: $\text{cont } (\lambda x. t x)$
assumes 2: $\bigwedge x. \text{cont } (\lambda y. f x y)$
assumes 3: $\bigwedge y. \text{cont } (\lambda x. f x y)$
shows $\text{cont } (\lambda x. (f x) (t x))$
 $\langle \text{proof} \rangle$

lemma cont-compose : $\text{cont } c \implies \text{cont } (\lambda x. f x) \implies \text{cont } (\lambda x. c (f x))$
 $\langle \text{proof} \rangle$

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 $\text{cont } (\lambda x. \bigsqcup i. F i x)$
 $\langle \text{proof} \rangle$

if-then-else is continuous

lemma cont-if [simp , cont2cont]: $\text{cont } f \implies \text{cont } g \implies \text{cont } (\lambda x. \text{if } b \text{ then } f x \text{ else } g x)$
 $\langle \text{proof} \rangle$

3.5 Finite chains and flat pcpos

Monotone functions map finite chains to finite chains.

lemma $\text{monofun-finch2finch}$: $\text{monofun } f \implies \text{finite-chain } Y \implies \text{finite-chain } (\lambda n. f (Y n))$

$\langle proof \rangle$

The same holds for continuous functions.

lemma *cont-finch2finch*: $cont\ f \implies finite-chain\ Y \implies finite-chain\ (\lambda n. f\ (Y\ n))$
 $\langle proof \rangle$

All monotone functions with chain-finite domain are continuous.

lemma *chfindom-monofun2cont*: $monofun\ f \implies cont\ f$
for $f :: 'a::chfin \Rightarrow 'b$
 $\langle proof \rangle$

All strict functions with flat domain are continuous.

lemma *flatdom-strict2mono*: $f\ \bot = \bot \implies monofun\ f$
for $f :: 'a::flat \Rightarrow 'b::pcpo$
 $\langle proof \rangle$

lemma *flatdom-strict2cont*: $f\ \bot = \bot \implies cont\ f$
for $f :: 'a::flat \Rightarrow 'b::pcpo$
 $\langle proof \rangle$

All functions with discrete domain are continuous.

lemma *cont-discrete-cpo* [*simp*, *cont2cont*]: $cont\ f$
for $f :: 'a::discrete-cpo \Rightarrow 'b$
 $\langle proof \rangle$

4 Admissibility and compactness

4.1 Definitions

context *cpo*
begin

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

lemma *admI*: $(\bigwedge Y. \llbracket chain\ Y; \forall i. P\ (Y\ i) \rrbracket \implies P\ (\bigsqcup i. Y\ i)) \implies adm\ P$
 $\langle proof \rangle$

lemma *admD*: $adm\ P \implies chain\ Y \implies (\bigwedge i. P\ (Y\ i)) \implies P\ (\bigsqcup i. Y\ i)$
 $\langle proof \rangle$

lemma *admD2*: $adm\ (\lambda x. \neg P\ x) \implies chain\ Y \implies P\ (\bigsqcup i. Y\ i) \implies \exists i. P\ (Y\ i)$
 $\langle proof \rangle$

lemma *triv-admI*: $\forall x. P\ x \implies adm\ P$
 $\langle proof \rangle$

end

4.2 Admissibility on chain-finite types

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

lemma *adm-chfin* [*simp*]: *adm P for P :: 'a::chfin \Rightarrow bool*
<proof>

4.3 Admissibility of special formulae and propagation

context *cpo*

begin

lemma *adm-const* [*simp*]: *adm ($\lambda x. t$)*
<proof>

lemma *adm-conj* [*simp*]: *adm ($\lambda x. P x$) \Rightarrow adm ($\lambda x. Q x$) \Rightarrow adm ($\lambda x. P x \wedge Q x$)*
<proof>

lemma *adm-all* [*simp*]: *($\bigwedge y. \text{adm } (\lambda x. P x y)$) \Rightarrow adm ($\lambda x. \forall y. P x y$)*
<proof>

lemma *adm-ball* [*simp*]: *($\bigwedge y. y \in A \Rightarrow \text{adm } (\lambda x. P x y)$) \Rightarrow adm ($\lambda x. \forall y \in A. P x y$)*
<proof>

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>

lemma *adm-disj-lemma2*: $\forall n::\text{nat}. P n \vee Q n \Rightarrow (\forall i. \exists j \geq i. P j) \vee (\forall i. \exists j \geq i. Q j)$
<proof>

lemma *adm-disj* [*simp*]: *adm ($\lambda x. P x$) \Rightarrow adm ($\lambda x. Q x$) \Rightarrow adm ($\lambda x. P x \vee Q x$)*
<proof>

lemma *adm-imp* [*simp*]: *adm ($\lambda x. \neg P x$) \Rightarrow adm ($\lambda x. Q x$) \Rightarrow adm ($\lambda x. P x \longrightarrow Q x$)*
<proof>

lemma *adm-iff* [*simp*]: *adm ($\lambda x. P x \longrightarrow Q x$) \Rightarrow adm ($\lambda x. Q x \longrightarrow P x$) \Rightarrow adm ($\lambda x. P x \longleftrightarrow Q x$)*
<proof>

end

admissibility and continuity

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

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

lemma *adm-subst*: $\text{cont } (\lambda x. t \ x) \implies \text{adm } P \implies \text{adm } (\lambda x. P \ (t \ x))$
 ⟨proof⟩

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

4.4 Compactness

context *cpo*
begin

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

lemma *compactI*: $\text{adm } (\lambda x. k \not\sqsubseteq x) \implies \text{compact } k$
 ⟨proof⟩

lemma *compactD*: $\text{compact } k \implies \text{adm } (\lambda x. k \not\sqsubseteq x)$
 ⟨proof⟩

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

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

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

end

lemma *compact-chfin* [simp]: $\text{compact } x \text{ for } x :: 'a::\text{chfin}$
 ⟨proof⟩

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

admissibility and compactness

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

$compact\ k \implies cont\ (\lambda x. t\ x) \implies adm\ (\lambda x. k \not\sqsubseteq t\ x)$
 $\langle proof \rangle$

lemma *adm-neq-compact* [simp]: $compact\ k \implies cont\ (\lambda x. t\ x) \implies adm\ (\lambda x. t\ x \neq k)$
 $\langle proof \rangle$

lemma *adm-compact-neq* [simp]: $compact\ k \implies cont\ (\lambda x. t\ x) \implies adm\ (\lambda x. k \neq t\ x)$
 $\langle proof \rangle$

lemma *compact-bottom* [simp, intro]: $compact\ \perp$
 $\langle proof \rangle$

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 $adm\ P$
 $\langle proof \rangle$

lemmas *adm-lemmas* =

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

5 Class instances for the full function space

5.1 Full function space is a partial order

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

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

instance $\langle proof \rangle$
end

instance *fun* :: (*type*, *po*) *po*
 $\langle proof \rangle$

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

lemma *fun-belowI*: $(\bigwedge x. f\ x \sqsubseteq g\ x) \implies f \sqsubseteq g$
 $\langle proof \rangle$

lemma *fun-belowD*: $f \sqsubseteq g \implies f\ x \sqsubseteq g\ x$
 $\langle \text{proof} \rangle$

5.2 Full function space is chain complete

Properties of chains of functions.

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

lemma *ch2ch-fun*: $\text{chain } S \implies \text{chain } (\lambda i. S\ i\ x)$
 $\langle \text{proof} \rangle$

lemma *ch2ch-lambda*: $(\bigwedge x. \text{chain } (\lambda i. S\ i\ x)) \implies \text{chain } S$
 $\langle \text{proof} \rangle$

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

lemma *is-lub-lambda*: $(\bigwedge x. \text{range } (\lambda i. Y\ i\ x) <<| f\ x) \implies \text{range } Y <<| f$
 $\langle \text{proof} \rangle$

lemma *is-lub-fun*: $\text{chain } S \implies \text{range } S <<| (\lambda x. \bigsqcup i. S\ i\ x)$
for $S :: \text{nat} \Rightarrow 'a::\text{type} \Rightarrow 'b$
 $\langle \text{proof} \rangle$

lemma *lub-fun*: $\text{chain } S \implies (\bigsqcup i. S\ i) = (\lambda x. \bigsqcup i. S\ i\ x)$
for $S :: \text{nat} \Rightarrow 'a::\text{type} \Rightarrow 'b$
 $\langle \text{proof} \rangle$

instance *fun* :: $(\text{type}, \text{cpo})\ \text{cpo}$
 $\langle \text{proof} \rangle$

instance *fun* :: $(\text{type}, \text{discrete-cpo})\ \text{discrete-cpo}$
 $\langle \text{proof} \rangle$

5.3 Full function space is pointed

lemma *minimal-fun*: $(\lambda x. \perp) \sqsubseteq f$
 $\langle \text{proof} \rangle$

instance *fun* :: $(\text{type}, \text{pcpo})\ \text{pcpo}$
 $\langle \text{proof} \rangle$

lemma *inst-fun-pcpo*: $\perp = (\lambda x. \perp)$
 $\langle \text{proof} \rangle$

lemma *app-strict [simp]*: $\perp\ x = \perp$
 $\langle \text{proof} \rangle$

lemma *lambda-strict*: $(\lambda x. \perp) = \perp$
 $\langle \text{proof} \rangle$

5.4 Propagation of monotonicity and continuity

The lub of a chain of monotone functions is monotone.

lemma *adm-monofun*: *adm monofun*
 ⟨*proof*⟩

The lub of a chain of continuous functions is continuous.

lemma *adm-cont*: *adm cont*
 ⟨*proof*⟩

Function application preserves monotonicity and continuity.

lemma *mono2mono-fun*: *monofun f* \implies *monofun* ($\lambda x. f\ x\ y$)
 ⟨*proof*⟩

lemma *cont2cont-fun*: *cont f* \implies *cont* ($\lambda x. f\ x\ y$)
 ⟨*proof*⟩

lemma *cont-fun*: *cont* ($\lambda f. f\ x$)
 ⟨*proof*⟩

⟨*ML*⟩

lemma *cont* ($\lambda f. f\ x$) **and** *cont* ($\lambda f. f\ x\ y$) **and** *cont* ($\lambda f. f\ x\ y\ z$)
 ⟨*proof*⟩

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 *monofun f*
 ⟨*proof*⟩

lemma *cont2cont-lambda* [*simp*]:
 assumes *f*: $\bigwedge y. \text{cont } (\lambda x. f\ x\ y)$
 shows *cont f*
 ⟨*proof*⟩

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* :: *nat* \Rightarrow '*a*::*type* \Rightarrow '*b*
 ⟨*proof*⟩

6 The cpo of cartesian products

6.1 Unit type is a pcpo

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

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

instance
 $\langle proof \rangle$

end

instance *unit* :: *pcpo*
 $\langle proof \rangle$

6.2 Product type is a partial order

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

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

instance $\langle proof \rangle$

end

instance *prod* :: (*po*, *po*) *po*
 $\langle proof \rangle$

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

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

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

Pair (-,-) is monotone in both arguments

lemma *monofun-pair1*: *monofun* ($\lambda x. (x, y)$)
 $\langle proof \rangle$

lemma *monofun-pair2*: *monofun* ($\lambda y. (x, y)$)
 $\langle proof \rangle$

lemma *monofun-pair*: $x1 \sqsubseteq x2 \implies y1 \sqsubseteq y2 \implies (x1, y1) \sqsubseteq (x2, y2)$
 $\langle proof \rangle$

lemma *ch2ch-Pair* [*simp*]: $chain\ X \implies chain\ Y \implies chain\ (\lambda i. (X\ i, Y\ i))$
 $\langle proof \rangle$

fst and *snd* are monotone

lemma *fst-monofun*: $x \sqsubseteq y \implies fst\ x \sqsubseteq fst\ y$
 $\langle proof \rangle$

lemma *snd-monofun*: $x \sqsubseteq y \implies \text{snd } x \sqsubseteq \text{snd } y$
 ⟨proof⟩

lemma *monofun-fst*: *monofun fst*
 ⟨proof⟩

lemma *monofun-snd*: *monofun snd*
 ⟨proof⟩

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⟩

6.4 Product type is a cpo

lemma *is-lub-Pair*: $\text{range } A <<| x \implies \text{range } B <<| y \implies \text{range } (\lambda i. (A \ i, B \ i)) <<| (x, y)$
 ⟨proof⟩

lemma *lub-Pair*: $\text{chain } A \implies \text{chain } B \implies (\bigsqcup i. (A \ i, B \ i)) = (\bigsqcup i. A \ i, \bigsqcup i. B \ i)$
 for $A :: \text{nat} \Rightarrow 'a$ and $B :: \text{nat} \Rightarrow 'b$
 ⟨proof⟩

lemma *is-lub-prod*:
 fixes $S :: \text{nat} \Rightarrow ('a \times 'b)$
 assumes *chain* *S*
 shows $\text{range } S <<| (\bigsqcup i. \text{fst } (S \ i), \bigsqcup i. \text{snd } (S \ i))$
 ⟨proof⟩

lemma *lub-prod*: $\text{chain } S \implies (\bigsqcup i. S \ i) = (\bigsqcup i. \text{fst } (S \ i), \bigsqcup i. \text{snd } (S \ i))$
 for $S :: \text{nat} \Rightarrow 'a \times 'b$
 ⟨proof⟩

instance *prod* :: (*cpo*, *cpo*) *cpo*
 ⟨proof⟩

instance *prod* :: (*discrete-cpo*, *discrete-cpo*) *discrete-cpo*
 ⟨proof⟩

6.5 Product type is pointed

lemma *minimal-prod*: $(\perp, \perp) \sqsubseteq p$
 ⟨proof⟩

instance *prod* :: (*pcpo*, *pcpo*) *pcpo*
 ⟨*proof*⟩

lemma *inst-prod-pcpo*: $\perp = (\perp, \perp)$
 ⟨*proof*⟩

lemma *Pair-bottom-iff* [*simp*]: $(x, y) = \perp \longleftrightarrow x = \perp \wedge y = \perp$
 ⟨*proof*⟩

lemma *fst-strict* [*simp*]: *fst* $\perp = \perp$
 ⟨*proof*⟩

lemma *snd-strict* [*simp*]: *snd* $\perp = \perp$
 ⟨*proof*⟩

lemma *Pair-strict* [*simp*]: $(\perp, \perp) = \perp$
 ⟨*proof*⟩

lemma *split-strict* [*simp*]: *case-prod* *f* $\perp = f \perp \perp$
 ⟨*proof*⟩

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

lemma *cont-pair1*: *cont* ($\lambda x. (x, y)$)
 ⟨*proof*⟩

lemma *cont-pair2*: *cont* ($\lambda y. (x, y)$)
 ⟨*proof*⟩

lemma *cont-fst*: *cont* *fst*
 ⟨*proof*⟩

lemma *cont-snd*: *cont* *snd*
 ⟨*proof*⟩

lemma *cont2cont-Pair* [*simp*, *cont2cont*]:
assumes *f*: *cont* ($\lambda x. f x$)
assumes *g*: *cont* ($\lambda x. g x$)
shows *cont* ($\lambda x. (f x, g x)$)
 ⟨*proof*⟩

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 : $\text{cont } (\lambda x. g \ x)$
shows $\text{cont } (\lambda x. \text{case } g \ x \text{ of } (a, b) \Rightarrow f \ x \ a \ b)$
 $\langle \text{proof} \rangle$

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 $\text{cont } f$
 $\langle \text{proof} \rangle$

lemma *prod-cont-iff*: $\text{cont } f \longleftrightarrow (\forall y. \text{cont } (\lambda x. f \ (x, y))) \wedge (\forall x. \text{cont } (\lambda y. f \ (x, y)))$
 $\langle \text{proof} \rangle$

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 $\text{cont } (\lambda x. \text{case-prod } (f \ x) \ (g \ x))$
 $\langle \text{proof} \rangle$

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 $\text{cont } (\lambda x. \text{case } p \text{ of } (a, b) \Rightarrow f \ x \ a \ b)$
 $\langle \text{proof} \rangle$

Admissibility of predicates on product types.

lemma *adm-case-prod* [*simp*]:
assumes $\text{adm } (\lambda x. P \ x \ (fst \ (f \ x)) \ (snd \ (f \ x)))$
shows $\text{adm } (\lambda x. \text{case } f \ x \text{ of } (a, b) \Rightarrow P \ x \ a \ b)$
 $\langle \text{proof} \rangle$

6.7 Compactness and chain-finiteness

lemma *fst-below-iff*: $\text{fst } x \sqsubseteq y \longleftrightarrow x \sqsubseteq (y, \text{snd } x)$ **for** $x :: 'a \times 'b$
 $\langle \text{proof} \rangle$

lemma *snd-below-iff*: $\text{snd } x \sqsubseteq y \longleftrightarrow x \sqsubseteq (\text{fst } x, y)$ **for** $x :: 'a \times 'b$
 $\langle \text{proof} \rangle$

lemma *compact-fst*: $\text{compact } x \Longrightarrow \text{compact } (\text{fst } x)$
 $\langle \text{proof} \rangle$

lemma *compact-snd*: $\text{compact } x \Longrightarrow \text{compact } (\text{snd } x)$
 $\langle \text{proof} \rangle$

lemma *compact-Pair*: $\text{compact } x \Longrightarrow \text{compact } y \Longrightarrow \text{compact } (x, y)$
 $\langle \text{proof} \rangle$

lemma *compact-Pair-iff* [simp]: *compact* $(x, y) \longleftrightarrow \text{compact } x \wedge \text{compact } y$
 ⟨proof⟩

instance *prod* :: (*chfin*, *chfin*) *chfin*
 ⟨proof⟩

7 Discrete cpo types

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

7.1 Discrete cpo class instance

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

definition ($(\sqsubseteq) :: 'a \text{ discr} \Rightarrow 'a \text{ discr} \Rightarrow \text{bool}$) = (=)

instance
 ⟨proof⟩

end

7.2 *undiscr*

definition *undiscr* :: *'a::type discr* \Rightarrow *'a*
where *undiscr* $x = (\text{case } x \text{ of } \text{Discr } y \Rightarrow y)$

lemma *undiscr-Discr* [simp]: *undiscr* (*Discr* x) = x
 ⟨proof⟩

lemma *Discr-undiscr* [simp]: *Discr* (*undiscr* y) = y
 ⟨proof⟩

end

8 Subtypes of pcpo

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

8.1 Proving a subtype is a partial order

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

theorem (*in below*) *typedef-class-po*:

```

fixes Abs :: 'b::po  $\Rightarrow$  'a
assumes type: type-definition Rep Abs A
  and below: ( $\sqsubseteq$ )  $\equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$ 
shows class.po below
  <proof>

```

```

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

```

8.2 Proving a subtype is finite

```

lemma typedef-finite-UNIV:
  fixes Abs :: 'a::type  $\Rightarrow$  'b::type
  assumes type: type-definition Rep Abs A
  shows finite A  $\implies$  finite (UNIV :: 'b set)
  <proof>

```

8.3 Proving a subtype is chain-finite

```

lemma ch2ch-Rep:
  assumes below: ( $\sqsubseteq$ )  $\equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$ 
  shows chain S  $\implies$  chain ( $\lambda i. \text{Rep } (S i)$ )
  <proof>

```

```

theorem typedef-chfin:
  fixes Abs :: 'a::chfin  $\Rightarrow$  'b::po
  assumes type: type-definition Rep Abs A
  and below: ( $\sqsubseteq$ )  $\equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$ 
  shows OFCLASS('b, chfin-class)
  <proof>

```

8.4 Proving a subtype is complete

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

```

lemma typedef-is-lubI:
  assumes below: ( $\sqsubseteq$ )  $\equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$ 
  shows range ( $\lambda i. \text{Rep } (S i)$ )  $<<| \text{Rep } x \implies \text{range } S <<| x$ 
  <proof>

```

```

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 chain S  $\implies \text{Rep } (Abs (\bigsqcup i. \text{Rep } (S i))) = (\bigsqcup i. \text{Rep } (S i))$ 
  <proof>

```

```

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 range S  $<<|$  Abs ( $\bigsqcup i. \text{Rep } (S \ i)$ )
<proof>

```

```

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>

```

8.4.1 Continuity of *Rep* and *Abs*

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

```

theorem typedef-cont-Rep:
  fixes Abs :: 'a::cpo  $\Rightarrow$  '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 ( $\lambda x. f \ x$ )  $\implies$  cont ( $\lambda x. \text{Rep } (f \ x)$ )
<proof>

```

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  $\implies$  cont ( $\lambda x. \text{Abs } (f \ x)$ )
<proof>

```

8.5 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 $compact (Rep\ k) \implies compact\ k$
 $\langle proof \rangle$

8.6 Proving a subtype is pointed

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

theorem *typedef-pcpo-generic*:
fixes $Abs :: 'a::cpo \Rightarrow 'b::cpo$
assumes $type: type-definition\ Rep\ Abs\ A$
and $below: (\sqsubseteq) \equiv \lambda x\ y. Rep\ x \sqsubseteq Rep\ y$
and $z-in-A: z \in A$
and $z-least: \bigwedge x. x \in A \implies z \sqsubseteq x$
shows $OFCLASS('b, pcpo-class)$
 $\langle proof \rangle$

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 $below: (\sqsubseteq) \equiv \lambda x\ y. Rep\ x \sqsubseteq Rep\ y$
and $bottom-in-A: \perp \in A$
shows $OFCLASS('b, pcpo-class)$
 $\langle proof \rangle$

8.6.1 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 $below: (\sqsubseteq) \equiv \lambda x\ y. Rep\ x \sqsubseteq Rep\ y$
and $bottom-in-A: \perp \in A$
shows $Abs\ \perp = \perp$
 $\langle proof \rangle$

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

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  $x \in A \implies (\text{Abs } x = \perp) = (x = \perp)$ 
 $\langle \text{proof} \rangle$ 

```

```

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  $(\text{Rep } x = \perp) = (x = \perp)$ 
 $\langle \text{proof} \rangle$ 

```

8.7 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)
 $\langle \text{proof} \rangle$ 

```

8.8 HOLCF type definition package

$\langle ML \rangle$

end

9 The type of continuous functions

```

theory Cfun
  imports Cpodef
begin

```

9.1 Definition of continuous function type

```

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

```

```

cpodef ('a, 'b) cfun ( $\langle \langle \text{notation} = \langle \text{infix } \rightarrow \rangle \rangle \rightarrow / - \rangle$  [1, 0] 0) = cfun :: ('a  $\Rightarrow$ 
  'b) set
 $\langle \text{proof} \rangle$ 

```

```

type-notation (ASCII)
  cfun (infixr  $\langle - \rangle$  0)

```

```

notation (ASCII)
  Rep-cfun ( $\langle \langle \text{notation} = \langle \text{infix } \$ \rangle \rangle \$ / - \rangle$  [999, 1000] 999)

```

notation

Rep-cfun ($\langle \langle \text{notation} = \langle \text{infix } \cdot \rangle \rangle \cdot \rangle \cdot \rangle \rangle [999, 1000] 999$)

9.2 Syntax for continuous lambda abstraction

syntax *-cabs* :: [*logic*, *logic*] \Rightarrow *logic*

$\langle ML \rangle$

Syntax for nested abstractions

syntax (*ASCII*)

-Lambda :: [*cargs*, *logic*] \Rightarrow *logic* ($\langle \langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } LAM \rangle \rangle LAM \cdot \rangle \rangle \cdot \rangle$ [1000, 10] 10)

syntax

-Lambda :: [*cargs*, *logic*] \Rightarrow *logic* ($\langle \langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder } \Lambda \rangle \rangle \Lambda \cdot \rangle \rangle \cdot \rangle$ [1000, 10] 10)

syntax-consts

-Lambda \Rightarrow *Abs-cfun*

$\langle ML \rangle$

Dummy patterns for continuous abstraction

translations

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

9.3 Continuous function space is pointed

lemma *bottom-cfun*: $\perp \in \text{cfun}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

instance *cfun* :: (*cpo*, *pcpo*) *pcpo*

$\langle \text{proof} \rangle$

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*]: $\perp \cdot x = \perp$

$\langle \text{proof} \rangle$

lemma *LAM-strict* [*simp*]: $(\Lambda x. \perp) = \perp$

$\langle \text{proof} \rangle$

for compatibility with old HOLCF-Version

lemma *inst-cfun-pcpo*: $\perp = (\Lambda x. \perp)$
 $\langle proof \rangle$

9.4 Basic properties of continuous functions

Beta-equality for continuous functions

lemma *Abs-cfun-inverse2*: $cont\ f \implies Rep-cfun\ (Abs-cfun\ f) = f$
 $\langle proof \rangle$

lemma *beta-cfun*: $cont\ f \implies (\Lambda x. f\ x) \cdot u = f\ u$
 $\langle proof \rangle$

9.4.1 Beta-reduction simproc

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

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

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

$\langle ML \rangle$

Eta-equality for continuous functions

lemma *eta-cfun*: $(\Lambda x. f \cdot x) = f$
 $\langle proof \rangle$

Extensionality for continuous functions

lemma *cfun-eq-iff*: $f = g \longleftrightarrow (\forall x. f \cdot x = g \cdot x)$
 $\langle proof \rangle$

lemma *cfun-eqI*: $(\bigwedge x. f \cdot x = g \cdot x) \implies f = g$
 $\langle proof \rangle$

Extensionality wrt. ordering for continuous functions

lemma *cfun-below-iff*: $f \sqsubseteq g \longleftrightarrow (\forall x. f \cdot x \sqsubseteq g \cdot x)$
 $\langle proof \rangle$

lemma *cfun-belowI*: $(\bigwedge x. f \cdot x \sqsubseteq g \cdot x) \implies f \sqsubseteq g$
 $\langle proof \rangle$

Congruence for continuous function application

lemma *cfun-cong*: $f = g \implies x = y \implies f \cdot x = g \cdot y$

$\langle proof \rangle$

lemma *cfun-fun-cong*: $f = g \implies f \cdot x = g \cdot x$
 $\langle proof \rangle$

lemma *cfun-arg-cong*: $x = y \implies f \cdot x = f \cdot y$
 $\langle proof \rangle$

9.5 Continuity of application

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

lemma *cont-Rep-cfun2*: $cont (\lambda x. f \cdot x)$
 $\langle proof \rangle$

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

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

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

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

lemma *contlub-cfun-arg*: $chain\ Y \implies f \cdot (\bigsqcup i. Y\ i) = (\bigsqcup i. f \cdot (Y\ i))$
 $\langle proof \rangle$

lemma *contlub-cfun-fun*: $chain\ F \implies (\bigsqcup i. F\ i) \cdot x = (\bigsqcup i. F\ i \cdot x)$
 $\langle proof \rangle$

monotonicity of application

lemma *monofun-cfun-fun*: $f \sqsubseteq g \implies f \cdot x \sqsubseteq g \cdot x$
 $\langle proof \rangle$

lemma *monofun-cfun-arg*: $x \sqsubseteq y \implies f \cdot x \sqsubseteq f \cdot y$
 $\langle proof \rangle$

lemma *monofun-cfun*: $f \sqsubseteq g \implies x \sqsubseteq y \implies f \cdot x \sqsubseteq g \cdot y$
 $\langle proof \rangle$

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

lemma *chain-monofun*: $chain\ Y \implies chain\ (\lambda i. f \cdot (Y\ i))$
 $\langle proof \rangle$

lemma *ch2ch-Rep-cfunR*: $chain\ Y \implies chain\ (\lambda i. f \cdot (Y\ i))$
 $\langle proof \rangle$

lemma *ch2ch-Rep-cfunL*: $chain\ F \implies chain\ (\lambda i. (F\ i) \cdot x)$
 $\langle proof \rangle$

lemma *ch2ch-Rep-cfun* [simp]: $chain\ F \implies chain\ Y \implies chain\ (\lambda i. (F\ i) \cdot (Y\ i))$

$\langle proof \rangle$

lemma *ch2ch-LAM* [simp]:

$(\bigwedge x. chain (\lambda i. S i x)) \implies (\bigwedge i. cont (\lambda x. S i x)) \implies chain (\lambda i. \bigwedge x. S i x)$
 $\langle proof \rangle$

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

lemma *lub-APP*: $chain F \implies chain Y \implies (\bigsqcup i. F i \cdot (Y i)) = (\bigsqcup i. F i) \cdot (\bigsqcup i. Y i)$
 $\langle proof \rangle$

lemma *lub-LAM*:

assumes $\bigwedge x. chain (\lambda i. F i x)$
and $\bigwedge i. cont (\lambda x. F i x)$
shows $(\bigsqcup i. \bigwedge x. F i x) = (\bigwedge x. \bigsqcup i. F i x)$
 $\langle proof \rangle$

lemmas *lub-distrib* = *lub-APP* *lub-LAM*

strictness

lemma *strictI*: $f \cdot x = \perp \implies f \cdot \perp = \perp$
 $\langle proof \rangle$

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

lemma *lub-cfun*: $chain F \implies (\bigsqcup i. F i) = (\bigwedge x. \bigsqcup i. F i x)$
 $\langle proof \rangle$

9.6 Continuity simplification procedure

cont2cont lemma for *Rep-cfun*

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

assumes $f: cont (\lambda x. f x)$
assumes $t: cont (\lambda x. t x)$
shows $cont (\lambda x. (f x) \cdot (t x))$
 $\langle proof \rangle$

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

lemma *cont-APP-app* [simp]: $cont f \implies cont g \implies cont (\lambda x. ((f x) \cdot (g x)) s)$
 $\langle proof \rangle$

lemma *cont-APP-app-app* [simp]: $cont f \implies cont g \implies cont (\lambda x. ((f x) \cdot (g x)) s t)$
 $\langle proof \rangle$

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

lemma *cont2mono-LAM*:

$$\begin{aligned} & \llbracket \bigwedge x. \text{cont } (\lambda y. f x y); \bigwedge y. \text{monofun } (\lambda x. f x y) \rrbracket \\ & \implies \text{monofun } (\lambda x. \bigwedge y. f x y) \\ & \langle \text{proof} \rangle \end{aligned}$$

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

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 $\text{cont } (\lambda x. \bigwedge y. f x y)$
 $\langle \text{proof} \rangle$

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 $f: \text{cont } (\lambda p. f (\text{fst } p) (\text{snd } p))$
shows $\text{cont } (\lambda x. \bigwedge y. f x y)$
 $\langle \text{proof} \rangle$

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

$(\bigwedge y::'a::\text{discrete-cpo}. \text{cont } (\lambda x. f x y)) \implies \text{cont } (\lambda x. \bigwedge y. f x y)$
 $\langle \text{proof} \rangle$

9.7 Miscellaneous

Monotonicity of *Abs-cfun*

lemma *monofun-LAM*: $\text{cont } f \implies \text{cont } g \implies (\bigwedge x. f x \sqsubseteq g x) \implies (\bigwedge x. f x) \sqsubseteq (\bigwedge x. g x)$
 $\langle \text{proof} \rangle$

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

lemma *chfin-Rep-cfunR*: $\text{chain } Y \implies \forall s. \exists n. (\text{LUB } i. Y i) \cdot s = Y n \cdot s$
for $Y :: \text{nat} \Rightarrow 'a::\text{cpo} \rightarrow 'b::\text{chfin}$
 $\langle \text{proof} \rangle$

lemma *adm-chfindom*: $\text{adm } (\lambda(u::'a::\text{cpo} \rightarrow 'b::\text{chfin}). P(u \cdot s))$
 $\langle \text{proof} \rangle$

9.8 Continuous injection-retraction pairs

Continuous retractions are strict.

lemma *retraction-strict*: $\forall x. f \cdot (g \cdot x) = x \implies f \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *injection-eq*: $\forall x. f \cdot (g \cdot x) = x \implies (g \cdot x = g \cdot y) = (x = y)$

$\langle proof \rangle$

lemma *injection-below*: $\forall x. f \cdot (g \cdot x) = x \implies (g \cdot x \sqsubseteq g \cdot y) = (x \sqsubseteq y)$
 $\langle proof \rangle$

lemma *injection-defined-rev*: $\forall x. f \cdot (g \cdot x) = x \implies g \cdot z = \perp \implies z = \perp$
 $\langle proof \rangle$

lemma *injection-defined*: $\forall x. f \cdot (g \cdot x) = x \implies z \neq \perp \implies g \cdot z \neq \perp$
 $\langle proof \rangle$

a result about functions with flat codomain

lemma *flat-eqI*: $x \sqsubseteq y \implies x \neq \perp \implies x = y$
for $x\ y :: 'a::flat$
 $\langle proof \rangle$

lemma *flat-codom*: $f \cdot x = c \implies f \cdot \perp = \perp \vee (\forall z. f \cdot z = c)$
for $c :: 'b::flat$
 $\langle proof \rangle$

9.9 Identity and composition

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

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

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

lemma *ID1* [*simp*]: $ID \cdot x = x$
 $\langle proof \rangle$

lemma *cfcomp1*: $(f\ oo\ g) = (\lambda x. f \cdot (g \cdot x))$
 $\langle proof \rangle$

lemma *cfcomp2* [*simp*]: $(f\ oo\ g) \cdot x = f \cdot (g \cdot x)$
 $\langle proof \rangle$

lemma *cfcomp-LAM*: $cont\ g \implies f\ oo\ (\lambda x. g\ x) = (\lambda x. f \cdot (g\ x))$
 $\langle proof \rangle$

lemma *cfcomp-strict* [*simp*]: $\perp\ oo\ f = \perp$
 $\langle proof \rangle$

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*]: $f \text{ oo } ID = f$
 $\langle \text{proof} \rangle$

lemma $ID3$ [*simp*]: $ID \text{ oo } f = f$
 $\langle \text{proof} \rangle$

lemma *assoc-oo*: $f \text{ oo } (g \text{ oo } h) = (f \text{ oo } g) \text{ oo } h$
 $\langle \text{proof} \rangle$

9.10 Strictified functions

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

lemma *cont2cont-if-bottom* [*cont2cont*, *simp*]:
assumes $f: \text{cont } (\lambda x. f x)$
and $g: \text{cont } (\lambda x. g x)$
shows $\text{cont } (\lambda x. \text{if } f x = \perp \text{ then } \perp \text{ else } g x)$
 $\langle \text{proof} \rangle$

lemma *seq-conv-if*: $seq \cdot x = (\text{if } x = \perp \text{ then } \perp \text{ else } ID)$
 $\langle \text{proof} \rangle$

lemma *seq-simps* [*simp*]:
 $seq \cdot \perp = \perp$
 $seq \cdot x \cdot \perp = \perp$
 $x \neq \perp \implies seq \cdot x = ID$
 $\langle \text{proof} \rangle$

definition *strictify* $:: ('a::pcpo \rightarrow 'b::pcpo) \rightarrow 'a \rightarrow 'b$
where $strictify = (\lambda f x. seq \cdot x \cdot (f \cdot x))$

lemma *strictify-conv-if*: $strictify \cdot f \cdot x = (\text{if } x = \perp \text{ then } \perp \text{ else } f \cdot x)$
 $\langle \text{proof} \rangle$

lemma *strictify1* [*simp*]: $strictify \cdot f \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *strictify2* [*simp*]: $x \neq \perp \implies strictify \cdot f \cdot x = f \cdot x$
 $\langle \text{proof} \rangle$

9.11 Continuity of let-bindings

lemma *cont2cont-Let*:

```

assumes  $f: cont (\lambda x. f x)$ 
assumes  $g1: \bigwedge y. cont (\lambda x. g x y)$ 
assumes  $g2: \bigwedge x. cont (\lambda y. g x y)$ 
shows  $cont (\lambda x. let y = f x in g x y)$ 
 $\langle proof \rangle$ 

```

```

lemma cont2cont-Let' [simp, cont2cont]:
  assumes  $f: cont (\lambda x. f x)$ 
  assumes  $g: cont (\lambda p. g (fst p) (snd p))$ 
  shows  $cont (\lambda x. let y = f x in g x y)$ 
   $\langle proof \rangle$ 

```

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

```

lemma cont2cont-Let-simple [simp, cont2cont]:
  assumes  $\bigwedge y. cont (\lambda x. g x y)$ 
  shows  $cont (\lambda x. let y = t in g x y)$ 
   $\langle proof \rangle$ 

```

end

10 Continuous deflations and ep-pairs

```

theory Deflation
  imports Cfun
begin

```

10.1 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:  $d \sqsubseteq ID$ 
 $\langle proof \rangle$ 

```

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

```

lemma fixes-eq-range:  $\{x. d \cdot x = x\} = range (\lambda x. d \cdot x)$ 
 $\langle proof \rangle$ 

```

```

lemma range-eq-fixes:  $range (\lambda x. d \cdot x) = \{x. d \cdot x = x\}$ 
 $\langle proof \rangle$ 

```

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 $d \sqsubseteq f$
 $\langle proof \rangle$

lemma *belowD*: $\llbracket f \sqsubseteq d; f \cdot x = x \rrbracket \implies d \cdot x = x$
 $\langle proof \rangle$

end

lemma *deflation-strict*: $deflation\ d \implies d \cdot \perp = \perp$
 $\langle proof \rangle$

lemma *adm-deflation*: $adm\ (\lambda d. deflation\ d)$
 $\langle proof \rangle$

lemma *deflation-ID*: $deflation\ ID$
 $\langle proof \rangle$

lemma *deflation-bottom*: $deflation\ \perp$
 $\langle proof \rangle$

lemma *deflation-below-iff*: $deflation\ p \implies deflation\ q \implies p \sqsubseteq q \longleftrightarrow (\forall x. p \cdot x = x \longrightarrow q \cdot x = x)$
 $\langle proof \rangle$

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 $f \sqsubseteq g \implies f \cdot (g \cdot x) = f \cdot x$
 $\langle proof \rangle$

lemma *deflation-below-comp2*: $deflation\ f \implies deflation\ g \implies f \sqsubseteq g \implies g \cdot (f \cdot x) = f \cdot x$
 $\langle proof \rangle$

10.2 Deflations with finite range

lemma *finite-range-imp-finite-fixes*:
assumes $finite\ (range\ f)$
shows $finite\ \{x. f \cdot x = x\}$
 $\langle proof \rangle$

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

lemma *finite-range*: $finite\ (range\ (\lambda x. d \cdot x))$

$\langle \text{proof} \rangle$

lemma *finite-image*: *finite* $((\lambda x. d \cdot x) \cdot A)$
 $\langle \text{proof} \rangle$

lemma *compact*: *compact* $(d \cdot x)$
 $\langle \text{proof} \rangle$

end

lemma *finite-deflation-intro*: *deflation* $d \implies \text{finite } \{x. d \cdot x = x\} \implies \text{finite-deflation } d$
 $\langle \text{proof} \rangle$

lemma *finite-deflation-imp-deflation*: *finite-deflation* $d \implies \text{deflation } d$
 $\langle \text{proof} \rangle$

lemma *finite-deflation-bottom*: *finite-deflation* \perp
 $\langle \text{proof} \rangle$

10.3 Continuous embedding-projection pairs

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

lemma *e-below-iff* [*simp*]: $e \cdot x \sqsubseteq e \cdot y \longleftrightarrow x \sqsubseteq y$
 $\langle \text{proof} \rangle$

lemma *e-eq-iff* [*simp*]: $e \cdot x = e \cdot y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *p-eq-iff*: $e \cdot (p \cdot x) = x \implies e \cdot (p \cdot y) = y \implies p \cdot x = p \cdot y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *p-inverse*: $(\exists x. y = e \cdot x) \longleftrightarrow e \cdot (p \cdot y) = y$
 $\langle \text{proof} \rangle$

lemma *e-below-iff-below-p*: $e \cdot x \sqsubseteq y \longleftrightarrow x \sqsubseteq p \cdot y$
 $\langle \text{proof} \rangle$

lemma *compact-e-rev*: *compact* $(e \cdot x) \implies \text{compact } x$
 $\langle \text{proof} \rangle$

lemma *compact-e*:
assumes *compact* x
shows *compact* $(e \cdot x)$

$\langle \text{proof} \rangle$

lemma *compact-e-iff*: $\text{compact } (e \cdot x) \longleftrightarrow \text{compact } x$
 $\langle \text{proof} \rangle$

Deflations from ep-pairs

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

lemma *deflation-e-d-p*:
 assumes *deflation d*
 shows $\text{deflation } (e \text{ oo } d \text{ oo } p)$
 $\langle \text{proof} \rangle$

lemma *finite-deflation-e-d-p*:
 assumes *finite-deflation d*
 shows $\text{finite-deflation } (e \text{ oo } d \text{ oo } p)$
 $\langle \text{proof} \rangle$

lemma *deflation-p-d-e*:
 assumes *deflation d*
 assumes $d: \bigwedge x. d \cdot x \sqsubseteq e \cdot (p \cdot x)$
 shows $\text{deflation } (p \text{ oo } d \text{ oo } e)$
 $\langle \text{proof} \rangle$

lemma *finite-deflation-p-d-e*:
 assumes *finite-deflation d*
 assumes $d: \bigwedge x. d \cdot x \sqsubseteq e \cdot (p \cdot x)$
 shows $\text{finite-deflation } (p \text{ oo } d \text{ oo } e)$
 $\langle \text{proof} \rangle$

end

10.4 Uniqueness of ep-pairs

lemma *ep-pair-unique-e-lemma*:
 assumes $1: \text{ep-pair } e1 \ p$
 and $2: \text{ep-pair } e2 \ p$
 shows $e1 \sqsubseteq e2$
 $\langle \text{proof} \rangle$

lemma *ep-pair-unique-e*: $\text{ep-pair } e1 \ p \implies \text{ep-pair } e2 \ p \implies e1 = e2$
 $\langle \text{proof} \rangle$

lemma *ep-pair-unique-p-lemma*:
 assumes $1: \text{ep-pair } e \ p1$
 and $2: \text{ep-pair } e \ p2$
 shows $p1 \sqsubseteq p2$
 $\langle \text{proof} \rangle$

lemma *ep-pair-unique-p*: $ep\text{-}pair\ e\ p1 \implies ep\text{-}pair\ e\ p2 \implies p1 = p2$
 $\langle proof \rangle$

10.5 Composing ep-pairs

lemma *ep-pair-ID-ID*: $ep\text{-}pair\ ID\ ID$
 $\langle proof \rangle$

lemma *ep-pair-comp*:
 assumes $ep\text{-}pair\ e1\ p1$ and $ep\text{-}pair\ e2\ p2$
 shows $ep\text{-}pair\ (e2\ oo\ e1)\ (p1\ oo\ p2)$
 $\langle proof \rangle$

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

lemma *e-strict [simp]*: $e \cdot \perp = \perp$
 $\langle proof \rangle$

lemma *e-bottom-iff [simp]*: $e \cdot x = \perp \longleftrightarrow x = \perp$
 $\langle proof \rangle$

lemma *e-defined*: $x \neq \perp \implies e \cdot x \neq \perp$
 $\langle proof \rangle$

lemma *p-strict [simp]*: $p \cdot \perp = \perp$
 $\langle proof \rangle$

lemmas *stricts* = $e\text{-}strict\ p\text{-}strict$

end

end

11 The type of strict products

theory *Sprod*
 imports *Cfun*
begin

11.1 Definition of strict product type

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

pcpodef ($'a::pcpo, 'b::pcpo$) *sprod* ($\langle \langle notation = \langle infix\ strict\ product \rangle \rangle - \otimes / - \rangle$)
 $[21, 20]\ 20) =$

sprod :: ('a × 'b) set
 ⟨proof⟩

instance *sprod* :: ({*chfin*,*pcpo*}, {*chfin*,*pcpo*}) *chfin*
 ⟨proof⟩

type-notation (*ASCII*)
sprod (**infixr** <*> 20)

11.2 Definitions of constants

definition *sfst* :: ('a::pcpo ** 'b::pcpo) → 'a
 where *sfst* = (λ p. fst (Rep-sprod p))

definition *ssnd* :: ('a::pcpo ** 'b::pcpo) → 'b
 where *ssnd* = (λ p. snd (Rep-sprod p))

definition *spair* :: 'a::pcpo → 'b::pcpo → ('a ** 'b)
 where *spair* = (λ a b. Abs-sprod (seq·b·a, seq·a·b))

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

syntax

-*stuple* :: [logic, args] ⇒ logic (⟨(⟨indent=1 notation=⟨mixfix strict tuple⟩⟩'(:-,/
 -:'))⟩)

syntax-consts

-*stuple* ⇐ *spair*

translations

(:x, y, z:) ⇐ (:x, (:y, z):)
 (:x, y:) ⇐ CONST *spair*·x·y

translations

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

11.3 Case analysis

lemma *spair-sprod*: (seq·b·a, seq·a·b) ∈ *sprod*
 ⟨proof⟩

lemma *Rep-sprod-spair*: Rep-sprod (:a, b:) = (seq·b·a, seq·a·b)
 ⟨proof⟩

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 $y \neq \perp$

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

11.4 Properties of *spair*

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *spair-inject*: $x \neq \perp \implies y \neq \perp \implies (:x, y:) = (:a, b:) \implies x = a \wedge y = b$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *sprodE2*: $(\bigwedge x y. p = (:x, y:) \implies Q) \implies Q$

$\langle proof \rangle$

11.5 Properties of *sfst* and *ssnd*

lemma *sfst-strict* [*simp*]: $sfst.\perp = \perp$
 $\langle proof \rangle$

lemma *ssnd-strict* [*simp*]: $ssnd.\perp = \perp$
 $\langle proof \rangle$

lemma *sfst-spair* [*simp*]: $y \neq \perp \implies sfst.(x, y) = x$
 $\langle proof \rangle$

lemma *ssnd-spair* [*simp*]: $x \neq \perp \implies ssnd.(x, y) = y$
 $\langle proof \rangle$

lemma *sfst-bottom-iff* [*simp*]: $sfst.p = \perp \longleftrightarrow p = \perp$
 $\langle proof \rangle$

lemma *ssnd-bottom-iff* [*simp*]: $ssnd.p = \perp \longleftrightarrow p = \perp$
 $\langle proof \rangle$

lemma *sfst-defined*: $p \neq \perp \implies sfst.p \neq \perp$
 $\langle proof \rangle$

lemma *ssnd-defined*: $p \neq \perp \implies ssnd.p \neq \perp$
 $\langle proof \rangle$

lemma *spair-sfst-ssnd*: $(:sfst.p, ssnd.p:) = p$
 $\langle proof \rangle$

lemma *below-sprod*: $x \sqsubseteq y \longleftrightarrow sfst.x \sqsubseteq sfst.y \wedge ssnd.x \sqsubseteq ssnd.y$
 $\langle proof \rangle$

lemma *eq-sprod*: $x = y \longleftrightarrow sfst.x = sfst.y \wedge ssnd.x = ssnd.y$
 $\langle proof \rangle$

lemma *sfst-below-iff*: $sfst.x \sqsubseteq y \longleftrightarrow x \sqsubseteq (:y, ssnd.x:)$
 $\langle proof \rangle$

lemma *ssnd-below-iff*: $ssnd.x \sqsubseteq y \longleftrightarrow x \sqsubseteq (:sfst.x, y:)$
 $\langle proof \rangle$

11.6 Compactness

lemma *compact-sfst*: $compact\ x \implies compact\ (sfst.x)$
 $\langle proof \rangle$

lemma *compact-ssnd*: $compact\ x \implies compact\ (ssnd.x)$
 $\langle proof \rangle$

lemma *compact-spair*: $\text{compact } x \implies \text{compact } y \implies \text{compact } (:x, y)$
 $\langle \text{proof} \rangle$

lemma *compact-spair-iff*: $\text{compact } (:x, y) \longleftrightarrow x = \perp \vee y = \perp \vee (\text{compact } x \wedge \text{compact } y)$
 $\langle \text{proof} \rangle$

11.7 Properties of *ssplit*

lemma *ssplit1* [*simp*]: $\text{ssplit}.f.\perp = \perp$
 $\langle \text{proof} \rangle$

lemma *ssplit2* [*simp*]: $x \neq \perp \implies y \neq \perp \implies \text{ssplit}.f.(:x, y) = f.x.y$
 $\langle \text{proof} \rangle$

lemma *ssplit3* [*simp*]: $\text{ssplit}.spair.z = z$
 $\langle \text{proof} \rangle$

11.8 Strict product preserves flatness

instance *sprod* :: $(\text{flat}, \text{flat}) \text{ flat}$
 $\langle \text{proof} \rangle$

end

12 The type of lifted values

theory *Up*
imports *Cfun*
begin

12.1 Definition of new type for lifting

datatype *'a u* ($\langle (\langle \text{notation} = \langle \text{postfix lifting} \rangle \rangle - \perp) \rangle [1000] 999$) = *Ibottom* | *Iup* *'a*

primrec *Ifup* :: $(\text{'a} \rightarrow \text{'b::pcpo}) \Rightarrow \text{'a u} \Rightarrow \text{'b}$
where
 $\text{Ifup } f \text{ Ibottom} = \perp$
 $\mid \text{Ifup } f (\text{Iup } x) = f.x$

12.2 Ordering on lifted cpo

instantiation *u* :: $(\text{cpo}) \text{ below}$
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 $\langle proof \rangle$

end

lemma *minimal-up* [iff]: $Ibottom \sqsubseteq z$
 $\langle proof \rangle$

lemma *not-Iup-below* [iff]: $Iup\ x \not\sqsubseteq Ibottom$
 $\langle proof \rangle$

lemma *Iup-below* [iff]: $(Iup\ x \sqsubseteq Iup\ y) = (x \sqsubseteq y)$
 $\langle proof \rangle$

12.3 Lifted cpo is a partial order

instance $u :: (cpo)\ po$
 $\langle proof \rangle$

12.4 Lifted cpo is a cpo

lemma *is-lub-Iup*: $range\ S \ll x \implies range\ (\lambda i. Iup\ (S\ i)) \ll Iup\ x$
 $\langle proof \rangle$

lemma *up-chain-lemma*:

assumes $Y: chain\ Y$

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

$| A\ k$ **where** $\forall i. Iup\ (A\ i) = Y\ (i + k)$ **and** *chain* A **and** $range\ Y \ll Iup$
 $(\bigsqcup i. A\ i)$
 $\langle proof \rangle$

instance $u :: (cpo)\ cpo$
 $\langle proof \rangle$

12.5 Lifted cpo is pointed

instance $u :: (cpo)\ pcpo$
 $\langle proof \rangle$

for compatibility with old HOLCF-Version

lemma *inst-up-pcpo*: $\perp = Ibottom$
 $\langle proof \rangle$

12.6 Continuity of *Iup* and *Ifup*

continuity for *Iup*

lemma *cont-Iup*: *cont Iup*
 $\langle proof \rangle$

continuity for *Ifup*

lemma *cont-Ifup1*: *cont* ($\lambda f. Ifup f x$)
 $\langle proof \rangle$

lemma *monofun-Ifup2*: *monofun* ($\lambda x. Ifup f x$)
 $\langle proof \rangle$

lemma *cont-Ifup2*: *cont* ($\lambda x. Ifup f x$)
 $\langle proof \rangle$

12.7 Continuous versions of constants

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

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

translations

case l of XCONST up.x $\Rightarrow t \rightleftharpoons$ CONST fup.($\Lambda x. t$).l
case l of (XCONST up :: 'a).x $\Rightarrow t \rightarrow$ CONST fup.($\Lambda x. t$).l
 $\Lambda(XCONST up.x). t \rightleftharpoons CONST fup.(\Lambda x. t)$

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

lemma *Exh-Up*: $z = \perp \vee (\exists x. z = up.x)$
 $\langle proof \rangle$

lemma *up-eq [simp]*: $(up.x = up.y) = (x = y)$
 $\langle proof \rangle$

lemma *up-inject*: $up.x = up.y \implies x = y$
 $\langle proof \rangle$

lemma *up-defined [simp]*: $up.x \neq \perp$
 $\langle proof \rangle$

lemma *not-up-less-UU*: $up.x \not\sqsubseteq \perp$
 $\langle proof \rangle$

lemma *up-below [simp]*: $up.x \sqsubseteq up.y \longleftrightarrow x \sqsubseteq y$
 $\langle proof \rangle$

lemma *upE [case-names bottom up, cases type: u]*: $\llbracket p = \perp \implies Q; \bigwedge x. p = up.x \implies Q \rrbracket \implies Q$
 $\langle proof \rangle$

lemma *up-induct* [*case-names bottom up, induct type: u*]: $P \perp \implies (\bigwedge x. P (up \cdot x)) \implies P x$
 ⟨*proof*⟩

lifting preserves chain-finiteness

lemma *up-chain-cases*:
assumes $Y: chain\ Y$
obtains $\forall i. Y\ i = \perp$
 $| A\ k$ **where** $\forall i. up \cdot (A\ i) = Y\ (i + k)$ **and** *chain* A **and** $(\bigsqcup i. Y\ i) = up \cdot (\bigsqcup i. A\ i)$
 ⟨*proof*⟩

lemma *compact-up*: $compact\ x \implies compact\ (up \cdot x)$
 ⟨*proof*⟩

lemma *compact-upD*: $compact\ (up \cdot x) \implies compact\ x$
 ⟨*proof*⟩

lemma *compact-up-iff* [*simp*]: $compact\ (up \cdot x) = compact\ x$
 ⟨*proof*⟩

instance $u :: (chfin)\ chfin$
 ⟨*proof*⟩

properties of fup

lemma *fup1* [*simp*]: $fup \cdot f \cdot \perp = \perp$
 ⟨*proof*⟩

lemma *fup2* [*simp*]: $fup \cdot f \cdot (up \cdot x) = f \cdot x$
 ⟨*proof*⟩

lemma *fup3* [*simp*]: $fup \cdot up \cdot x = x$
 ⟨*proof*⟩

end

13 Lifting types of class type to flat pcpo's

theory *Lift*
imports *Up*
begin

pcpodef $'a::type\ lift = UNIV :: 'a\ discr\ u\ set$
 ⟨*proof*⟩

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

definition

$Def :: 'a::type \Rightarrow 'a \text{ lift where}$
 $Def\ x = Abs\text{-lift}\ (up.(Discr\ x))$

13.1 Lift as a datatype

lemma *lift-induct*: $\llbracket P\ \perp; \bigwedge x. P\ (Def\ x) \rrbracket \Longrightarrow P\ y$
 $\langle proof \rangle$

old-rep-datatype $\perp :: 'a::type \text{ lift } Def$
 $\langle proof \rangle$

\perp and Def

lemma *not-Undef-is-Def*: $(x \neq \perp) = (\exists y. x = Def\ y)$
 $\langle proof \rangle$

lemma *lift-definedE*: $\llbracket x \neq \perp; \bigwedge a. x = Def\ a \Longrightarrow R \rrbracket \Longrightarrow R$
 $\langle proof \rangle$

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

$\langle ML \rangle$

lemma *DefE*: $Def\ x = \perp \Longrightarrow R$
 $\langle proof \rangle$

lemma *DefE2*: $\llbracket x = Def\ s; x = \perp \rrbracket \Longrightarrow R$
 $\langle proof \rangle$

lemma *Def-below-Def*: $Def\ x \sqsubseteq Def\ y \longleftrightarrow x = y$
 $\langle proof \rangle$

lemma *Def-below-iff [simp]*: $Def\ x \sqsubseteq y \longleftrightarrow Def\ x = y$
 $\langle proof \rangle$

13.2 Lift is flat

instance *lift* :: $(type) \text{ flat}$
 $\langle proof \rangle$

13.3 Continuity of case-lift

lemma *case-lift-eq*: $case\text{-lift}\ \perp\ f\ x = fup.(\bigwedge y. f\ (undiscr\ y)) . (Rep\text{-lift}\ x)$
 $\langle proof \rangle$

lemma *cont2cont-case-lift [simp]*:
 $\llbracket \bigwedge y. cont\ (\lambda x. f\ x\ y); cont\ g \rrbracket \Longrightarrow cont\ (\lambda x. case\text{-lift}\ \perp\ (f\ x)\ (g\ x))$
 $\langle proof \rangle$

13.4 Further operations

definition

$flift1 :: ('a::type \Rightarrow 'b::pcpo) \Rightarrow ('a \text{ lift} \rightarrow 'b)$ (**binder** $\langle FLIFT \rangle 10$) **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

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

lemma $flift1\text{-Def}$ $[simp]$: $flift1 f.(\text{Def } x) = (f x)$
 $\langle proof \rangle$

lemma $flift2\text{-Def}$ $[simp]$: $flift2 f.(\text{Def } x) = \text{Def } (f x)$
 $\langle proof \rangle$

lemma $flift1\text{-strict}$ $[simp]$: $flift1 f.\perp = \perp$
 $\langle proof \rangle$

lemma $flift2\text{-strict}$ $[simp]$: $flift2 f.\perp = \perp$
 $\langle proof \rangle$

lemma $flift2\text{-defined}$ $[simp]$: $x \neq \perp \Longrightarrow (flift2 f).x \neq \perp$
 $\langle proof \rangle$

lemma $flift2\text{-bottom-iff}$ $[simp]$: $(flift2 f.x = \perp) = (x = \perp)$
 $\langle proof \rangle$

lemma $FLIFT\text{-mono}$:
 $(\bigwedge x. f x \sqsubseteq g x) \Longrightarrow (FLIFT x. f x) \sqsubseteq (FLIFT x. g x)$
 $\langle proof \rangle$

lemma $cont2cont\text{-flift1}$ $[simp, cont2cont]$:
 $\llbracket \bigwedge y. cont (\lambda x. f x y) \rrbracket \Longrightarrow cont (\lambda x. FLIFT y. f x y)$
 $\langle proof \rangle$

end

14 The type of lifted booleans

theory *Tr*
imports *Lift*
begin

14.1 Type definition and constructors

type-synonym $tr = bool \text{ lift}$

translations

(type) $tr \leftarrow (type) \text{ bool lift}$

definition $TT :: tr$

where $TT = Def \ True$

definition $FF :: tr$

where $FF = Def \ False$

Exhaustion and Elimination for type tr

lemma *Exh-tr*: $t = \perp \vee t = TT \vee t = FF$

<proof>

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

$\llbracket p = \perp \implies Q; p = TT \implies Q; p = FF \implies Q \rrbracket \implies Q$

<proof>

lemma *tr-induct* [case-names bottom $TT \ FF$, induct type: tr]:

$P \perp \implies P \ TT \implies P \ FF \implies P \ x$

<proof>

distinctness for type tr

lemma *dist-below-tr* [simp]:

$TT \not\sqsubseteq \perp \ FF \not\sqsubseteq \perp \ TT \not\sqsubseteq FF \ FF \not\sqsubseteq TT$

<proof>

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

<proof>

lemma *TT-below-iff* [simp]: $TT \sqsubseteq x \longleftrightarrow x = TT$

<proof>

lemma *FF-below-iff* [simp]: $FF \sqsubseteq x \longleftrightarrow x = FF$

<proof>

lemma *not-below-TT-iff* [simp]: $x \not\sqsubseteq TT \longleftrightarrow x = FF$

<proof>

lemma *not-below-FF-iff* [simp]: $x \not\sqsubseteq FF \longleftrightarrow x = TT$

<proof>

14.2 Case analysis

definition *tr-case* :: $'a::pcpo \rightarrow 'a \rightarrow tr \rightarrow 'a$

where *tr-case* = $(\Lambda \ t \ e \ (Def \ b). \text{ if } b \text{ then } t \text{ else } e)$

abbreviation *cifte-syn* :: $[tr, 'c::pcpo, 'c] \Rightarrow 'c \ (\langle \langle notation = \langle mixfix \text{ If expression } \rangle \rangle \text{ If } (-) / \text{ then } (-) / \text{ else } (-) \rangle [0, 0, 60] \ 60)$

where *If b then e1 else e2* \equiv *tr-case.e1.e2.b*

translations

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

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

lemma *ifte-thms* [simp]:

If \perp then e1 else e2 $= \perp$

If FF then e1 else e2 $= e2$

If TT then e1 else e2 $= e1$

<proof>

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* $:: tr \Rightarrow tr \Rightarrow tr$ ($\langle \text{- andalso -} \rangle$ [36,35] 35)

where *x andalso y* \equiv *trand.x.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* $:: tr \Rightarrow tr \Rightarrow tr$ ($\langle \text{- orelse -} \rangle$ [31,30] 30)

where *x orelse y* \equiv *tror.x.y*

definition *neg* $:: tr \rightarrow tr$

where *neg* $= \text{flift2 Not}$

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

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 orelse-def neg-def tr-case-def TT-def FF-def*

lemmas about andalso, orelse, neg and if

lemma *andalso-thms* [simp]:

(TT andalso y) $= y$

(FF andalso y) $= FF$

(\perp andalso y) $= \perp$

(y andalso TT) $= y$

(y andalso y) $= y$

<proof>

lemma *orelse-thms* [simp]:

(TT orelse y) $= TT$

(FF orelse y) $= y$

(\perp orelse y) $= \perp$

(y orelse FF) $= y$

$(y \text{ orelse } y) = y$
 $\langle \text{proof} \rangle$

lemma *neg-thms* [simp]:

$\text{neg} \cdot TT = FF$
 $\text{neg} \cdot FF = TT$
 $\text{neg} \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

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

lemma *split-If2*: $P \text{ (If2 } Q \ x \ y) \longleftrightarrow ((Q = \perp \longrightarrow P \ \perp) \wedge (Q = TT \longrightarrow P \ x) \wedge (Q = FF \longrightarrow P \ y))$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

14.4 Rewriting of HOLCF operations to HOL functions

lemma *andalso-or*: $t \neq \perp \implies (t \text{ andalso } s) = FF \longleftrightarrow t = FF \vee s = FF$
 $\langle \text{proof} \rangle$

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

lemma *Def-bool1* [simp]: $\text{Def } x \neq FF \longleftrightarrow x$
 $\langle \text{proof} \rangle$

lemma *Def-bool2* [simp]: $\text{Def } x = FF \longleftrightarrow \neg x$
 $\langle \text{proof} \rangle$

lemma *Def-bool3* [simp]: $\text{Def } x = TT \longleftrightarrow x$
 $\langle \text{proof} \rangle$

lemma *Def-bool4* [simp]: $\text{Def } x \neq TT \longleftrightarrow \neg x$
 $\langle \text{proof} \rangle$

lemma *If-and-if*: $(\text{If } \text{Def } P \text{ then } A \text{ else } B) = (\text{if } P \text{ then } A \text{ else } B)$
 $\langle \text{proof} \rangle$

14.5 Compactness

lemma *compact-TT*: $\text{compact } TT$
 $\langle \text{proof} \rangle$

lemma *compact-FF*: $\text{compact } FF$
 $\langle \text{proof} \rangle$

end

15 The type of strict sums

```
theory Ssum
  imports Tr
begin
```

15.1 Definition of strict sum type

definition *ssum* =

$$\{p :: tr \times ('a::pcpo \times 'b::pcpo). p = \perp \vee$$

$$(fst\ p = TT \wedge fst\ (snd\ p) \neq \perp \wedge snd\ (snd\ p) = \perp) \vee$$

$$(fst\ p = FF \wedge fst\ (snd\ p) = \perp \wedge snd\ (snd\ p) \neq \perp)\}$$

pcpodef (*'a::pcpo*, *'b::pcpo*) *ssum* ($\langle \langle notation = \langle infix\ strict\ sum \rangle - \oplus / - \rangle [21,$

20] 20) =

$$ssum :: (tr \times 'a \times 'b)\ set$$

$$\langle proof \rangle$$

instance *ssum* :: ($\{chfin,pcpo\}, \{chfin,pcpo\}$) *chfin*

$$\langle proof \rangle$$

type-notation (*ASCII*)

ssum (**infixr** $\langle ++ \rangle$ 10)

15.2 Definitions of constructors

definition *sinl* :: *'a::pcpo* \rightarrow (*'a* ++ *'b::pcpo*)

where *sinl* = ($\Lambda\ a.\ Abs_ssum\ (seq \cdot a \cdot TT, a, \perp)$)

definition *sinr* :: *'b::pcpo* \rightarrow (*'a::pcpo* ++ *'b*)

where *sinr* = ($\Lambda\ b.\ Abs_ssum\ (seq \cdot b \cdot FF, \perp, b)$)

lemma *sinl-ssum*: ($seq \cdot a \cdot TT, a, \perp$) \in *ssum*

$\langle proof \rangle$

lemma *sinr-ssum*: ($seq \cdot b \cdot FF, \perp, b$) \in *ssum*

$\langle proof \rangle$

lemma *Rep-ssum-sinl*: *Rep-ssum* (*sinl* \cdot *a*) = ($seq \cdot a \cdot TT, a, \perp$)

$\langle proof \rangle$

lemma *Rep-ssum-sinr*: *Rep-ssum* (*sinr* \cdot *b*) = ($seq \cdot b \cdot FF, \perp, b$)

$\langle proof \rangle$

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]: $\text{sinl} \cdot x \sqsubseteq \text{sinl} \cdot y \longleftrightarrow x \sqsubseteq y$
 $\langle \text{proof} \rangle$

lemma *sinr-below* [simp]: $\text{sinr} \cdot x \sqsubseteq \text{sinr} \cdot y \longleftrightarrow x \sqsubseteq y$
 $\langle \text{proof} \rangle$

lemma *sinl-below-sinr* [simp]: $\text{sinl} \cdot x \sqsubseteq \text{sinr} \cdot y \longleftrightarrow x = \perp$
 $\langle \text{proof} \rangle$

lemma *sinr-below-sinl* [simp]: $\text{sinr} \cdot x \sqsubseteq \text{sinl} \cdot y \longleftrightarrow x = \perp$
 $\langle \text{proof} \rangle$

Equality

lemma *sinl-eq* [simp]: $\text{sinl} \cdot x = \text{sinl} \cdot y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *sinr-eq* [simp]: $\text{sinr} \cdot x = \text{sinr} \cdot y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma *sinl-eq-sinr* [simp]: $\text{sinl} \cdot x = \text{sinr} \cdot y \longleftrightarrow x = \perp \wedge y = \perp$
 $\langle \text{proof} \rangle$

lemma *sinr-eq-sinl* [simp]: $\text{sinr} \cdot x = \text{sinl} \cdot y \longleftrightarrow x = \perp \wedge y = \perp$
 $\langle \text{proof} \rangle$

lemma *sinl-inject*: $\text{sinl} \cdot x = \text{sinl} \cdot y \implies x = y$
 $\langle \text{proof} \rangle$

lemma *sinr-inject*: $\text{sinr} \cdot x = \text{sinr} \cdot y \implies x = y$
 $\langle \text{proof} \rangle$

Strictness

lemma *sinl-strict* [simp]: $\text{sinl} \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *sinr-strict* [simp]: $\text{sinr} \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *sinl-bottom-iff* [simp]: $\text{sinl} \cdot x = \perp \longleftrightarrow x = \perp$
 $\langle \text{proof} \rangle$

lemma *sinr-bottom-iff* [simp]: $\text{sinr} \cdot x = \perp \longleftrightarrow x = \perp$
 $\langle \text{proof} \rangle$

lemma *sinl-defined*: $x \neq \perp \implies \text{sinl} \cdot x \neq \perp$
 $\langle \text{proof} \rangle$

lemma *sinr-defined*: $x \neq \perp \implies \text{sinr} \cdot x \neq \perp$
 $\langle \text{proof} \rangle$

Compactness

lemma *compact-sinl*: $\text{compact } x \implies \text{compact } (\text{sinl} \cdot x)$
 $\langle \text{proof} \rangle$

lemma *compact-sinr*: $\text{compact } x \implies \text{compact } (\text{sinr} \cdot x)$
 $\langle \text{proof} \rangle$

lemma *compact-sinlD*: $\text{compact } (\text{sinl} \cdot x) \implies \text{compact } x$
 $\langle \text{proof} \rangle$

lemma *compact-sinrD*: $\text{compact } (\text{sinr} \cdot x) \implies \text{compact } x$
 $\langle \text{proof} \rangle$

lemma *compact-sinl-iff* [simp]: $\text{compact } (\text{sinl} \cdot x) = \text{compact } x$
 $\langle \text{proof} \rangle$

lemma *compact-sinr-iff* [simp]: $\text{compact } (\text{sinr} \cdot x) = \text{compact } x$
 $\langle \text{proof} \rangle$

15.4 Case analysis

lemma *ssumE* [case-names bottom sinl sinr, cases type: ssum]:
obtains $p = \perp$
 $| x \text{ where } p = \text{sinl} \cdot x \text{ and } x \neq \perp$
 $| y \text{ where } p = \text{sinr} \cdot y \text{ and } y \neq \perp$
 $\langle \text{proof} \rangle$

lemma *ssum-induct* [case-names bottom sinl sinr, induct type: ssum]:
 $\llbracket P \perp;$
 $\bigwedge x. x \neq \perp \implies P (\text{sinl} \cdot x);$
 $\bigwedge y. y \neq \perp \implies P (\text{sinr} \cdot y) \rrbracket \implies P x$
 $\langle \text{proof} \rangle$

lemma *ssumE2* [case-names sinl sinr]:
 $\llbracket \bigwedge x. p = \text{sinl} \cdot x \implies Q; \bigwedge y. p = \text{sinr} \cdot y \implies Q \rrbracket \implies Q$
 $\langle \text{proof} \rangle$

lemma *below-sinlD*: $p \sqsubseteq \text{sinl} \cdot x \implies \exists y. p = \text{sinl} \cdot y \wedge y \sqsubseteq x$
 $\langle \text{proof} \rangle$

lemma *below-sinrD*: $p \sqsubseteq \text{sinr} \cdot x \implies \exists y. p = \text{sinr} \cdot y \wedge y \sqsubseteq x$
 $\langle \text{proof} \rangle$

15.5 Case analysis combinator

definition *sscase* :: $('a::\text{pcpo} \rightarrow 'c::\text{pcpo}) \rightarrow ('b::\text{pcpo} \rightarrow 'c) \rightarrow ('a ++ 'b) \rightarrow 'c$

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 beta-sscase : $sscase \cdot f \cdot g \cdot s = (\lambda(t, x, y). \text{If } t \text{ then } f \cdot x \text{ else } g \cdot y) (\text{Rep-ssum } s)$
 $\langle \text{proof} \rangle$

lemma sscase1 $[\text{simp}]$: $sscase \cdot f \cdot g \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma sscase2 $[\text{simp}]$: $x \neq \perp \implies \text{sscase} \cdot f \cdot g \cdot (\text{sinl} \cdot x) = f \cdot x$
 $\langle \text{proof} \rangle$

lemma sscase3 $[\text{simp}]$: $y \neq \perp \implies \text{sscase} \cdot f \cdot g \cdot (\text{sinr} \cdot y) = g \cdot y$
 $\langle \text{proof} \rangle$

lemma sscase4 $[\text{simp}]$: $sscase \cdot \text{sinl} \cdot \text{sinr} \cdot z = z$
 $\langle \text{proof} \rangle$

15.6 Strict sum preserves flatness

instance $\text{ssum} :: (\text{flat}, \text{flat}) \text{ flat}$
 $\langle \text{proof} \rangle$

end

16 The Strict Function Type

theory Sfun
imports Cfun
begin

pcpodef $('a :: \text{pcpo}, 'b :: \text{pcpo}) \text{ sfun } (\text{infixr } \langle \rightarrow ! \rangle 0) = \{ f :: 'a \rightarrow 'b. f \cdot \perp = \perp \}$
 $\langle \text{proof} \rangle$

type-notation (ASCII)
 $\text{sfun } (\text{infixr } \langle \rightarrow ! \rangle 0)$

TODO: Define nice syntax for abstraction, application.

definition $\text{sfun-abs} :: ('a :: \text{pcpo} \rightarrow 'b :: \text{pcpo}) \rightarrow ('a \rightarrow ! 'b)$
where $\text{sfun-abs} = (\Lambda f. \text{Abs-sfun } (\text{strictify} \cdot f))$

definition $\text{sfun-rep} :: ('a::\text{pcpo} \rightarrow! 'b::\text{pcpo}) \rightarrow 'a \rightarrow 'b$
where $\text{sfun-rep} = (\lambda f. \text{Rep-sfun } f)$

lemma sfun-rep-beta : $\text{sfun-rep} \cdot f = \text{Rep-sfun } f$
 $\langle \text{proof} \rangle$

lemma sfun-rep-strict1 $[\text{simp}]$: $\text{sfun-rep} \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma sfun-rep-strict2 $[\text{simp}]$: $\text{sfun-rep} \cdot f \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma strictify-cancel : $f \cdot \perp = \perp \implies \text{strictify} \cdot f = f$
 $\langle \text{proof} \rangle$

lemma sfun-abs-sfun-rep $[\text{simp}]$: $\text{sfun-abs} \cdot (\text{sfun-rep} \cdot f) = f$
 $\langle \text{proof} \rangle$

lemma sfun-rep-sfun-abs $[\text{simp}]$: $\text{sfun-rep} \cdot (\text{sfun-abs} \cdot f) = \text{strictify} \cdot f$
 $\langle \text{proof} \rangle$

lemma sfun-eq-iff : $f = g \iff \text{sfun-rep} \cdot f = \text{sfun-rep} \cdot g$
 $\langle \text{proof} \rangle$

lemma sfun-below-iff : $f \sqsubseteq g \iff \text{sfun-rep} \cdot f \sqsubseteq \text{sfun-rep} \cdot g$
 $\langle \text{proof} \rangle$

end

17 Map functions for various types

theory *Map-Functions*

imports *Deflation Sprod Ssum Sfun Up*

begin

17.1 Map operator for continuous function space

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

lemma cfun-map-beta $[\text{simp}]$: $\text{cfun-map} \cdot a \cdot b \cdot f \cdot x = b \cdot (f \cdot (a \cdot x))$
 $\langle \text{proof} \rangle$

lemma cfun-map-ID : $\text{cfun-map} \cdot \text{ID} \cdot \text{ID} = \text{ID}$
 $\langle \text{proof} \rangle$

lemma cfun-map-map : $\text{cfun-map} \cdot f1 \cdot g1 \cdot (\text{cfun-map} \cdot f2 \cdot g2 \cdot p) = \text{cfun-map} \cdot (\lambda x. f2 \cdot (f1 \cdot x)) \cdot (\lambda x. g1 \cdot (g2 \cdot x)) \cdot p$

<proof>

lemma *ep-pair-cfun-map*:

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

shows *ep-pair (cfun-map.p1.e2) (cfun-map.e1.p2)*

<proof>

lemma *deflation-cfun-map*:

assumes *deflation d1 and deflation d2*

shows *deflation (cfun-map.d1.d2)*

<proof>

lemma *finite-range-cfun-map*:

assumes *a: finite (range (λx. a.x))*

assumes *b: finite (range (λy. b.y))*

shows *finite (range (λf. cfun-map.a.b.f)) (is finite (range ?h))*

<proof>

lemma *finite-deflation-cfun-map*:

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

shows *finite-deflation (cfun-map.d1.d2)*

<proof>

Finite deflations are compact elements of the function space

lemma *finite-deflation-imp-compact*: *finite-deflation d \implies compact d*

<proof>

17.2 Map operator for product type

definition *prod-map* :: $('a \rightarrow 'b) \rightarrow ('c \rightarrow 'd) \rightarrow 'a \times 'c \rightarrow 'b \times 'd$

where *prod-map* = $(\Lambda f g p. (f.(fst p), g.(snd p)))$

lemma *prod-map-Pair [simp]*: *prod-map.f.g.(x, y) = (f.x, g.y)*

<proof>

lemma *prod-map-ID*: *prod-map.ID.ID = ID*

<proof>

lemma *prod-map-map*: *prod-map.f1.g1.(prod-map.f2.g2.p) = prod-map.($\Lambda x. f1.(f2.x)$).($\Lambda x. g1.(g2.x)$).p*

<proof>

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>

lemma *deflation-prod-map*:

assumes *deflation d1 and deflation d2*

shows *deflation* (*prod-map*·*d1*·*d2*)
 $\langle \text{proof} \rangle$

lemma *finite-deflation-prod-map*:
assumes *finite-deflation* *d1* **and** *finite-deflation* *d2*
shows *finite-deflation* (*prod-map*·*d1*·*d2*)
 $\langle \text{proof} \rangle$

17.3 Map function for lifted cpo

definition *u-map* :: (*'a* → *'b*) → *'a* *u* → *'b* *u*
where *u-map* = ($\Lambda f. \text{fup} \cdot (\text{up} \text{ oo } f)$)

lemma *u-map-strict* [*simp*]: *u-map*·*f*· \perp = \perp
 $\langle \text{proof} \rangle$

lemma *u-map-up* [*simp*]: *u-map*·*f*·(*up*·*x*) = *up*·(*f*·*x*)
 $\langle \text{proof} \rangle$

lemma *u-map-ID*: *u-map*·*ID* = *ID*
 $\langle \text{proof} \rangle$

lemma *u-map-map*: *u-map*·*f*·(*u-map*·*g*·*p*) = *u-map*·($\Lambda x. f \cdot (g \cdot x)$)·*p*
 $\langle \text{proof} \rangle$

lemma *u-map-oo*: *u-map*·(*f* oo *g*) = *u-map*·*f* oo *u-map*·*g*
 $\langle \text{proof} \rangle$

lemma *ep-pair-u-map*: *ep-pair* *e* *p* \implies *ep-pair* (*u-map*·*e*) (*u-map*·*p*)
 $\langle \text{proof} \rangle$

lemma *deflation-u-map*: *deflation* *d* \implies *deflation* (*u-map*·*d*)
 $\langle \text{proof} \rangle$

lemma *finite-deflation-u-map*:
assumes *finite-deflation* *d*
shows *finite-deflation* (*u-map*·*d*)
 $\langle \text{proof} \rangle$

17.4 Map function for strict products

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

lemma *sprod-map-strict* [*simp*]: *sprod-map*·*a*·*b*· \perp = \perp
 $\langle \text{proof} \rangle$

lemma *sprod-map-spair* [*simp*]: *x* ≠ $\perp \implies y$ ≠ $\perp \implies$ *sprod-map*·*f*·*g*·(*x*, *y*) =
(:*f*·*x*, *g*·*y*:)

$\langle \text{proof} \rangle$

lemma *sprod-map-spair'*: $f \cdot \perp = \perp \implies g \cdot \perp = \perp \implies \text{sprod-map} \cdot f \cdot g \cdot (:x, y) = (:f \cdot x, g \cdot y)$
 $\langle \text{proof} \rangle$

lemma *sprod-map-ID*: $\text{sprod-map} \cdot \text{ID} \cdot \text{ID} = \text{ID}$
 $\langle \text{proof} \rangle$

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

lemma *ep-pair-sprod-map*:
assumes *ep-pair* $e1$ $p1$ **and** *ep-pair* $e2$ $p2$
shows *ep-pair* $(\text{sprod-map} \cdot e1 \cdot e2)$ $(\text{sprod-map} \cdot p1 \cdot p2)$
 $\langle \text{proof} \rangle$

lemma *deflation-sprod-map*:
assumes *deflation* $d1$ **and** *deflation* $d2$
shows *deflation* $(\text{sprod-map} \cdot d1 \cdot d2)$
 $\langle \text{proof} \rangle$

lemma *finite-deflation-sprod-map*:
assumes *finite-deflation* $d1$ **and** *finite-deflation* $d2$
shows *finite-deflation* $(\text{sprod-map} \cdot d1 \cdot d2)$
 $\langle \text{proof} \rangle$

17.5 Map function for strict sums

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

lemma *ssum-map-strict* [simp]: $\text{ssum-map} \cdot f \cdot g \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *ssum-map-sinl* [simp]: $x \neq \perp \implies \text{ssum-map} \cdot f \cdot g \cdot (\text{sinl} \cdot x) = \text{sinl} \cdot (f \cdot x)$
 $\langle \text{proof} \rangle$

lemma *ssum-map-sinr* [simp]: $x \neq \perp \implies \text{ssum-map} \cdot f \cdot g \cdot (\text{sinr} \cdot x) = \text{sinr} \cdot (g \cdot x)$
 $\langle \text{proof} \rangle$

lemma *ssum-map-sinl'*: $f \cdot \perp = \perp \implies \text{ssum-map} \cdot f \cdot g \cdot (\text{sinl} \cdot x) = \text{sinl} \cdot (f \cdot x)$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *ssum-map-ID*: $\text{ssum-map} \cdot \text{ID} \cdot \text{ID} = \text{ID}$

$\langle \text{proof} \rangle$

lemma *ssum-map-map*:

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

$\langle \text{proof} \rangle$

lemma *ep-pair-ssum-map*:

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

$\langle \text{proof} \rangle$

lemma *deflation-ssum-map*:

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

$\langle \text{proof} \rangle$

lemma *finite-deflation-ssum-map*:

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

$\langle \text{proof} \rangle$

17.6 Map operator for strict function space

definition *sfun-map* :: $(b::\text{pcpo} \rightarrow a::\text{pcpo}) \rightarrow (c::\text{pcpo} \rightarrow d::\text{pcpo}) \rightarrow (a \rightarrow! c) \rightarrow (b \rightarrow! d)$

where $\text{sfun-map} = (\lambda a\ b. \text{sfun-abs} \text{ oo } \text{cfun-map} \cdot a \cdot b \text{ oo } \text{sfun-rep})$

lemma *sfun-map-ID*: $\text{sfun-map} \cdot \text{ID} \cdot \text{ID} = \text{ID}$

$\langle \text{proof} \rangle$

lemma *sfun-map-map*:

assumes $f2 \cdot \perp = \perp$ **and** $g2 \cdot \perp = \perp$
shows $\text{sfun-map} \cdot f1 \cdot g1 \cdot (\text{sfun-map} \cdot f2 \cdot g2 \cdot p) =$
 $\text{sfun-map} \cdot (\lambda x. f2 \cdot (f1 \cdot x)) \cdot (\lambda x. g1 \cdot (g2 \cdot x)) \cdot p$

$\langle \text{proof} \rangle$

lemma *ep-pair-sfun-map*:

assumes $1: \text{ep-pair}\ e1\ p1$
assumes $2: \text{ep-pair}\ e2\ p2$
shows *ep-pair* $(\text{sfun-map} \cdot p1 \cdot e2)\ (\text{sfun-map} \cdot e1 \cdot p2)$

$\langle \text{proof} \rangle$

lemma *deflation-sfun-map*:

assumes $1: \text{deflation}\ d1$

```

assumes 2: deflation d2
shows deflation (sfun-map.d1.d2)
  <proof>

lemma finite-deflation-sfun-map:
  assumes finite-deflation d1
    and finite-deflation d2
  shows finite-deflation (sfun-map.d1.d2)
  <proof>

end

```

18 The cpo of cartesian products

```

theory Cprod
  imports Cfun
begin

```

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
  <proof>

```

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.⊥.⊥
  <proof>

```

```

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

```

$\langle \text{proof} \rangle$

end

19 Profinite and bifinite cpos

theory *Bifinite*

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

19.1 Chains of finite deflations

locale *approx-chain* =

fixes *approx* :: $\text{nat} \Rightarrow 'a \rightarrow 'a$

assumes *chain-approx* [*simp*]: $\text{chain } (\lambda i. \text{approx } i)$

assumes *lub-approx* [*simp*]: $(\bigsqcup i. \text{approx } i) = \text{ID}$

assumes *finite-deflation-approx* [*simp*]: $\bigwedge i. \text{finite-deflation } (\text{approx } i)$

begin

lemma *deflation-approx*: $\text{deflation } (\text{approx } i)$

$\langle \text{proof} \rangle$

lemma *approx-idem*: $\text{approx } i \cdot (\text{approx } i \cdot x) = \text{approx } i \cdot x$

$\langle \text{proof} \rangle$

lemma *approx-below*: $\text{approx } i \cdot x \sqsubseteq x$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *compact-eq-approx*: $\text{compact } x \implies \exists i. \text{approx } i \cdot x = x$

$\langle \text{proof} \rangle$

end

19.2 Omega-profinite and bifinite domains

class *bifinite* = *pcpo* +

assumes *bifinite*: $\exists (a :: \text{nat} \Rightarrow 'a \rightarrow 'a). \text{approx-chain } a$

class *profinite* = *cpo* +

assumes *profinite*: $\exists (a :: \text{nat} \Rightarrow 'a_{\perp} \rightarrow 'a_{\perp}). \text{approx-chain } a$

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* ($\lambda i. f \text{ oo } a \text{ i oo } g$)

<proof>

lemma *approx-chain-u-map*:

assumes *approx-chain a*

shows *approx-chain* ($\lambda i. u\text{-map} \cdot (a \text{ i})$)

<proof>

lemma *approx-chain-sfun-map*:

assumes *approx-chain a* **and** *approx-chain b*

shows *approx-chain* ($\lambda i. sfun\text{-map} \cdot (a \text{ i}) \cdot (b \text{ i})$)

<proof>

lemma *approx-chain-sprod-map*:

assumes *approx-chain a* **and** *approx-chain b*

shows *approx-chain* ($\lambda i. sprod\text{-map} \cdot (a \text{ i}) \cdot (b \text{ i})$)

<proof>

lemma *approx-chain-ssum-map*:

assumes *approx-chain a* **and** *approx-chain b*

shows *approx-chain* ($\lambda i. ssum\text{-map} \cdot (a \text{ i}) \cdot (b \text{ i})$)

<proof>

lemma *approx-chain-cfun-map*:

assumes *approx-chain a* **and** *approx-chain b*

shows *approx-chain* ($\lambda i. cfun\text{-map} \cdot (a \text{ i}) \cdot (b \text{ i})$)

<proof>

lemma *approx-chain-prod-map*:

assumes *approx-chain a* **and** *approx-chain b*

shows *approx-chain* ($\lambda i. prod\text{-map} \cdot (a \text{ i}) \cdot (b \text{ i})$)

<proof>

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*

<proof>

lemma *lub-discr-approx* [*simp*]: ($\bigsqcup i. \text{discr-approx } i$) = *ID*

<proof>

lemma *inj-on-undiscr* [*simp*]: *inj-on undiscr A*

$\langle \text{proof} \rangle$

lemma *finite-deflation-dscr-approx*: *finite-deflation* (*dscr-approx* *i*)
 $\langle \text{proof} \rangle$

lemma *dscr-approx*: *approx-chain* *dscr-approx*
 $\langle \text{proof} \rangle$

19.4 Class instance proofs

instance *bifinite* \subseteq *profinite*
 $\langle \text{proof} \rangle$

instance *u* :: (*profinite*) *bifinite*
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *encode-decode-cfun* [*simp*]: *encode-cfun* · (*decode-cfun* · *y*) = *y*
 $\langle \text{proof} \rangle$

instance *cfun* :: (*profinite*, *bifinite*) *bifinite*
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

lemma *encode-decode-prod-u* [*simp*]: *encode-prod-u* · (*decode-prod-u* · *y*) = *y*
 $\langle \text{proof} \rangle$

instance *prod* :: (*profinite*, *profinite*) *profinite*
 $\langle \text{proof} \rangle$

instance *prod* :: (*bifinite*, *bifinite*) *bifinite*
 $\langle \text{proof} \rangle$

instance *sfun* :: (*bifinite*, *bifinite*) *bifinite*
 $\langle \text{proof} \rangle$

```

instance sprod :: (bifinite, bifinite) bifinite
  ⟨proof⟩

instance ssum :: (bifinite, bifinite) bifinite
  ⟨proof⟩

lemma approx-chain-unit: approx-chain ( $\perp :: \text{nat} \Rightarrow \text{unit} \rightarrow \text{unit}$ )
  ⟨proof⟩

instance unit :: bifinite
  ⟨proof⟩

instance discr :: (countable) profinite
  ⟨proof⟩

instance lift :: (countable) bifinite
  ⟨proof⟩

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  $\prec$  50)
  assumes r-refl:  $x \prec x$ 
  assumes r-trans:  $\llbracket x \prec y; y \prec z \rrbracket \Longrightarrow x \prec z$ 
begin

definition
  ideal :: 'a set  $\Rightarrow$  bool where
  ideal A =  $((\exists x. x \in A) \wedge (\forall x \in A. \forall y \in A. \exists z \in A. x \prec z \wedge y \prec z) \wedge$ 
     $(\forall x y. x \prec y \longrightarrow y \in A \longrightarrow x \in A))$ 

lemma idealI:
  assumes  $\exists x. x \in A$ 
  assumes  $\bigwedge x y. \llbracket x \in A; y \in A \rrbracket \Longrightarrow \exists z \in A. x \prec z \wedge y \prec z$ 
  assumes  $\bigwedge x y. \llbracket x \prec y; y \in A \rrbracket \Longrightarrow x \in A$ 
  shows ideal A
  ⟨proof⟩

lemma idealD1:
  ideal A  $\Longrightarrow \exists x. x \in A$ 

```


$\langle proof \rangle$

lemma *idealD2*:

$\llbracket ideal\ A; x \in A; y \in A \rrbracket \implies \exists z \in A. x \preceq z \wedge y \preceq z$
 $\langle proof \rangle$

lemma *idealD3*:

$\llbracket ideal\ A; x \preceq y; y \in A \rrbracket \implies x \in A$
 $\langle proof \rangle$

lemma *ideal-principal*: *ideal* $\{x. x \preceq z\}$
 $\langle proof \rangle$

lemma *ex-ideal*: $\exists A. A \in \{A. ideal\ A\}$
 $\langle proof \rangle$

The set of ideals is a cpo

lemma *ideal-UN*:

fixes $A :: nat \Rightarrow 'a\ set$
assumes *ideal-A*: $\bigwedge i. ideal\ (A\ i)$
assumes *chain-A*: $\bigwedge i\ j. i \leq j \implies A\ i \subseteq A\ j$
shows *ideal* $(\bigcup i. A\ i)$
 $\langle proof \rangle$

lemma *typedef-ideal-po*:

fixes $Abs :: 'a\ set \Rightarrow 'b::below$
assumes *type*: *type-definition* $Rep\ Abs\ \{S. ideal\ S\}$
assumes *below*: $\bigwedge x\ y. x \sqsubseteq y \longleftrightarrow Rep\ x \subseteq Rep\ y$
shows *OFCLASS* $('b, po-class)$
 $\langle proof \rangle$

lemma

fixes $Abs :: 'a\ set \Rightarrow 'b::po$
assumes *type*: *type-definition* $Rep\ Abs\ \{S. ideal\ S\}$
assumes *below*: $\bigwedge x\ y. x \sqsubseteq y \longleftrightarrow Rep\ x \subseteq Rep\ y$
assumes *S*: *chain* S
shows *typedef-ideal-lub*: $range\ S <<| Abs\ (\bigcup i. Rep\ (S\ i))$
and *typedef-ideal-rep-lub*: $Rep\ (\bigsqcup i. S\ i) = (\bigcup i. Rep\ (S\ i))$
 $\langle proof \rangle$

lemma *typedef-ideal-cpo*:

fixes $Abs :: 'a\ set \Rightarrow 'b::po$
assumes *type*: *type-definition* $Rep\ Abs\ \{S. ideal\ S\}$
assumes *below*: $\bigwedge x\ y. x \sqsubseteq y \longleftrightarrow Rep\ x \subseteq Rep\ y$
shows *OFCLASS* $('b, cpo-class)$
 $\langle proof \rangle$

end

interpretation *below*: *preorder below* :: 'a::po \Rightarrow 'a \Rightarrow bool
 <proof>

20.2 Lemmas about least upper bounds

lemma *is-ub-the-lub-ex*: $\llbracket \exists u. S <<| u; x \in S \rrbracket \Longrightarrow x \sqsubseteq \text{lub } S$
 <proof>

lemma *is-lub-the-lub-ex*: $\llbracket \exists u. S <<| u; S <| x \rrbracket \Longrightarrow \text{lub } S \sqsubseteq x$
 <proof>

20.3 Locale for ideal completion

hide-const (open) *Filter.principal*

locale *ideal-completion* = *preorder* +
fixes *principal* :: 'a::type \Rightarrow 'b
fixes *rep* :: 'b \Rightarrow 'a::type set
assumes *ideal-rep*: $\bigwedge x. \text{ideal } (\text{rep } x)$
assumes *rep-lub*: $\bigwedge Y. \text{chain } Y \Longrightarrow \text{rep } (\bigsqcup i. Y i) = (\bigcup i. \text{rep } (Y i))$
assumes *rep-principal*: $\bigwedge a. \text{rep } (\text{principal } a) = \{b. b \preceq a\}$
assumes *belowI*: $\bigwedge x y. \text{rep } x \subseteq \text{rep } y \Longrightarrow x \sqsubseteq y$
assumes *countable*: $\exists f::'a \Rightarrow \text{nat}. \text{inj } f$
begin

lemma *rep-mono*: $x \sqsubseteq y \Longrightarrow \text{rep } x \subseteq \text{rep } y$
 <proof>

lemma *below-def*: $x \sqsubseteq y \longleftrightarrow \text{rep } x \subseteq \text{rep } y$
 <proof>

lemma *principal-below-iff-mem-rep*: $\text{principal } a \sqsubseteq x \longleftrightarrow a \in \text{rep } x$
 <proof>

lemma *principal-below-iff [simp]*: $\text{principal } a \sqsubseteq \text{principal } b \longleftrightarrow a \preceq b$
 <proof>

lemma *principal-eq-iff*: $\text{principal } a = \text{principal } b \longleftrightarrow a \preceq b \wedge b \preceq a$
 <proof>

lemma *eq-iff*: $x = y \longleftrightarrow \text{rep } x = \text{rep } y$
 <proof>

lemma *principal-mono*: $a \preceq b \Longrightarrow \text{principal } a \sqsubseteq \text{principal } b$
 <proof>

lemma *ch2ch-principal [simp]*:
 $\forall i. Y i \preceq Y (\text{Suc } i) \Longrightarrow \text{chain } (\lambda i. \text{principal } (Y i))$
 <proof>

20.3.1 Principal ideals approximate all elements

lemma *compact-principal* [*simp*]: *compact* (*principal a*)
 ⟨*proof*⟩

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*⟩

lemma *principal-induct*:
assumes *adm*: *adm P*
assumes $P: \bigwedge a. P\ (principal\ a)$
shows $P\ x$
 ⟨*proof*⟩

lemma *compact-imp-principal*: *compact x* $\implies \exists a. x = principal\ a$
 ⟨*proof*⟩

20.4 Defining functions in terms of basis elements

definition

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

lemma *extension-lemma*:
fixes $f :: 'a::type \Rightarrow 'c$
assumes *f-mono*: $\bigwedge a\ b. a \preceq b \implies f\ a \sqsubseteq f\ b$
shows $\exists u. f\ 'rep\ x <<| u$
 ⟨*proof*⟩

lemma *extension-beta*:
fixes $f :: 'a::type \Rightarrow 'c$
assumes *f-mono*: $\bigwedge a\ b. a \preceq b \implies f\ a \sqsubseteq f\ b$
shows *extension f* $x = lub\ (f\ 'rep\ x)$
 ⟨*proof*⟩

lemma *extension-principal*:
fixes $f :: 'a::type \Rightarrow 'c$
assumes *f-mono*: $\bigwedge a\ b. a \preceq b \implies f\ a \sqsubseteq f\ b$
shows *extension f* $(principal\ a) = f\ a$
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

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* $(\lambda x. \text{extension } (\lambda a. f\ x\ a))$

$\langle \text{proof} \rangle$

end

lemma (**in** *preorder*) *typedef-ideal-completion*:

fixes *Abs* :: $'a\ \text{set} \Rightarrow 'b$

assumes *type*: *type-definition* *Rep* *Abs* $\{S. \text{ideal } S\}$

assumes *below*: $\bigwedge x\ y. x \sqsubseteq y \longleftrightarrow \text{Rep } x \subseteq \text{Rep } y$

assumes *principal*: $\bigwedge a. \text{principal } a = \text{Abs } \{b. b \preceq a\}$

assumes *countable*: $\exists f :: 'a \Rightarrow \text{nat}. \text{inj } f$

shows *ideal-completion* *r* *principal* *Rep*

$\langle \text{proof} \rangle$

end

21 A universal bifinite domain

theory *Universal*

imports *Bifinite Completion HOL-Library.Nat-Bijection*

begin

unbundle *no binomial-syntax*

21.1 Basis for universal domain

21.1.1 Basis datatype

type-synonym *ubasis* = *nat*

definition

node :: $\text{nat} \Rightarrow \text{ubasis} \Rightarrow \text{ubasis set} \Rightarrow \text{ubasis}$

where

node *i* *a* *S* = *Suc* (*prod-encode* (*i*, *prod-encode* (*a*, *set-encode* *S*)))

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

$\langle \text{proof} \rangle$

lemma *node-gt-0* [*simp*]: $0 < \text{node } i\ a\ S$

$\langle \text{proof} \rangle$

lemma *node-inject* [*simp*]:

$\llbracket \text{finite } S; \text{finite } T \rrbracket$

$\implies \text{node } i\ a\ S = \text{node } j\ b\ T \longleftrightarrow i = j \wedge a = b \wedge S = T$

$\langle \text{proof} \rangle$

lemma *node-gt0*: $i < \text{node } i \text{ a } S$
 $\langle \text{proof} \rangle$

lemma *node-gt1*: $a < \text{node } i \text{ a } S$
 $\langle \text{proof} \rangle$

lemma *nat-less-power2*: $n < 2^{\hat{n}}$
 $\langle \text{proof} \rangle$

lemma *node-gt2*: $\llbracket \text{finite } S; b \in S \rrbracket \implies b < \text{node } i \text{ a } S$
 $\langle \text{proof} \rangle$

lemma *eq-prod-encode-pairI*:
 $\llbracket \text{fst } (\text{prod-decode } x) = a; \text{snd } (\text{prod-decode } x) = b \rrbracket \implies x = \text{prod-encode } (a, b)$
 $\langle \text{proof} \rangle$

lemma *node-cases*:
assumes 1: $x = 0 \implies P$
assumes 2: $\bigwedge i \text{ a } S. \llbracket \text{finite } S; x = \text{node } i \text{ a } S \rrbracket \implies P$
shows P
 $\langle \text{proof} \rangle$

lemma *node-induct*:
assumes 1: $P \ 0$
assumes 2: $\bigwedge i \text{ a } S. \llbracket P \ a; \text{finite } S; \forall b \in S. P \ b \rrbracket \implies P \ (\text{node } i \text{ a } S)$
shows $P \ x$
 $\langle \text{proof} \rangle$

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*:

$\text{finite } S \implies \text{ubasis-le } a \ (\text{node } i \text{ a } S)$

| *ubasis-le-upper*:

$\llbracket \text{finite } S; b \in S; \text{ubasis-le } a \ b \rrbracket \implies \text{ubasis-le } (\text{node } i \text{ a } S) \ b$

lemma *ubasis-le-minimal*: $\text{ubasis-le } 0 \ x$
 $\langle \text{proof} \rangle$

interpretation *uodom*: *preorder ubasis-le*
 $\langle \text{proof} \rangle$

21.1.3 Generic take function

function

$ubasis\text{-}until :: (ubasis \Rightarrow bool) \Rightarrow ubasis \Rightarrow ubasis$
where
 $ubasis\text{-}until P\ 0 = 0$
 $| \text{finite } S \Longrightarrow ubasis\text{-}until P (\text{node } i\ a\ S) =$
 $\quad (\text{if } P (\text{node } i\ a\ S) \text{ then node } i\ a\ S \text{ else } ubasis\text{-}until P\ a)$
 $\langle \text{proof} \rangle$

termination $ubasis\text{-}until$
 $\langle \text{proof} \rangle$

lemma $ubasis\text{-}until$: $P\ 0 \Longrightarrow P (ubasis\text{-}until P\ x)$
 $\langle \text{proof} \rangle$

lemma $ubasis\text{-}until'$: $0 < ubasis\text{-}until P\ x \Longrightarrow P (ubasis\text{-}until P\ x)$
 $\langle \text{proof} \rangle$

lemma $ubasis\text{-}until\text{-}same$: $P\ x \Longrightarrow ubasis\text{-}until P\ x = x$
 $\langle \text{proof} \rangle$

lemma $ubasis\text{-}until\text{-}idem$:
 $P\ 0 \Longrightarrow ubasis\text{-}until P (ubasis\text{-}until P\ x) = ubasis\text{-}until P\ x$
 $\langle \text{proof} \rangle$

lemma $ubasis\text{-}until\text{-}0$:
 $\forall x. x \neq 0 \longrightarrow \neg P\ x \Longrightarrow ubasis\text{-}until P\ x = 0$
 $\langle \text{proof} \rangle$

lemma $ubasis\text{-}until\text{-}less$: $ubasis\text{-}le (ubasis\text{-}until P\ x)\ x$
 $\langle \text{proof} \rangle$

lemma $ubasis\text{-}until\text{-}chain$:
assumes PQ : $\bigwedge x. P\ x \Longrightarrow Q\ x$
shows $ubasis\text{-}le (ubasis\text{-}until P\ x) (ubasis\text{-}until Q\ x)$
 $\langle \text{proof} \rangle$

lemma $ubasis\text{-}until\text{-}mono$:
assumes $\bigwedge i\ a\ S\ b. \llbracket \text{finite } S; P (\text{node } i\ a\ S); b \in S; ubasis\text{-}le\ a\ b \rrbracket \Longrightarrow P\ b$
shows $ubasis\text{-}le\ a\ b \Longrightarrow ubasis\text{-}le (ubasis\text{-}until P\ a) (ubasis\text{-}until P\ b)$
 $\langle \text{proof} \rangle$

lemma $finite\text{-}range\text{-}ubasis\text{-}until$:
 $finite\ \{x. P\ x\} \Longrightarrow finite\ (range\ (ubasis\text{-}until P))$
 $\langle \text{proof} \rangle$

21.2 Defining the universal domain by ideal completion

typedef $u\text{dom} = \{S. u\text{dom.ideal } S\}$
 $\langle \text{proof} \rangle$

instantiation *udom* :: *below*
begin

definition

$$x \sqsubseteq y \longleftrightarrow \text{Rep-udom } x \subseteq \text{Rep-udom } y$$

instance $\langle \text{proof} \rangle$
end

instance *udom* :: *po*
 $\langle \text{proof} \rangle$

instance *udom* :: *cpo*
 $\langle \text{proof} \rangle$

definition

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

lemma *ubasis-countable*: $\exists f :: \text{ubasis} \Rightarrow \text{nat}. \text{inj } f$
 $\langle \text{proof} \rangle$

interpretation *udom*:

ideal-completion *ubasis-le* *udom-principal* *Rep-udom*
 $\langle \text{proof} \rangle$

Universal domain is pointed

lemma *udom-minimal*: *udom-principal* 0 \sqsubseteq *x*
 $\langle \text{proof} \rangle$

instance *udom* :: *pcpo*
 $\langle \text{proof} \rangle$

lemma *inst-udom-pcpo*: $\perp = \text{udom-principal } 0$
 $\langle \text{proof} \rangle$

21.3 Compact bases of domains

typedef *'a compact-basis* = {*x*::*'a*::*pcpo*. *compact* *x*}
 $\langle \text{proof} \rangle$

lemma *Rep-compact-basis'* [*simp*]: *compact* (*Rep-compact-basis* *a*)
 $\langle \text{proof} \rangle$

lemma *Abs-compact-basis-inverse'* [*simp*]:
compact *x* \implies *Rep-compact-basis* (*Abs-compact-basis* *x*) = *x*
 $\langle \text{proof} \rangle$

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

begin

definition

compact-le-def:

$(\sqsubseteq) \equiv (\lambda x y. \text{Rep-compact-basis } x \sqsubseteq \text{Rep-compact-basis } y)$

instance $\langle \text{proof} \rangle$

end

instance *compact-basis* :: (pcpo) po

$\langle \text{proof} \rangle$

definition

approximants :: 'a::pcpo \Rightarrow 'a compact-basis set **where**

approximants = $(\lambda x. \{a. \text{Rep-compact-basis } a \sqsubseteq x\})$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \Longrightarrow \exists x \in A. \forall y \in A. x \sqsubseteq y \longrightarrow x = y$

$\langle \text{proof} \rangle$

definition

choose :: 'a compact-basis set \Rightarrow 'a compact-basis

where

choose A = (SOME x. $x \in \{x \in A. \forall y \in A. x \sqsubseteq y \longrightarrow x = y\}$)

lemma *choose-lemma:*

$\llbracket \text{finite } A; A \neq \{\} \rrbracket \Longrightarrow \text{choose } A \in \{x \in A. \forall y \in A. x \sqsubseteq y \longrightarrow x = y\}$

$\langle \text{proof} \rangle$

lemma *maximal-choose*:

$\llbracket \text{finite } A; y \in A; \text{choose } A \sqsubseteq y \rrbracket \implies \text{choose } A = y$
 $\langle \text{proof} \rangle$

lemma *choose-in*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \implies \text{choose } A \in A$
 $\langle \text{proof} \rangle$

function

choose-pos :: 'a compact-basis set \Rightarrow 'a compact-basis \Rightarrow nat
where
choose-pos A x =
 (if finite A \wedge x \in A \wedge x \neq choose A
 then Suc (choose-pos (A - {choose A}) x) else 0)
 $\langle \text{proof} \rangle$

termination *choose-pos*
 $\langle \text{proof} \rangle$

declare *choose-pos.simps* [simp del]

lemma *choose-pos-choose*: finite A \implies choose-pos A (choose A) = 0
 $\langle \text{proof} \rangle$

lemma *inj-on-choose-pos* [OF refl]:
 $\llbracket \text{card } A = n; \text{finite } A \rrbracket \implies \text{inj-on } (\text{choose-pos } A) A$
 $\langle \text{proof} \rangle$

lemma *choose-pos-bounded* [OF refl]:
 $\llbracket \text{card } A = n; \text{finite } A; x \in A \rrbracket \implies \text{choose-pos } A x < n$
 $\langle \text{proof} \rangle$

lemma *choose-pos-lessD*:
 $\llbracket \text{choose-pos } A x < \text{choose-pos } A y; \text{finite } A; x \in A; y \in A \rrbracket \implies x \not\sqsupseteq y$
 $\langle \text{proof} \rangle$

21.4.2 Compact basis take function

primrec

cb-take :: nat \Rightarrow 'a compact-basis \Rightarrow 'a compact-basis **where**
cb-take 0 = (λx . compact-bot)
 $|$ *cb-take* (Suc n) = (λa . Abs-compact-basis (approx n (Rep-compact-basis a)))

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

lemma *cb-take-zero* [simp]: *cb-take* 0 a = compact-bot
 $\langle \text{proof} \rangle$

lemma *Rep-cb-take*:

Rep-compact-basis $(cb\text{-}take\ (Suc\ n)\ a) = approx\ n \cdot (Rep\text{-}compact\text{-}basis\ a)$
 $\langle proof \rangle$

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

lemma *cb-take-covers*: $\exists n. cb\text{-}take\ n\ x = x$
 $\langle proof \rangle$

lemma *cb-take-less*: $cb\text{-}take\ n\ x \sqsubseteq x$
 $\langle proof \rangle$

lemma *cb-take-idem*: $cb\text{-}take\ n\ (cb\text{-}take\ n\ x) = cb\text{-}take\ n\ x$
 $\langle proof \rangle$

lemma *cb-take-mono*: $x \sqsubseteq y \implies cb\text{-}take\ n\ x \sqsubseteq cb\text{-}take\ n\ y$
 $\langle proof \rangle$

lemma *cb-take-chain-le*: $m \leq n \implies cb\text{-}take\ m\ x \sqsubseteq cb\text{-}take\ n\ x$
 $\langle proof \rangle$

lemma *finite-range-cb-take*: $finite\ (range\ (cb\text{-}take\ n))$
 $\langle proof \rangle$

21.4.3 Rank of basis elements

definition

$rank :: 'a\ compact\text{-}basis \Rightarrow nat$

where

$rank\ x = (LEAST\ n. cb\text{-}take\ n\ x = x)$

lemma *compact-approx-rank*: $cb\text{-}take\ (rank\ x)\ x = x$
 $\langle proof \rangle$

lemma *rank-leD*: $rank\ x \leq n \implies cb\text{-}take\ n\ x = x$
 $\langle proof \rangle$

lemma *rank-leI*: $cb\text{-}take\ n\ x = x \implies rank\ x \leq n$
 $\langle proof \rangle$

lemma *rank-le-iff*: $rank\ x \leq n \longleftrightarrow cb\text{-}take\ n\ x = x$
 $\langle proof \rangle$

lemma *rank-compact-bot* [*simp*]: $rank\ compact\text{-}bot = 0$
 $\langle proof \rangle$

lemma *rank-eq-0-iff* [*simp*]: $rank\ x = 0 \longleftrightarrow x = compact\text{-}bot$
 $\langle proof \rangle$

definition

rank-le :: 'a compact-basis \Rightarrow 'a compact-basis set
where
rank-le $x = \{y. \text{rank } y \leq \text{rank } x\}$

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*: $\text{rank } x = \text{rank } y \implies \text{rank-eq } x = \text{rank-eq } y$
 $\langle \text{proof} \rangle$

lemma *rank-lt-cong*: $\text{rank } x = \text{rank } y \implies \text{rank-lt } x = \text{rank-lt } y$
 $\langle \text{proof} \rangle$

lemma *rank-eq-subset*: $\text{rank-eq } x \subseteq \text{rank-le } x$
 $\langle \text{proof} \rangle$

lemma *rank-lt-subset*: $\text{rank-lt } x \subseteq \text{rank-le } x$
 $\langle \text{proof} \rangle$

lemma *finite-rank-le*: $\text{finite } (\text{rank-le } x)$
 $\langle \text{proof} \rangle$

lemma *finite-rank-eq*: $\text{finite } (\text{rank-eq } x)$
 $\langle \text{proof} \rangle$

lemma *finite-rank-lt*: $\text{finite } (\text{rank-lt } x)$
 $\langle \text{proof} \rangle$

lemma *rank-lt-Int-rank-eq*: $\text{rank-lt } x \cap \text{rank-eq } x = \{\}$
 $\langle \text{proof} \rangle$

lemma *rank-lt-Un-rank-eq*: $\text{rank-lt } x \cup \text{rank-eq } x = \text{rank-le } x$
 $\langle \text{proof} \rangle$

21.4.4 Sequencing basis elements

definition
place :: 'a compact-basis \Rightarrow nat
where
place $x = \text{card } (\text{rank-lt } x) + \text{choose-pos } (\text{rank-eq } x) \ x$

lemma *place-bounded*: $\text{place } x < \text{card } (\text{rank-le } x)$

<proof>

lemma *place-ge*: $\text{card } (\text{rank-lt } x) \leq \text{place } x$
<proof>

lemma *place-rank-mono*:
fixes $x \ y :: 'a \text{ compact-basis}$
shows $\text{rank } x < \text{rank } y \implies \text{place } x < \text{place } y$
<proof>

lemma *place-eqD*: $\text{place } x = \text{place } y \implies x = y$
<proof>

lemma *inj-place*: inj place
<proof>

21.4.5 Embedding and projection on basis elements

definition

$\text{sub} :: 'a \text{ compact-basis} \Rightarrow 'a \text{ compact-basis}$

where

$\text{sub } x = (\text{case rank } x \text{ of } 0 \Rightarrow \text{compact-bot} \mid \text{Suc } k \Rightarrow \text{cb-take } k \ x)$

lemma *rank-sub-less*: $x \neq \text{compact-bot} \implies \text{rank } (\text{sub } x) < \text{rank } x$
<proof>

lemma *place-sub-less*: $x \neq \text{compact-bot} \implies \text{place } (\text{sub } x) < \text{place } x$
<proof>

lemma *sub-below*: $\text{sub } x \sqsubseteq x$
<proof>

lemma *rank-less-imp-below-sub*: $\llbracket x \sqsubseteq y; \text{rank } x < \text{rank } y \rrbracket \implies x \sqsubseteq \text{sub } y$
<proof>

function *basis-emb* :: $'a \text{ compact-basis} \Rightarrow \text{ubasis}$
where $\text{basis-emb } x = (\text{if } x = \text{compact-bot} \text{ then } 0 \text{ else}$
 $\text{node } (\text{place } x) (\text{basis-emb } (\text{sub } x))$
 $(\text{basis-emb } ' \{y. \text{place } y < \text{place } x \wedge x \sqsubseteq y\}))$
<proof>

termination *basis-emb*
<proof>

declare *basis-emb.simps* [*simp del*]

lemma *basis-emb-compact-bot* [*simp*]:
 $\text{basis-emb } \text{compact-bot} = 0$
<proof>

lemma *basis-emb-rec*:

$\text{basis-emb } x = \text{node } (\text{place } x) (\text{basis-emb } (\text{sub } x)) (\text{basis-emb } ' \{y. \text{place } y < \text{place } x \wedge x \sqsubseteq y\})$
if $x \neq \text{compact-bot}$
 $\langle \text{proof} \rangle$

lemma *basis-emb-eq-0-iff* [simp]:

$\text{basis-emb } x = 0 \longleftrightarrow x = \text{compact-bot}$
 $\langle \text{proof} \rangle$

lemma *fin1*: $\text{finite } \{y. \text{place } y < \text{place } x \wedge x \sqsubseteq y\}$
 $\langle \text{proof} \rangle$

lemma *fin2*: $\text{finite } (\text{basis-emb } ' \{y. \text{place } y < \text{place } x \wedge x \sqsubseteq y\})$
 $\langle \text{proof} \rangle$

lemma *rank-place-mono*:

$\llbracket \text{place } x < \text{place } y; x \sqsubseteq y \rrbracket \implies \text{rank } x < \text{rank } y$
 $\langle \text{proof} \rangle$

lemma *basis-emb-mono*:

$x \sqsubseteq y \implies \text{ubasis-le } (\text{basis-emb } x) (\text{basis-emb } y)$
 $\langle \text{proof} \rangle$

lemma *inj-basis-emb*: inj basis-emb

$\langle \text{proof} \rangle$

definition

$\text{basis-prj} :: \text{ubasis} \Rightarrow 'a \text{ compact-basis}$

where

$\text{basis-prj } x = \text{inv basis-emb}$
 $(\text{ubasis-until } (\lambda x. x \in \text{range } (\text{basis-emb} :: 'a \text{ compact-basis} \Rightarrow \text{ubasis})) x)$

lemma *basis-prj-basis-emb*: $\bigwedge x. \text{basis-prj } (\text{basis-emb } x) = x$
 $\langle \text{proof} \rangle$

lemma *basis-prj-node*:

$\llbracket \text{finite } S; \text{node } i \text{ a } S \notin \text{range } (\text{basis-emb} :: 'a \text{ compact-basis} \Rightarrow \text{nat}) \rrbracket$
 $\implies \text{basis-prj } (\text{node } i \text{ a } S) = (\text{basis-prj } a :: 'a \text{ compact-basis})$
 $\langle \text{proof} \rangle$

lemma *basis-prj-0*: $\text{basis-prj } 0 = \text{compact-bot}$

$\langle \text{proof} \rangle$

lemma *node-eq-basis-emb-iff*:

$\text{finite } S \implies \text{node } i \text{ a } S = \text{basis-emb } x \longleftrightarrow$
 $x \neq \text{compact-bot} \wedge i = \text{place } x \wedge a = \text{basis-emb } (\text{sub } x) \wedge$
 $S = \text{basis-emb } ' \{y. \text{place } y < \text{place } x \wedge x \sqsubseteq y\}$

<proof>

lemma *basis-prj-mono*: $ubasis-le\ a\ b \implies basis-prj\ a \sqsubseteq basis-prj\ b$
<proof>

lemma *basis-emb-prj-less*: $ubasis-le\ (basis-emb\ (basis-prj\ x))\ x$
<proof>

lemma *ideal-completion*:
ideal-completion below Rep-compact-basis (approximants :: 'a \Rightarrow -)
<proof>

end

interpretation *compact-basis*:
ideal-completion below Rep-compact-basis
approximants :: 'a::bifinite \Rightarrow 'a compact-basis set
<proof>

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

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

definition
udom-emb :: 'a \rightarrow udom
where
udom-emb = compact-basis.extension ($\lambda x.$ udom-principal (basis-emb x))

definition
udom-prj :: udom \rightarrow 'a
where
udom-prj = udom.extension ($\lambda x.$ Rep-compact-basis (basis-prj x))

lemma *udom-emb-principal*:
udom-emb.(Rep-compact-basis x) = udom-principal (basis-emb x)
<proof>

lemma *udom-prj-principal*:
udom-prj.(udom-principal x) = Rep-compact-basis (basis-prj x)
<proof>

lemma *ep-pair-udom*: *ep-pair udom-emb udom-prj*
<proof>

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*)
 ⟨proof⟩

lemma *adm-mem-finite*: $\llbracket \text{cont } f; \text{ finite } S \rrbracket \implies \text{adm } (\lambda x. f\ x \in S)$
 ⟨proof⟩

lemma *udom-approx-principal*:

udom-approx *i* · (*udom-principal* *x*) =
udom-principal (*ubasis-until* ($\lambda y.$ $y \leq i$) *x*)
 ⟨proof⟩

lemma *finite-deflation-udom-approx*: *finite-deflation* (*udom-approx* *i*)
 ⟨proof⟩

interpretation *udom-approx*: *finite-deflation* *udom-approx* *i*
 ⟨proof⟩

lemma *chain-udom-approx* [simp]: *chain* ($\lambda i.$ *udom-approx* *i*)
 ⟨proof⟩

lemma *lub-udom-approx* [simp]: $(\bigsqcup i. \text{udom-approx } i) = ID$
 ⟨proof⟩

lemma *udom-approx* [simp]: *approx-chain* *udom-approx*
 ⟨proof⟩

instance *udom* :: *bifinite*
 ⟨proof⟩

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

unbundle *binomial-syntax*

end

22 Algebraic deflations

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

22.1 Type constructor for finite deflations

```
typedef 'a::bifinite fin-defl = {d::'a → 'a. finite-deflation d}
⟨proof⟩
```

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

```
definition below-fin-defl-def:
  below ≡ λx y. Rep-fin-defl x ⊆ Rep-fin-defl y
```

```
instance ⟨proof⟩
end
```

```
instance fin-defl :: (bifinite) po
⟨proof⟩
```

```
lemma finite-deflation-Rep-fin-defl: finite-deflation (Rep-fin-defl d)
⟨proof⟩
```

```
lemma deflation-Rep-fin-defl: deflation (Rep-fin-defl d)
⟨proof⟩
```

```
interpretation Rep-fin-defl: finite-deflation Rep-fin-defl d
⟨proof⟩
```

```
lemma fin-defl-belowI:
  (⋀x. Rep-fin-defl a·x = x ⟹ Rep-fin-defl b·x = x) ⟹ a ⊆ b
⟨proof⟩
```

```
lemma fin-defl-belowD:
  [a ⊆ b; Rep-fin-defl a·x = x] ⟹ Rep-fin-defl b·x = x
⟨proof⟩
```

```
lemma fin-defl-eqI:
  a = b if (⋀x. Rep-fin-defl a·x = x ⟷ Rep-fin-defl b·x = x)
⟨proof⟩
```

```
lemma Rep-fin-defl-mono: a ⊆ b ⟹ Rep-fin-defl a ⊆ Rep-fin-defl b
⟨proof⟩
```

```
lemma Abs-fin-defl-mono:
  [finite-deflation a; finite-deflation b; a ⊆ b]
  ⟹ Abs-fin-defl a ⊆ Abs-fin-defl b
```


<proof>

lemma (in *finite-deflation*) *compact-belowI*:

$d \sqsubseteq f$ if $\bigwedge x. \text{compact } x \implies d \cdot x = x \implies f \cdot x = x$

<proof>

lemma *compact-Rep-fin-defl [simp]*: *compact (Rep-fin-defl a)*

<proof>

22.2 Defining algebraic deflations by ideal completion

typedef *'a::bifinite defl* = $\{S::'a \text{ fin-defl set. below.ideal } S\}$

<proof>

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

begin

definition $x \sqsubseteq y \iff \text{Rep-defl } x \subseteq \text{Rep-defl } y$

instance *<proof>*

end

instance *defl* :: (*bifinite*) *po*

<proof>

instance *defl* :: (*bifinite*) *cpo*

<proof>

definition *defl-principal* :: *'a::bifinite fin-defl* \Rightarrow *'a defl*

where *defl-principal* *t* = *Abs-defl* $\{u. u \sqsubseteq t\}$

lemma *fin-defl-countable*: $\exists f::'a::bifinite \text{ fin-defl} \Rightarrow \text{nat. inj } f$

<proof>

interpretation *defl*: *ideal-completion below defl-principal Rep-defl*

<proof>

Algebraic deflations are pointed

lemma *defl-minimal*: *defl-principal (Abs-fin-defl \perp)* $\sqsubseteq x$

<proof>

instance *defl* :: (*bifinite*) *pcpo*

<proof>

lemma *inst-defl-pcpo*: $\perp = \text{defl-principal (Abs-fin-defl } \perp)$

<proof>

22.3 Applying algebraic deflations

definition $cast :: 'a::bifinite\ defl \rightarrow 'a \rightarrow 'a$
where $cast = defl.extension\ Rep-fin-defl$

lemma $cast-defl-principal$: $cast \cdot (defl-principal\ a) = Rep-fin-defl\ a$
 $\langle proof \rangle$

lemma $deflation-cast$: $deflation\ (cast \cdot d)$
 $\langle proof \rangle$

lemma $finite-deflation-cast$: $compact\ d \implies finite-deflation\ (cast \cdot d)$
 $\langle proof \rangle$

interpretation $cast$: $deflation\ cast \cdot d$
 $\langle proof \rangle$

declare $cast.idem$ $[simp]$

lemma $compact-cast$ $[simp]$: $compact\ (cast \cdot d)$ **if** $compact\ d$
 $\langle proof \rangle$

lemma $cast-below-cast$: $cast \cdot A \sqsubseteq cast \cdot B \longleftrightarrow A \sqsubseteq B$
 $\langle proof \rangle$

lemma $compact-cast-iff$: $compact\ (cast \cdot d) \longleftrightarrow compact\ d$
 $\langle proof \rangle$

lemma $cast-below-imp-below$: $cast \cdot A \sqsubseteq cast \cdot B \implies A \sqsubseteq B$
 $\langle proof \rangle$

lemma $cast-eq-imp-eq$: $cast \cdot A = cast \cdot B \implies A = B$
 $\langle proof \rangle$

lemma $cast-strict1$ $[simp]$: $cast \cdot \perp = \perp$
 $\langle proof \rangle$

lemma $cast-strict2$ $[simp]$: $cast \cdot A \cdot \perp = \perp$
 $\langle proof \rangle$

22.4 Deflation combinators

definition
 $defl-fun1\ e\ p\ f =$
 $defl.extension\ (\lambda a.$
 $defl-principal\ (Abs-fin-defl$
 $(e\ oo\ f \cdot (Rep-fin-defl\ a)\ oo\ p)))$

definition
 $defl-fun2\ e\ p\ f =$

```

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

```

lemma *cast-defl-fun1*:

```

  assumes ep: ep-pair e p
  assumes f: ∧a. finite-deflation a ⇒ finite-deflation (f·a)
  shows cast·(defl-fun1 e p f·A) = e oo f·(cast·A) oo p
<proof>

```

lemma *cast-defl-fun2*:

```

  assumes ep: ep-pair e p
  assumes f: ∧a b. finite-deflation a ⇒ finite-deflation b ⇒
    finite-deflation (f·a·b)
  shows cast·(defl-fun2 e p f·A·B) = e oo f·(cast·A)·(cast·B) oo p
<proof>

```

end

23 Representable domains

theory *Representable*

imports *Algebraic Map-Functions HOL-Library.Countable*

begin

23.1 Class of representable domains

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

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

```

class predomain-syn = cpo +
  fixes liftemb :: 'a⊥ → udom⊥
  fixes liftprj :: udom⊥ → 'a⊥
  fixes liftdefl :: 'a itself ⇒ udom u defl

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

syntax -LIFTDEFL :: type ⇒ logic (⟨(1LIFTDEFL/(1'(-)))⟩)
syntax-consts -LIFTDEFL ⇐ liftdefl
translations LIFTDEFL('t) ⇐ CONST liftdefl TYPE('t)

```

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

lemma *cast-liftdefl-of*: $\text{cast} \cdot (\text{liftdefl-of} \cdot t) = \text{u-map} \cdot (\text{cast} \cdot t)$
 $\langle \text{proof} \rangle$

class *domain* = *predomain-syn* + *pcpo* +
 fixes *emb* :: 'a \rightarrow *u*dom
 fixes *prj* :: *u*dom \rightarrow 'a
 fixes *defl* :: 'a *itself* \Rightarrow *u*dom *defl*
 assumes *ep-pair-emb-prj*: *ep-pair emb prj*
 assumes *cast-DEFL*: $\text{cast} \cdot (\text{defl } \text{TYPE('a)}) = \text{emb } \text{oo } \text{prj}$
 assumes *liftemb-eq*: $\text{liftemb} = \text{u-map} \cdot \text{emb}$
 assumes *liftprj-eq*: $\text{liftprj} = \text{u-map} \cdot \text{prj}$
 assumes *liftdefl-eq*: $\text{liftdefl } \text{TYPE('a)} = \text{liftdefl-of} \cdot (\text{defl } \text{TYPE('a)})$

syntax *-DEFL* :: *type* \Rightarrow *logic* ($\langle (1\text{DEFL} / (1'(-))) \rangle$)
syntax-consts *-DEFL* \Leftarrow *defl*
translations *DEFL('t)* \Leftarrow *CONST defl TYPE('t)*

instance *domain* \subseteq *predomain*
 $\langle \text{proof} \rangle$

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

$\langle \text{ML} \rangle$

interpretation *predomain*: *pcpo-ep-pair liftemb liftprj*
 $\langle \text{proof} \rangle$

interpretation *domain*: *pcpo-ep-pair emb prj*
 $\langle \text{proof} \rangle$

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* \rightarrow 'b::*bifinite*) (*p*::'b \rightarrow 'a)
 assumes *cast-t*: $\text{cast} \cdot t = e \text{ oo } p$
 shows $\exists (a::\text{nat} \Rightarrow 'a::\text{pcpo} \rightarrow 'a). \text{approx-chain } a$
 $\langle \text{proof} \rangle$

instance *domain* \subseteq *bifinite*
 $\langle \text{proof} \rangle$

instance *predomain* \subseteq *profinite*
 $\langle \text{proof} \rangle$

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 $\text{ep-pair-u: ep-pair } u\text{-emb } u\text{-prj}$
<proof>

lemma $\text{ep-pair-prod: ep-pair } \text{prod-emb } \text{prod-prj}$
<proof>

lemma $\text{ep-pair-sprod: ep-pair } \text{sprod-emb } \text{sprod-prj}$
<proof>

lemma $\text{ep-pair-ssum: ep-pair } \text{ssum-emb } \text{ssum-prj}$
<proof>

lemma $\text{ep-pair-sfun: ep-pair } \text{sfun-emb } \text{sfun-prj}$
<proof>

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 $\text{sfun-defl} = \text{defl-fun2 sfun-emb sfun-prj sfun-map}$

lemma cast-u-defl :

$\text{cast} \cdot (\text{u-defl} \cdot A) = \text{u-emb} \text{ oo } \text{u-map} \cdot (\text{cast} \cdot A) \text{ oo } \text{u-prj}$
 $\langle \text{proof} \rangle$

lemma cast-prod-defl :

$\text{cast} \cdot (\text{prod-defl} \cdot A \cdot B) =$
 $\text{prod-emb} \text{ oo } \text{prod-map} \cdot (\text{cast} \cdot A) \cdot (\text{cast} \cdot B) \text{ oo } \text{prod-prj}$
 $\langle \text{proof} \rangle$

lemma cast-sprod-defl :

$\text{cast} \cdot (\text{sprod-defl} \cdot A \cdot B) =$
 $\text{sprod-emb} \text{ oo } \text{sprod-map} \cdot (\text{cast} \cdot A) \cdot (\text{cast} \cdot B) \text{ oo } \text{sprod-prj}$
 $\langle \text{proof} \rangle$

lemma cast-ssum-defl :

$\text{cast} \cdot (\text{ssum-defl} \cdot A \cdot B) =$
 $\text{ssum-emb} \text{ oo } \text{ssum-map} \cdot (\text{cast} \cdot A) \cdot (\text{cast} \cdot B) \text{ oo } \text{ssum-prj}$
 $\langle \text{proof} \rangle$

lemma cast-sfun-defl :

$\text{cast} \cdot (\text{sfun-defl} \cdot A \cdot B) =$
 $\text{sfun-emb} \text{ oo } \text{sfun-map} \cdot (\text{cast} \cdot A) \cdot (\text{cast} \cdot B) \text{ oo } \text{sfun-prj}$
 $\langle \text{proof} \rangle$

Special deflation combinator for unpointed types.

definition $\text{u-liftdefl} :: \text{udom } u \text{ defl} \rightarrow \text{udom defl}$

where $\text{u-liftdefl} = \text{defl-fun1 u-emb u-prj ID}$

lemma cast-u-liftdefl :

$\text{cast} \cdot (\text{u-liftdefl} \cdot A) = \text{u-emb} \text{ oo } \text{cast} \cdot A \text{ oo } \text{u-prj}$
 $\langle \text{proof} \rangle$

lemma $\text{u-liftdefl-liftdefl-of}$:

$\text{u-liftdefl} \cdot (\text{liftdefl-of} \cdot A) = \text{u-defl} \cdot A$
 $\langle \text{proof} \rangle$

23.5 Class instance proofs

23.5.1 Universal domain

instantiation $\text{udom} :: \text{domain}$

begin

definition $[\text{simp}]$:

$\text{emb} = (\text{ID} :: \text{udom} \rightarrow \text{udom})$

definition $[\text{simp}]$:

$\text{prj} = (\text{ID} :: \text{udom} \rightarrow \text{udom})$

definition

$$\text{defl } (t::\text{udom } \text{itself}) = (\bigsqcup i. \text{defl-principal } (\text{Abs-fin-defl } (\text{udom-approx } i)))$$
definition

$$(\text{liftemb} :: \text{udom } u \rightarrow \text{udom } u) = u\text{-map} \cdot \text{emb}$$
definition

$$(\text{liftprj} :: \text{udom } u \rightarrow \text{udom } u) = u\text{-map} \cdot \text{prj}$$
definition

$$\text{liftdefl } (t::\text{udom } \text{itself}) = \text{liftdefl-of} \cdot \text{DEFL}(\text{udom})$$

instance $\langle \text{proof} \rangle$

end

23.5.2 Lifted cpo

instantiation $u :: (\text{predomain}) \text{ domain}$

begin

definition

$$\text{emb} = u\text{-emb} \text{ oo } \text{liftemb}$$
definition

$$\text{prj} = \text{liftprj} \text{ oo } u\text{-prj}$$
definition

$$\text{defl } (t::'a \text{ } u \text{ itself}) = u\text{-liftdefl} \cdot \text{LIFTDEFL}('a)$$
definition

$$(\text{liftemb} :: 'a \text{ } u \text{ } u \rightarrow \text{udom } u) = u\text{-map} \cdot \text{emb}$$
definition

$$(\text{liftprj} :: \text{udom } u \rightarrow 'a \text{ } u \text{ } u) = u\text{-map} \cdot \text{prj}$$
definition

$$\text{liftdefl } (t::'a \text{ } u \text{ itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \text{ } u)$$

instance $\langle \text{proof} \rangle$

end

lemma $\text{DEFL-}u$: $\text{DEFL}('a::\text{predomain } u) = u\text{-liftdefl} \cdot \text{LIFTDEFL}('a)$

$\langle \text{proof} \rangle$

23.5.3 Strict function space

instantiation $\text{sfun} :: (\text{domain}, \text{domain}) \text{ domain}$

begin

definition

$emb = sfun-emb \circ sfun-map \cdot prj \cdot emb$

definition

$prj = sfun-map \cdot emb \cdot prj \circ sfun-prj$

definition

$defl \ (t :: ('a \rightarrow! \ 'b) \ itself) = sfun-defl \cdot DEFL('a) \cdot DEFL('b)$

definition

$(liftemb :: ('a \rightarrow! \ 'b) \ u \rightarrow \ udom \ u) = u-map \cdot emb$

definition

$(liftprj :: udom \ u \rightarrow ('a \rightarrow! \ 'b) \ u) = u-map \cdot prj$

definition

$liftdefl \ (t :: ('a \rightarrow! \ 'b) \ itself) = liftdefl-of \cdot DEFL('a \rightarrow! \ 'b)$

instance $\langle proof \rangle$

end

lemma $DEFL-sfun$:

$DEFL('a :: domain \rightarrow! \ 'b :: domain) = sfun-defl \cdot DEFL('a) \cdot DEFL('b)$
 $\langle proof \rangle$

23.5.4 Continuous function space

instantiation $cfun :: (premain, domain) \ domain$

begin

definition

$emb = emb \circ encode-cfun$

definition

$prj = decode-cfun \circ prj$

definition

$defl \ (t :: ('a \rightarrow \ 'b) \ itself) = DEFL('a \ u \rightarrow! \ 'b)$

definition

$(liftemb :: ('a \rightarrow \ 'b) \ u \rightarrow \ udom \ u) = u-map \cdot emb$

definition

$(liftprj :: udom \ u \rightarrow ('a \rightarrow \ 'b) \ u) = u-map \cdot prj$

definition

$\text{liftdefl } (t::('a \rightarrow 'b) \text{ itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \rightarrow 'b)$

instance $\langle \text{proof} \rangle$

end

lemma *DEFL-cfun*:

$\text{DEFL}('a::\text{predomain} \rightarrow 'b::\text{domain}) = \text{DEFL}('a \ u \rightarrow! 'b)$
 $\langle \text{proof} \rangle$

23.5.5 Strict product

instantiation $\text{sprod} :: (\text{domain}, \text{domain}) \text{ domain}$
begin

definition

$\text{emb} = \text{sprod-emb} \text{ oo } \text{sprod-map} \cdot \text{emb} \cdot \text{emb}$

definition

$\text{prj} = \text{sprod-map} \cdot \text{prj} \cdot \text{prj} \text{ oo } \text{sprod-prj}$

definition

$\text{defl } (t::('a \otimes 'b) \text{ itself}) = \text{sprod-defl} \cdot \text{DEFL}('a) \cdot \text{DEFL}('b)$

definition

$(\text{liftemb} :: ('a \otimes 'b) \ u \rightarrow \text{udom } u) = u\text{-map} \cdot \text{emb}$

definition

$(\text{liftprj} :: \text{udom } u \rightarrow ('a \otimes 'b) \ u) = u\text{-map} \cdot \text{prj}$

definition

$\text{liftdefl } (t::('a \otimes 'b) \text{ itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \otimes 'b)$

instance $\langle \text{proof} \rangle$

end

lemma *DEFL-sprod*:

$\text{DEFL}('a::\text{domain} \otimes 'b::\text{domain}) = \text{sprod-defl} \cdot \text{DEFL}('a) \cdot \text{DEFL}('b)$
 $\langle \text{proof} \rangle$

23.5.6 Cartesian product

definition $\text{prod-liftdefl} :: \text{udom } u \text{ defl} \rightarrow \text{udom } u \text{ defl} \rightarrow \text{udom } u \text{ defl}$

where $\text{prod-liftdefl} = \text{defl-fun2 } (u\text{-map} \cdot \text{prod-emb} \text{ oo } \text{decode-prod-u})$
 $(\text{encode-prod-u} \text{ oo } u\text{-map} \cdot \text{prod-prj}) \text{ sprod-map}$

lemma *cast-prod-liftdefl*:

$\text{cast} \cdot (\text{prod-liftdefl} \cdot a \cdot b) =$
 $(u\text{-map} \cdot \text{prod-emb} \text{ oo } \text{decode-prod-u}) \text{ oo } \text{sprod-map} \cdot (\text{cast} \cdot a) \cdot (\text{cast} \cdot b) \text{ oo}$

$(\text{encode-prod-u } \text{oo } \text{u-map-prod-prj})$
 $\langle \text{proof} \rangle$

instantiation $\text{prod} :: (\text{predomain}, \text{predomain}) \text{ predomain}$
begin

definition
 $\text{liftemb} = (\text{u-map-prod-emb } \text{oo } \text{decode-prod-u}) \text{ oo}$
 $(\text{sprod-map-liftemb-liftemb } \text{oo } \text{encode-prod-u})$

definition
 $\text{liftprj} = (\text{decode-prod-u } \text{oo } \text{sprod-map-liftprj-liftprj}) \text{ oo}$
 $(\text{encode-prod-u } \text{oo } \text{u-map-prod-prj})$

definition
 $\text{liftdefl } (t :: ('a \times 'b) \text{ itself}) = \text{prod-liftdefl} \cdot \text{LIFTDEFL}('a) \cdot \text{LIFTDEFL}('b)$

instance $\langle \text{proof} \rangle$

end

instantiation $\text{prod} :: (\text{domain}, \text{domain}) \text{ domain}$
begin

definition
 $\text{emb} = \text{prod-emb } \text{oo } \text{prod-map-emb-emb}$

definition
 $\text{prj} = \text{prod-map-prj-prj } \text{oo } \text{prod-prj}$

definition
 $\text{defl } (t :: ('a \times 'b) \text{ itself}) = \text{prod-defl} \cdot \text{DEFL}('a) \cdot \text{DEFL}('b)$

instance $\langle \text{proof} \rangle$

end

lemma DEFL-prod :
 $\text{DEFL}('a :: \text{domain} \times 'b :: \text{domain}) = \text{prod-defl} \cdot \text{DEFL}('a) \cdot \text{DEFL}('b)$
 $\langle \text{proof} \rangle$

lemma LIFTDEFL-prod :
 $\text{LIFTDEFL}('a :: \text{predomain} \times 'b :: \text{predomain}) =$
 $\text{prod-liftdefl} \cdot \text{LIFTDEFL}('a) \cdot \text{LIFTDEFL}('b)$
 $\langle \text{proof} \rangle$

23.5.7 Unit type

instantiation $\text{unit} :: \text{domain}$

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 $\langle proof \rangle$

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 $\langle proof \rangle$

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

$defl (t :: ('a \oplus 'b) \text{ itself}) = ssum-defl \cdot DEFL('a) \cdot DEFL('b)$

definition

$(liftemb :: ('a \oplus 'b) \rightarrow udom \ u) = u-map \cdot emb$

definition

$(liftprj :: udom \ u \rightarrow ('a \oplus 'b) \ u) = u-map \cdot prj$

definition

$liftdefl (t :: ('a \oplus 'b) \text{ itself}) = liftdefl-of \cdot DEFL('a \oplus 'b)$

instance $\langle proof \rangle$

end

lemma $DEFL-ssum$:

$DEFL('a :: domain \oplus 'b :: domain) = ssum-defl \cdot DEFL('a) \cdot DEFL('b)$
 $\langle proof \rangle$

23.5.10 Lifted HOL type

instantiation $lift :: (countable) \ domain$

begin

definition

$emb = emb \circ (\Lambda \ x. \ Rep-lift \ x)$

definition

$prj = (\Lambda \ y. \ Abs-lift \ y) \circ prj$

definition

$defl (t :: 'a \ lift \text{ itself}) = DEFL('a \ discr \ u)$

definition

$(liftemb :: 'a \ lift \ u \rightarrow udom \ u) = u-map \cdot emb$

definition

$(liftprj :: udom \ u \rightarrow 'a \ lift \ u) = u-map \cdot prj$

definition

$liftdefl (t :: 'a \ lift \text{ itself}) = liftdefl-of \cdot DEFL('a \ lift)$

instance $\langle proof \rangle$

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*: $t = \perp \vee t = ONE$
 $\langle proof \rangle$

lemma *oneE* [*case-names bottom ONE*]: $\llbracket p = \perp \implies Q; p = ONE \implies Q \rrbracket \implies Q$
 $\langle proof \rangle$

lemma *one-induct* [*case-names bottom ONE*]: $P \perp \implies P ONE \implies P x$
 $\langle proof \rangle$

lemma *dist-below-one* [*simp*]: $ONE \not\sqsubseteq \perp$
 $\langle proof \rangle$

lemma *below-ONE* [*simp*]: $x \sqsubseteq ONE$
 $\langle proof \rangle$

lemma *ONE-below-iff* [*simp*]: $ONE \sqsubseteq x \longleftrightarrow x = ONE$
 $\langle proof \rangle$

lemma *ONE-defined* [*simp*]: $ONE \neq \perp$
 $\langle proof \rangle$

lemma *one-neq-iffs* [*simp*]:
 $x \neq ONE \longleftrightarrow x = \perp$
 $ONE \neq x \longleftrightarrow x = \perp$
 $x \neq \perp \longleftrightarrow x = ONE$
 $\perp \neq x \longleftrightarrow x = ONE$
 $\langle proof \rangle$

lemma *compact-ONE*: *compact ONE*

$\langle \text{proof} \rangle$

Case analysis function for type *one*

definition *one-case* :: $'a::\text{pcpo} \rightarrow \text{one} \rightarrow 'a$
where *one-case* = $(\Lambda a x. \text{seq} \cdot x \cdot a)$

translations

case *x* of *XCONST ONE* $\Rightarrow t \Leftrightarrow \text{CONST } \text{one-case} \cdot t \cdot x$
case *x* of *XCONST ONE* :: $'a \Rightarrow t \mapsto \text{CONST } \text{one-case} \cdot t \cdot x$
 $\Lambda (\text{XCONST ONE}). t \Leftrightarrow \text{CONST } \text{one-case} \cdot t$

lemma *one-case1* [*simp*]: $(\text{case } \perp \text{ of } \text{ONE} \Rightarrow t) = \perp$
 $\langle \text{proof} \rangle$

lemma *one-case2* [*simp*]: $(\text{case } \text{ONE} \text{ of } \text{ONE} \Rightarrow t) = t$
 $\langle \text{proof} \rangle$

lemma *one-case3* [*simp*]: $(\text{case } x \text{ of } \text{ONE} \Rightarrow \text{ONE}) = x$
 $\langle \text{proof} \rangle$

end

theory *Fixrec*
imports *Cprod Sprod Ssum Up One Tr Cfun*
keywords *fixrec* :: *thy-defn*
begin

25 Fixed point operator and admissibility

25.1 Iteration

primrec *iterate* :: $\text{nat} \Rightarrow ('a \rightarrow 'a) \rightarrow ('a \rightarrow 'a)$
where
iterate 0 = $(\Lambda F x. x)$
 $| \text{iterate } (\text{Suc } n) = (\Lambda F x. F \cdot (\text{iterate } n \cdot F \cdot x))$

Derive inductive properties of *iterate* from primitive recursion

lemma *iterate-0* [*simp*]: $\text{iterate } 0 \cdot F \cdot x = x$
 $\langle \text{proof} \rangle$

lemma *iterate-Suc* [*simp*]: $\text{iterate } (\text{Suc } n) \cdot F \cdot x = F \cdot (\text{iterate } n \cdot F \cdot x)$
 $\langle \text{proof} \rangle$

declare *iterate.simps* [*simp del*]

lemma *iterate-Suc2*: $\text{iterate } (\text{Suc } n) \cdot F \cdot x = \text{iterate } n \cdot F \cdot (F \cdot x)$
 $\langle \text{proof} \rangle$

lemma *iterate-iterate*: $\text{iterate } m \cdot F \cdot (\text{iterate } n \cdot F \cdot x) = \text{iterate } (m + n) \cdot F \cdot x$
 ⟨proof⟩

The sequence of function iterations is a chain.

lemma *chain-iterate* [simp]: $\text{chain } (\lambda i. \text{iterate } i \cdot F \cdot \perp)$
 ⟨proof⟩

25.2 Least fixed point operator

definition $\text{fix} :: ('a :: \text{pcpo} \Rightarrow 'a) \Rightarrow 'a$
where $\text{fix} = (\lambda F. \bigsqcup i. \text{iterate } i \cdot F \cdot \perp)$

Binder syntax for fix

abbreviation $\text{fix-syn} :: ('a :: \text{pcpo} \Rightarrow 'a) \Rightarrow 'a$ (**binder** $\langle \mu \rangle 10$)
where $\text{fix-syn } (\lambda x. f \ x) \equiv \text{fix} \cdot (\lambda x. f \ x)$

notation (*ASCII*)
 fix-syn (**binder** $\langle \text{FIX} \rangle 10$)

Properties of fix

direct connection between fix and iteration

lemma *fix-def2*: $\text{fix} \cdot F = (\bigsqcup i. \text{iterate } i \cdot F \cdot \perp)$
 ⟨proof⟩

lemma *iterate-below-fix*: $\text{iterate } n \cdot f \cdot \perp \sqsubseteq \text{fix} \cdot f$
 ⟨proof⟩

Kleene’s fixed point theorems for continuous functions in pointed omega cpo’s

lemma *fix-eq*: $\text{fix} \cdot F = F \cdot (\text{fix} \cdot F)$
 ⟨proof⟩

lemma *fix-least-below*: $F \cdot x \sqsubseteq x \implies \text{fix} \cdot F \sqsubseteq x$
 ⟨proof⟩

lemma *fix-least*: $F \cdot x = x \implies \text{fix} \cdot F \sqsubseteq x$
 ⟨proof⟩

lemma *fix-eqI*:
assumes *fixed*: $F \cdot x = x$
and *least*: $\bigwedge z. F \cdot z = z \implies x \sqsubseteq z$
shows $\text{fix} \cdot F = x$
 ⟨proof⟩

lemma *fix-eq2*: $f \equiv \text{fix} \cdot F \implies f = F \cdot f$
 ⟨proof⟩

lemma *fix-eq3*: $f \equiv \text{fix} \cdot F \implies f \cdot x = F \cdot f \cdot x$

$\langle proof \rangle$

lemma *fix-eq4*: $f = \text{fix} \cdot F \implies f = F \cdot f$
 $\langle proof \rangle$

lemma *fix-eq5*: $f = \text{fix} \cdot F \implies f \cdot x = F \cdot f \cdot x$
 $\langle proof \rangle$

strictness of *fix*

lemma *fix-bottom-iff*: $\text{fix} \cdot F = \perp \iff F \cdot \perp = \perp$
 $\langle proof \rangle$

lemma *fix-strict*: $F \cdot \perp = \perp \implies \text{fix} \cdot F = \perp$
 $\langle proof \rangle$

lemma *fix-defined*: $F \cdot \perp \neq \perp \implies \text{fix} \cdot F \neq \perp$
 $\langle proof \rangle$

fix applied to identity and constant functions

lemma *fix-id*: $(\mu \ x. \ x) = \perp$
 $\langle proof \rangle$

lemma *fix-const*: $(\mu \ x. \ c) = c$
 $\langle proof \rangle$

25.3 Fixed point induction

lemma *fix-ind*: $\text{adm } P \implies P \perp \implies (\bigwedge x. P \ x \implies P \ (F \cdot x)) \implies P \ (\text{fix} \cdot F)$
 $\langle proof \rangle$

lemma *cont-fix-ind*: $\text{cont } F \implies \text{adm } P \implies P \perp \implies (\bigwedge x. P \ x \implies P \ (F \ x)) \implies P \ (\text{fix} \cdot (\text{Abs-cfun } F))$
 $\langle proof \rangle$

lemma *def-fix-ind*: $\llbracket f \equiv \text{fix} \cdot F; \text{adm } P; P \perp; \bigwedge x. P \ x \implies P \ (F \cdot x) \rrbracket \implies P \ f$
 $\langle proof \rangle$

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 $P \ (\text{fix} \cdot F)$
 $\langle proof \rangle$

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 $P \ (\text{fix} \cdot F) \ (\text{fix} \cdot G)$

$\langle proof \rangle$

lemma *cont-parallel-fix-ind*:
assumes *cont F and cont G*
assumes *adm ($\lambda x. P (fst\ x) (snd\ x)$)*
assumes *$P \perp \perp$*
assumes *$\bigwedge x\ y. P\ x\ y \implies P\ (F\ x)\ (G\ y)$*
shows *$P\ (fix.(Abs-cfun\ F))\ (fix.(Abs-cfun\ G))$*
 $\langle proof \rangle$

25.4 Fixed-points on product types

Bekic’s Theorem: Simultaneous fixed points over pairs can be written in terms of separate fixed points.

lemma *fix-cprod*:
fixes *$F :: 'a::pcpo \times 'b::pcpo \rightarrow 'a \times 'b$*
shows
 $fix.F =$
 $(\mu\ x. fst\ (F.(x, \mu\ y. snd\ (F.(x, y)))),$
 $\mu\ y. snd\ (F.(\mu\ x. fst\ (F.(x, \mu\ y. snd\ (F.(x, y)))), y)))$
(is $fix.F = (?x, ?y)$)
 $\langle proof \rangle$

26 Package for defining recursive functions in HOLCF

26.1 Pattern-match monad

pcpodef *$'a\ match = UNIV::(one\ ++\ 'a\ u)\ set$*
 $\langle proof \rangle$

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*]:
 $\llbracket p = \perp \implies Q; p = fail \implies Q; \bigwedge x. p = succeed.x \implies Q \rrbracket \implies Q$
 $\langle proof \rangle$

lemma *succeed-defined* [*simp*]: $succeed.x \neq \perp$
 $\langle proof \rangle$

lemma *fail-defined* [*simp*]: $fail \neq \perp$
 $\langle proof \rangle$

lemma *succeed-eq* [simp]: $(\text{succeed} \cdot x = \text{succeed} \cdot y) = (x = y)$
 $\langle \text{proof} \rangle$

lemma *succeed-neq-fail* [simp]:
 $\text{succeed} \cdot x \neq \text{fail} \text{ fail} \neq \text{succeed} \cdot x$
 $\langle \text{proof} \rangle$

26.1.1 Run operator

definition

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

rewrite rules for run

lemma *run-strict* [simp]: $\text{run} \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *run-fail* [simp]: $\text{run} \cdot \text{fail} = \perp$
 $\langle \text{proof} \rangle$

lemma *run-succeed* [simp]: $\text{run} \cdot (\text{succeed} \cdot x) = x$
 $\langle \text{proof} \rangle$

26.1.2 Monad plus operator

definition

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

abbreviation

$\text{mplus-syn} :: ['a \text{ match}, 'a \text{ match}] \Rightarrow 'a \text{ match} \text{ (infixr } \langle +++ \rangle \text{ 65) where}$
 $m1 \ +++ \ m2 == \text{mplus} \cdot m1 \cdot m2$

rewrite rules for mplus

lemma *mplus-strict* [simp]: $\perp \ +++ \ m = \perp$
 $\langle \text{proof} \rangle$

lemma *mplus-fail* [simp]: $\text{fail} \ +++ \ m = m$
 $\langle \text{proof} \rangle$

lemma *mplus-succeed* [simp]: $\text{succeed} \cdot x \ +++ \ m = \text{succeed} \cdot x$
 $\langle \text{proof} \rangle$

lemma *mplus-fail2* [simp]: $m \ +++ \ \text{fail} = m$
 $\langle \text{proof} \rangle$

lemma *mplus-assoc*: $(x \ +++ \ y) \ +++ \ z = x \ +++ \ (y \ +++ \ z)$
 $\langle \text{proof} \rangle$

26.2 Match functions for built-in types

definition

$$\text{match-bottom} :: 'a::\text{pcpo} \rightarrow 'c \text{ match} \rightarrow 'c \text{ match}$$

where

$$\text{match-bottom} = (\Lambda x k. \text{seq} \cdot x \cdot \text{fail})$$

definition

$$\text{match-Pair} :: 'a \times 'b \rightarrow ('a \rightarrow 'b \rightarrow 'c \text{ match}) \rightarrow 'c \text{ match}$$

where

$$\text{match-Pair} = (\Lambda x k. \text{csplit} \cdot k \cdot x)$$

definition

$$\text{match-spair} :: 'a::\text{pcpo} \otimes 'b::\text{pcpo} \rightarrow ('a \rightarrow 'b \rightarrow 'c \text{ match}) \rightarrow 'c::\text{pcpo} \text{ match}$$

where

$$\text{match-spair} = (\Lambda x k. \text{ssplit} \cdot k \cdot x)$$

definition

$$\text{match-sinl} :: 'a::\text{pcpo} \oplus 'b::\text{pcpo} \rightarrow ('a \rightarrow 'c::\text{pcpo} \text{ match}) \rightarrow 'c \text{ match}$$

where

$$\text{match-sinl} = (\Lambda x k. \text{sscase} \cdot k \cdot (\Lambda b. \text{fail}) \cdot x)$$

definition

$$\text{match-sinr} :: 'a::\text{pcpo} \oplus 'b::\text{pcpo} \rightarrow ('b \rightarrow 'c::\text{pcpo} \text{ match}) \rightarrow 'c \text{ match}$$

where

$$\text{match-sinr} = (\Lambda x k. \text{sscase} \cdot (\Lambda a. \text{fail}) \cdot k \cdot x)$$

definition

$$\text{match-up} :: 'a \text{ u} \rightarrow ('a \rightarrow 'c::\text{pcpo} \text{ match}) \rightarrow 'c \text{ match}$$

where

$$\text{match-up} = (\Lambda x k. \text{fup} \cdot k \cdot x)$$

definition

$$\text{match-ONE} :: \text{one} \rightarrow 'c::\text{pcpo} \text{ match} \rightarrow 'c \text{ match}$$

where

$$\text{match-ONE} = (\Lambda \text{ONE} k. k)$$

definition

$$\text{match-TT} :: \text{tr} \rightarrow 'c::\text{pcpo} \text{ match} \rightarrow 'c \text{ match}$$

where

$$\text{match-TT} = (\Lambda x k. \text{If } x \text{ then } k \text{ else fail})$$

definition

$$\text{match-FF} :: \text{tr} \rightarrow 'c::\text{pcpo} \text{ match} \rightarrow 'c \text{ match}$$

where

$$\text{match-FF} = (\Lambda x k. \text{If } x \text{ then fail else } k)$$

lemma *match-bottom-simps* [simp]:

$$\text{match-bottom} \cdot x \cdot k = (\text{if } x = \perp \text{ then } \perp \text{ else fail})$$

(proof)

lemma *match-Pair-simps* [simp]:

$$\text{match-Pair} \cdot (x, y) \cdot k = k \cdot x \cdot y$$

$\langle \text{proof} \rangle$

lemma *match-spair-simps* [simp]:

$$\llbracket x \neq \perp; y \neq \perp \rrbracket \implies \text{match-spair} \cdot (x, y) \cdot k = k \cdot x \cdot y$$

$$\text{match-spair} \cdot \perp \cdot k = \perp$$

$\langle \text{proof} \rangle$

lemma *match-sinl-simps* [simp]:

$$x \neq \perp \implies \text{match-sinl} \cdot (\text{sinl} \cdot x) \cdot k = k \cdot x$$

$$y \neq \perp \implies \text{match-sinl} \cdot (\text{sinr} \cdot y) \cdot k = \text{fail}$$

$$\text{match-sinl} \cdot \perp \cdot k = \perp$$

$\langle \text{proof} \rangle$

lemma *match-sinr-simps* [simp]:

$$x \neq \perp \implies \text{match-sinr} \cdot (\text{sinl} \cdot x) \cdot k = \text{fail}$$

$$y \neq \perp \implies \text{match-sinr} \cdot (\text{sinr} \cdot y) \cdot k = k \cdot y$$

$$\text{match-sinr} \cdot \perp \cdot k = \perp$$

$\langle \text{proof} \rangle$

lemma *match-up-simps* [simp]:

$$\text{match-up} \cdot (\text{up} \cdot x) \cdot k = k \cdot x$$

$$\text{match-up} \cdot \perp \cdot k = \perp$$

$\langle \text{proof} \rangle$

lemma *match-ONE-simps* [simp]:

$$\text{match-ONE} \cdot \text{ONE} \cdot k = k$$

$$\text{match-ONE} \cdot \perp \cdot k = \perp$$

$\langle \text{proof} \rangle$

lemma *match-TT-simps* [simp]:

$$\text{match-TT} \cdot \text{TT} \cdot k = k$$

$$\text{match-TT} \cdot \text{FF} \cdot k = \text{fail}$$

$$\text{match-TT} \cdot \perp \cdot k = \perp$$

$\langle \text{proof} \rangle$

lemma *match-FF-simps* [simp]:

$$\text{match-FF} \cdot \text{FF} \cdot k = k$$

$$\text{match-FF} \cdot \text{TT} \cdot k = \text{fail}$$

$$\text{match-FF} \cdot \perp \cdot k = \perp$$

$\langle \text{proof} \rangle$

26.3 Mutual recursion

The following rules are used to prove unfolding theorems from fixed-point definitions of mutually recursive functions.

lemma *Pair-equalI*: $\llbracket x \equiv \text{fst } p; y \equiv \text{snd } p \rrbracket \implies (x, y) \equiv p$

$\langle proof \rangle$

lemma *Pair-eqD1*: $(x, y) = (x', y') \implies x = x'$
 $\langle proof \rangle$

lemma *Pair-eqD2*: $(x, y) = (x', y') \implies y = y'$
 $\langle proof \rangle$

lemma *def-cont-fix-eq*:
 $\llbracket f \equiv \text{fix} \cdot (\text{Abs-cfun } F); \text{cont } F \rrbracket \implies f = F f$
 $\langle proof \rangle$

lemma *def-cont-fix-ind*:
 $\llbracket f \equiv \text{fix} \cdot (\text{Abs-cfun } F); \text{cont } F; \text{adm } P; P \perp; \bigwedge x. P x \implies P (F x) \rrbracket \implies P f$
 $\langle proof \rangle$

lemma for proving rewrite rules

lemma *ssubst-lhs*: $\llbracket t = s; P s = Q \rrbracket \implies P t = Q$
 $\langle proof \rangle$

26.4 Initializing the fixrec package

$\langle ML \rangle$

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

end

27 Domain package

theory *Domain*
imports *Representable Map-Functions Fixrec*
keywords
lazy unsafe and
domaindef domain :: thy-defn and
domain-isomorphism :: thy-decl
begin

27.1 Continuous isomorphisms

A locale for continuous isomorphisms

locale *iso* =
fixes *abs* :: $'a::pcpo \rightarrow 'b::pcpo$
fixes *rep* :: $'b \rightarrow 'a$
assumes *abs-iso* [*simp*]: $\text{rep} \cdot (\text{abs} \cdot x) = x$
assumes *rep-iso* [*simp*]: $\text{abs} \cdot (\text{rep} \cdot y) = y$
begin

lemma *swap: iso rep abs*
 $\langle proof \rangle$

lemma *abs-below*: $(abs \cdot x \sqsubseteq abs \cdot y) = (x \sqsubseteq y)$
 $\langle proof \rangle$

lemma *rep-below*: $(rep \cdot x \sqsubseteq rep \cdot y) = (x \sqsubseteq y)$
 $\langle proof \rangle$

lemma *abs-eq*: $(abs \cdot x = abs \cdot y) = (x = y)$
 $\langle proof \rangle$

lemma *rep-eq*: $(rep \cdot x = rep \cdot y) = (x = y)$
 $\langle proof \rangle$

lemma *abs-strict*: $abs \cdot \perp = \perp$
 $\langle proof \rangle$

lemma *rep-strict*: $rep \cdot \perp = \perp$
 $\langle proof \rangle$

lemma *abs-defin'*: $abs \cdot x = \perp \implies x = \perp$
 $\langle proof \rangle$

lemma *rep-defin'*: $rep \cdot z = \perp \implies z = \perp$
 $\langle proof \rangle$

lemma *abs-defined*: $z \neq \perp \implies abs \cdot z \neq \perp$
 $\langle proof \rangle$

lemma *rep-defined*: $z \neq \perp \implies rep \cdot z \neq \perp$
 $\langle proof \rangle$

lemma *abs-bottom-iff*: $(abs \cdot x = \perp) = (x = \perp)$
 $\langle proof \rangle$

lemma *rep-bottom-iff*: $(rep \cdot x = \perp) = (x = \perp)$
 $\langle proof \rangle$

lemma *casedist-rule*: $rep \cdot x = \perp \vee P \implies x = \perp \vee P$
 $\langle proof \rangle$

lemma *compact-abs-rev*: $compact (abs \cdot x) \implies compact x$
 $\langle proof \rangle$

lemma *compact-rep-rev*: $compact (rep \cdot x) \implies compact x$
 $\langle proof \rangle$

lemma *compact-abs*: $compact x \implies compact (abs \cdot x)$

$\langle proof \rangle$

lemma *compact-rep*: $compact\ x \implies compact\ (rep\cdot x)$
 $\langle proof \rangle$

lemma *iso-swap*: $(x = abs\cdot y) = (rep\cdot x = y)$
 $\langle proof \rangle$

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 $deflation\ d \implies deflation\ (abs\ oo\ d\ oo\ rep)$
 $\langle proof \rangle$

lemma *deflation-chain-min*:
assumes *chain*: $chain\ d$
assumes *deft*: $\bigwedge n. deflation\ (d\ n)$
shows $d\ m\cdot(d\ n\cdot x) = d\ (min\ m\ n)\cdot x$
 $\langle proof \rangle$

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** $x = y$
 $\langle proof \rangle$

lemma *lub-ID-reach*:
assumes *chain* *t* **and** $(\bigsqcup n. t\ n) = ID$
shows $(\bigsqcup n. t\ n\cdot x) = x$
 $\langle proof \rangle$

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** $P\ x$
 $\langle proof \rangle$

27.3 Finiteness

Let a “decisive” function be a deflation that maps every input to either itself or bottom. Then if a domain’s take functions are all decisive, then all values in the domain are finite.

definition

$decisive :: ('a::pcpo \rightarrow 'a) \Rightarrow bool$

where

$decisive\ d \iff (\forall x. d \cdot x = x \vee d \cdot x = \perp)$

lemma *decisiveI*: $(\bigwedge x. d \cdot x = x \vee d \cdot x = \perp) \implies decisive\ d$
 $\langle proof \rangle$

lemma *decisive-cases*:

assumes *decisive d* **obtains** $d \cdot x = x \mid d \cdot x = \perp$
 $\langle proof \rangle$

lemma *decisive-bottom*: $decisive\ \perp$
 $\langle proof \rangle$

lemma *decisive-ID*: $decisive\ ID$
 $\langle proof \rangle$

lemma *decisive-ssum-map*:

assumes $f: decisive\ f$
assumes $g: decisive\ g$
shows $decisive\ (ssum\text{-}map \cdot f \cdot g)$
 $\langle proof \rangle$

lemma *decisive-sprod-map*:

assumes $f: decisive\ f$
assumes $g: decisive\ g$
shows $decisive\ (sprod\text{-}map \cdot f \cdot g)$
 $\langle proof \rangle$

lemma *decisive-abs-rep*:

fixes $abs\ rep$
assumes $iso: iso\ abs\ rep$
assumes $d: decisive\ d$
shows $decisive\ (abs\ oo\ d\ oo\ rep)$
 $\langle proof \rangle$

lemma *lub-ID-finite*:

assumes $chain: chain\ d$
assumes $lub: (\bigsqcup n. d\ n) = ID$
assumes $decisive: \bigwedge n. decisive\ (d\ n)$
shows $\exists n. d\ n \cdot x = x$
 $\langle proof \rangle$

lemma *lub-ID-finite-take-induct*:

assumes $chain\ d$ **and** $(\bigsqcup n. d\ n) = ID$ **and** $\bigwedge n. decisive\ (d\ n)$
shows $(\bigwedge n. P\ (d\ n \cdot x)) \implies P\ x$
 $\langle proof \rangle$

27.4 Proofs about constructor functions

Lemmas for proving nchotomy rule:

lemma *ex-one-bottom-iff*:

$$(\exists x. P\ x \wedge x \neq \perp) = P\ ONE$$

<proof>

lemma *ex-up-bottom-iff*:

$$(\exists x. P\ x \wedge x \neq \perp) = (\exists x. P\ (up\cdot x))$$

<proof>

lemma *ex-sprod-bottom-iff*:

$$(\exists y. P\ y \wedge y \neq \perp) =$$

$$(\exists x\ y. (P\ (:x, y:) \wedge x \neq \perp) \wedge y \neq \perp)$$

<proof>

lemma *ex-sprod-up-bottom-iff*:

$$(\exists y. P\ y \wedge y \neq \perp) =$$

$$(\exists x\ y. P\ (:up\cdot x, y:) \wedge y \neq \perp)$$

<proof>

lemma *ex-ssum-bottom-iff*:

$$(\exists x. P\ x \wedge x \neq \perp) =$$

$$((\exists x. P\ (sinl\cdot x) \wedge x \neq \perp) \vee$$

$$(\exists x. P\ (sinr\cdot x) \wedge x \neq \perp))$$

<proof>

lemma *exh-start*: $p = \perp \vee (\exists x. p = x \wedge x \neq \perp)$

<proof>

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*: $\llbracket R; R \Longrightarrow P \rrbracket \Longrightarrow P$

<proof>

lemma *exh-casedist1*: $((P \vee Q \Longrightarrow R) \Longrightarrow S) \equiv (\llbracket P \Longrightarrow R; Q \Longrightarrow R \rrbracket \Longrightarrow S)$

<proof>

lemma *exh-casedist2*: $(\exists x. P\ x \Longrightarrow Q) \equiv (\bigwedge x. P\ x \Longrightarrow Q)$

<proof>

lemma *exh-casedist3*: $(P \wedge Q \Longrightarrow R) \equiv (P \Longrightarrow Q \Longrightarrow R)$

<proof>

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
 ⟨proof⟩

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⟩

27.6 Representations of types

lemma *emb-prj*: *emb.((prj.x)::'a::domain) = cast.DEFL('a).x*
 ⟨proof⟩

lemma *emb-prj-emb*:
 fixes $x :: 'a::domain$
 assumes $DEFL('a) \sqsubseteq DEFL('b)$
 shows $emb \cdot (prj \cdot (emb \cdot x) :: 'b::domain) = emb \cdot x$
 $\langle proof \rangle$

lemma *prj-emb-prj*:
 assumes $DEFL('a::domain) \sqsubseteq DEFL('b::domain)$
 shows $prj \cdot (emb \cdot (prj \cdot x :: 'b)) = (prj \cdot x :: 'a)$
 $\langle proof \rangle$

Isomorphism lemmas used internally by the domain package:

lemma *domain-abs-iso*:
 fixes *abs* and *rep*
 assumes $DEFL: DEFL('b::domain) = DEFL('a::domain)$
 assumes *abs-def*: $(abs :: 'a \rightarrow 'b) \equiv prj \circ emb$
 assumes *rep-def*: $(rep :: 'b \rightarrow 'a) \equiv prj \circ emb$
 shows $rep \cdot (abs \cdot x) = x$
 $\langle proof \rangle$

lemma *domain-rep-iso*:
 fixes *abs* and *rep*
 assumes $DEFL: DEFL('b::domain) = DEFL('a::domain)$
 assumes *abs-def*: $(abs :: 'a \rightarrow 'b) \equiv prj \circ emb$
 assumes *rep-def*: $(rep :: 'b \rightarrow 'a) \equiv prj \circ emb$
 shows $abs \cdot (rep \cdot x) = x$
 $\langle proof \rangle$

27.7 Deflations as sets

definition *defl-set* :: $'a::bifinite \text{ defl} \Rightarrow 'a \text{ set}$
 where $defl\text{-set } A = \{x. \text{cast} \cdot A \cdot x = x\}$

lemma *adm-defl-set*: $adm (\lambda x. x \in defl\text{-set } A)$
 $\langle proof \rangle$

lemma *defl-set-bottom*: $\perp \in defl\text{-set } A$
 $\langle proof \rangle$

lemma *defl-set-cast* [*simp*]: $\text{cast} \cdot A \cdot x \in defl\text{-set } A$
 $\langle proof \rangle$

lemma *defl-set-subset-iff*: $defl\text{-set } A \subseteq defl\text{-set } B \longleftrightarrow A \sqsubseteq B$
 $\langle proof \rangle$

27.8 Proving a subtype is representable

Temporarily relax type constraints.

$\langle ML \rangle$

lemma *typedef-domain-class*:
fixes *Rep* :: 'a::pcpo \Rightarrow udom
fixes *Abs* :: udom \Rightarrow 'a::pcpo
fixes *t* :: udom defl
assumes *type*: type-definition *Rep Abs* (defl-set *t*)
assumes *below*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
assumes *emb*: $\text{emb} \equiv (\Lambda x. \text{Rep } x)$
assumes *prj*: $\text{prj} \equiv (\Lambda x. \text{Abs } (\text{cast} \cdot t \cdot x))$
assumes *defl*: $\text{defl} \equiv (\lambda a::'a \text{ itself}. t)$
assumes *liftemb*: $(\text{liftemb} :: 'a \ u \rightarrow \text{udom } u) \equiv u\text{-map} \cdot \text{emb}$
assumes *liftprj*: $(\text{liftprj} :: \text{udom } u \rightarrow 'a \ u) \equiv u\text{-map} \cdot \text{prj}$
assumes *liftdefl*: $(\text{liftdefl} :: 'a \ \text{itself} \Rightarrow -) \equiv (\lambda t. \text{liftdefl-of} \cdot \text{DEFL}('a))$
shows *OFCLASS*('a, domain-class)
 $\langle \text{proof} \rangle$

lemma *typedef-DEFL*:
assumes *defl* $\equiv (\lambda a::'a::pcpo \text{ itself}. t)$
shows *DEFL*('a::pcpo) = *t*
 $\langle \text{proof} \rangle$

Restore original typing constraints.

$\langle ML \rangle$

27.9 Isomorphic deflations

definition *isodefl* :: ('a::domain \rightarrow 'a) \Rightarrow udom defl \Rightarrow bool
where *isodefl* *d t* $\iff \text{cast} \cdot t = \text{emb} \circ d \circ \text{prj}$

definition *isodefl'* :: ('a::predomain \rightarrow 'a) \Rightarrow udom u defl \Rightarrow bool
where *isodefl'* *d t* $\iff \text{cast} \cdot t = \text{liftemb} \circ u\text{-map} \cdot d \circ \text{liftprj}$

lemma *isodeflI*: $(\bigwedge x. \text{cast} \cdot t \cdot x = \text{emb} \cdot (d \cdot (\text{prj} \cdot x))) \implies \text{isodefl } d \ t$
 $\langle \text{proof} \rangle$

lemma *cast-isodefl*: $\text{isodefl } d \ t \implies \text{cast} \cdot t = (\Lambda x. \text{emb} \cdot (d \cdot (\text{prj} \cdot x)))$
 $\langle \text{proof} \rangle$

lemma *isodefl-strict*: $\text{isodefl } d \ t \implies d \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *isodefl-imp-deflation*:
fixes *d* :: 'a::domain \rightarrow 'a
assumes *isodefl* *d t* **shows** *deflation* *d*
 $\langle \text{proof} \rangle$

lemma *isodefl-ID-DEFL*: $\text{isodefl } (\text{ID} :: 'a \rightarrow 'a) \ \text{DEFL}('a::\text{domain})$
 $\langle \text{proof} \rangle$

lemma *isodefl-LIFTDEFL*:

isodefl' (*ID* :: 'a → 'a) *LIFTDEFL*('a::predomain)
 ⟨proof⟩

lemma *isodefl-DEFL-imp-ID*: *isodefl* (*d* :: 'a → 'a) *DEFL*('a::domain) $\implies d = ID$
 ⟨proof⟩

lemma *isodefl-bottom*: *isodefl* $\perp \perp$
 ⟨proof⟩

lemma *adm-isodefl*:

cont *f* \implies *cont* *g* \implies *adm* ($\lambda x. \text{isodefl } (f \ x) \ (g \ x)$)
 ⟨proof⟩

lemma *isodefl-lub*:

assumes *chain* *d* **and** *chain* *t*
assumes $\bigwedge i. \text{isodefl } (d \ i) \ (t \ i)$
shows *isodefl* ($\bigsqcup i. d \ i$) ($\bigsqcup i. t \ i$)
 ⟨proof⟩

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*)
 ⟨proof⟩

lemma *isodefl-abs-rep*:

fixes *abs* **and** *rep* **and** *d*
assumes *DEFL*: *DEFL*('b::domain) = *DEFL*('a::domain)
assumes *abs-def*: (*abs* :: 'a → 'b) \equiv *prj* oo *emb*
assumes *rep-def*: (*rep* :: 'b → 'a) \equiv *prj* oo *emb*
shows *isodefl* *d* *t* \implies *isodefl* (*abs* oo *d* oo *rep*) *t*
 ⟨proof⟩

lemma *isodefl'-liftdefl-of*: *isodefl* *d* *t* \implies *isodefl'* *d* (*liftdefl-of*·*t*)
 ⟨proof⟩

lemma *isodefl-sfun*:

isodefl *d1* *t1* \implies *isodefl* *d2* *t2* \implies
isodefl (*sfun-map*·*d1*·*d2*) (*sfun-defl*·*t1*·*t2*)
 ⟨proof⟩

lemma *isodefl-ssum*:

isodefl *d1* *t1* \implies *isodefl* *d2* *t2* \implies
isodefl (*ssum-map*·*d1*·*d2*) (*ssum-defl*·*t1*·*t2*)
 ⟨proof⟩

lemma *isodefl-sprod*:

$isodefl\ d1\ t1 \implies isodefl\ d2\ t2 \implies$
 $isodefl\ (sprod\text{-}map\cdot d1\cdot d2)\ (sprod\text{-}defl\cdot t1\cdot t2)$
 $\langle proof \rangle$

lemma *isodefl-prod*:
 $isodefl\ d1\ t1 \implies isodefl\ d2\ t2 \implies$
 $isodefl\ (prod\text{-}map\cdot d1\cdot d2)\ (prod\text{-}defl\cdot t1\cdot t2)$
 $\langle proof \rangle$

lemma *isodefl-u*:
 $isodefl\ d\ t \implies isodefl\ (u\text{-}map\cdot d)\ (u\text{-}defl\cdot t)$
 $\langle proof \rangle$

lemma *isodefl-u-liftdefl*:
 $isodefl'\ d\ t \implies isodefl\ (u\text{-}map\cdot d)\ (u\text{-}liftdefl\cdot t)$
 $\langle proof \rangle$

lemma *encode-prod-u-map*:
 $encode\text{-}prod\text{-}u\cdot (u\text{-}map\cdot (prod\text{-}map\cdot f\cdot g)\cdot (decode\text{-}prod\text{-}u\cdot x))$
 $= sprod\text{-}map\cdot (u\text{-}map\cdot f)\cdot (u\text{-}map\cdot g)\cdot x$
 $\langle proof \rangle$

lemma *isodefl-prod-u*:
assumes $isodefl'\ d1\ t1$ **and** $isodefl'\ d2\ t2$
shows $isodefl'\ (prod\text{-}map\cdot d1\cdot d2)\ (prod\text{-}liftdefl\cdot t1\cdot t2)$
 $\langle proof \rangle$

lemma *encode-cfun-map*:
 $encode\text{-}cfun\cdot (cfun\text{-}map\cdot f\cdot g\cdot (decode\text{-}cfun\cdot x))$
 $= sfun\text{-}map\cdot (u\text{-}map\cdot f)\cdot g\cdot x$
 $\langle proof \rangle$

lemma *isodefl-cfun*:
assumes $isodefl\ (u\text{-}map\cdot d1)\ t1$ **and** $isodefl\ d2\ t2$
shows $isodefl\ (cfun\text{-}map\cdot d1\cdot d2)\ (sfun\text{-}defl\cdot t1\cdot t2)$
 $\langle proof \rangle$

27.10 Setting up the domain package

named-theorems *domain-defl-simps* theorems like $DEFL('a\ t) = t\text{-}defl\ \$\ DEFL('a)$
and *domain-isodefl* theorems like $isodefl\ d\ t \implies isodefl\ (foo\text{-}map\ \$\ d)\ (foo\text{-}defl\ \$\ t)$

$\langle ML \rangle$

lemmas [*domain-defl-simps*] =
 $DEFL\text{-}cfun\ DEFL\text{-}sfun\ DEFL\text{-}ssum\ DEFL\text{-}sprod\ DEFL\text{-}prod\ DEFL\text{-}u$
 $liftdefl\text{-}eq\ LIFTDEFL\text{-}prod\ u\text{-}liftdefl\text{-}liftdefl\text{-}of$

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

<ML>

end

28 A compact basis for powerdomains

theory *Compact-Basis*
imports *Universal*
begin

28.1 A compact basis for powerdomains

definition *pd-basis* = $\{S :: 'a :: \text{bifinite compact-basis set. finite } S \wedge S \neq \{\}\}$

typedef *'a :: bifinite pd-basis* = *pd-basis* :: *'a compact-basis set set*
<proof>

lemma *finite-Rep-pd-basis [simp]: finite (Rep-pd-basis u)*
<proof>

lemma *Rep-pd-basis-nonempty [simp]: Rep-pd-basis u $\neq \{\}$*
<proof>

The powerdomain basis type is countable.

lemma *pd-basis-countable: $\exists f :: 'a :: \text{bifinite pd-basis} \Rightarrow \text{nat. inj } f \text{ (is } Ex ?P)$*
<proof>

28.2 Unit and plus constructors

definition
PDUnit :: *'a :: bifinite compact-basis \Rightarrow 'a pd-basis* **where**
PDUnit = $(\lambda x. \text{Abs-pd-basis } \{x\})$

definition
PDPlus :: *'a :: bifinite pd-basis \Rightarrow 'a pd-basis \Rightarrow 'a pd-basis* **where**
PDPlus *t u* = *Abs-pd-basis (Rep-pd-basis t \cup Rep-pd-basis u)*

lemma *Rep-PDUnit:*

Rep-pd-basis (*PDUnit* *x*) = {*x*}
 ⟨*proof*⟩

lemma *Rep-PDPlus*:

Rep-pd-basis (*PDPlus* *u v*) = *Rep-pd-basis* *u* ∪ *Rep-pd-basis* *v*
 ⟨*proof*⟩

lemma *PDUnit-inject* [*simp*]: (*PDUnit* *a* = *PDUnit* *b*) = (*a* = *b*)
 ⟨*proof*⟩

lemma *PDPlus-assoc*: *PDPlus* (*PDPlus* *t u*) *v* = *PDPlus* *t* (*PDPlus* *u v*)
 ⟨*proof*⟩

lemma *PDPlus-commute*: *PDPlus* *t u* = *PDPlus* *u t*
 ⟨*proof*⟩

lemma *PDPlus-absorb*: *PDPlus* *t t* = *t*
 ⟨*proof*⟩

lemma *pd-basis-induct1* [*case-names* *PDUnit PDPlus*]:
 assumes *PDUnit*: $\bigwedge a. P \text{ (PDUnit } a)$
 assumes *PDPlus*: $\bigwedge a t. P t \implies P \text{ (PDPlus (PDUnit } a) t)$
 shows *P x*
 ⟨*proof*⟩

lemma *pd-basis-induct* [*case-names* *PDUnit PDPlus*]:
 assumes *PDUnit*: $\bigwedge a. P \text{ (PDUnit } a)$
 assumes *PDPlus*: $\bigwedge t u. \llbracket P t; P u \rrbracket \implies P \text{ (PDPlus } t u)$
 shows *P x*
 ⟨*proof*⟩

28.3 Fold operator

definition

fold-pd ::
 (*'a*::*bifinite 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-PDUnit*:

assumes *semilattice* *f*
 shows *fold-pd* *g f* (*PDUnit* *x*) = *g x*
 ⟨*proof*⟩

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*⟩

end

29 Upper powerdomain

```
theory UpperPD
imports Compact-Basis
begin
```

29.1 Basis preorder

definition

upper-le :: 'a::bifinite 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]: $t \leq_{\#} t$
 <proof>

lemma *upper-le-trans*: $\llbracket t \leq_{\#} u; u \leq_{\#} v \rrbracket \Longrightarrow t \leq_{\#} v$
 <proof>

interpretation *upper-le*: preorder *upper-le*
 <proof>

lemma *upper-le-minimal* [simp]: *PDUnit compact-bot* $\leq_{\#} t$
 <proof>

lemma *PDUnit-upper-mono*: $x \sqsubseteq y \Longrightarrow \text{PDUnit } x \leq_{\#} \text{PDUnit } y$
 <proof>

lemma *PDPlus-upper-mono*: $\llbracket s \leq_{\#} t; u \leq_{\#} v \rrbracket \Longrightarrow \text{PDPlus } s \ u \leq_{\#} \text{PDPlus } t \ v$
 <proof>

lemma *PDPlus-upper-le*: *PDPlus* $t \ u \leq_{\#} t$
 <proof>

lemma *upper-le-PDUnit-PDUnit-iff* [simp]:
 $(\text{PDUnit } a \leq_{\#} \text{PDUnit } b) = (a \sqsubseteq b)$
 <proof>

lemma *upper-le-PDPlus-PDUnit-iff*:
 $(\text{PDPlus } t \ u \leq_{\#} \text{PDUnit } a) = (t \leq_{\#} \text{PDUnit } a \vee u \leq_{\#} \text{PDUnit } a)$
 <proof>

lemma *upper-le-PDPlus-iff*: $(t \leq_{\#} \text{PDPlus } u \ v) = (t \leq_{\#} u \wedge t \leq_{\#} v)$
 <proof>

lemma *upper-le-induct* [induct set: *upper-le*]:
 assumes *le*: $t \leq_{\#} u$
 assumes 1: $\bigwedge a \ b. a \sqsubseteq b \Longrightarrow P (\text{PDUnit } a) (\text{PDUnit } b)$
 assumes 2: $\bigwedge t \ u \ a. P \ t \ (\text{PDUnit } a) \Longrightarrow P (\text{PDPlus } t \ u) (\text{PDUnit } a)$
 assumes 3: $\bigwedge t \ u \ v. \llbracket P \ t \ u; P \ t \ v \rrbracket \Longrightarrow P \ t \ (\text{PDPlus } u \ v)$
 shows $P \ t \ u$

$\langle proof \rangle$

29.2 Type definition

typedef $'a::bifinite\ upper-pd$ ($\langle \langle notation = \langle postfix\ upper-pd \rangle \rangle '(-)^\# \rangle \rangle =$
 $\{S::'a\ pd-basis\ set.\ upper-le.\ ideal\ S\}$
 $\langle proof \rangle$

instantiation $upper-pd :: (bifinite)\ below$
begin

definition

$x \sqsubseteq y \longleftrightarrow Rep-upper-pd\ x \subseteq Rep-upper-pd\ y$

instance $\langle proof \rangle$
end

instance $upper-pd :: (bifinite)\ po$
 $\langle proof \rangle$

instance $upper-pd :: (bifinite)\ cpo$
 $\langle proof \rangle$

definition

$upper-principal :: 'a::bifinite\ pd-basis \Rightarrow 'a\ upper-pd$ **where**
 $upper-principal\ t = Abs-upper-pd\ \{u.\ u \leq^\# t\}$

interpretation $upper-pd:$

$ideal-completion\ upper-le\ upper-principal\ Rep-upper-pd$
 $\langle proof \rangle$

Upper powerdomain is pointed

lemma $upper-pd-minimal: upper-principal\ (PDUnit\ compact-bot) \sqsubseteq ys$
 $\langle proof \rangle$

instance $upper-pd :: (bifinite)\ pcpo$
 $\langle proof \rangle$

lemma $inst-upper-pd-pcpo: \perp = upper-principal\ (PDUnit\ compact-bot)$
 $\langle proof \rangle$

29.3 Monadic unit and plus

definition

$upper-unit :: 'a::bifinite \rightarrow 'a\ upper-pd$ **where**
 $upper-unit = compact-basis.extension\ (\lambda a.\ upper-principal\ (PDUnit\ a))$

definition

$upper-plus :: 'a::bifinite\ upper-pd \rightarrow 'a\ upper-pd \rightarrow 'a\ upper-pd$ **where**

upper-plus = *upper-pd.extension* ($\lambda t.$ *upper-pd.extension* ($\lambda u.$
upper-principal (*PDPlus* *t u*)))

abbreviation

upper-add :: 'a::bifinite *upper-pd* \Rightarrow 'a *upper-pd* \Rightarrow 'a *upper-pd*
 (infixl $\langle \cup^\# \rangle$ 65) **where**
xs $\cup^\#$ *ys* == *upper-plus*·*xs*·*ys*

syntax

-*upper-pd* :: *args* \Rightarrow *logic* ($\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix upper-pd enumeration} \rangle \rangle \{-\}^\#) \rangle$)

translations

$\{x, xs\}^\# == \{x\}^\# \cup^\# \{xs\}^\#$
 $\{x\}^\# == \text{CONST upper-unit} \cdot x$

lemma *upper-unit-Rep-compact-basis* [simp]:

$\{\text{Rep-compact-basis } a\}^\# = \text{upper-principal } (\text{PDUnit } a)$
 $\langle \text{proof} \rangle$

lemma *upper-plus-principal* [simp]:

$\text{upper-principal } t \cup^\# \text{upper-principal } u = \text{upper-principal } (\text{PDPlus } t \ u)$
 $\langle \text{proof} \rangle$

interpretation *upper-add*: *semilattice upper-add* $\langle \text{proof} \rangle$

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*: $xs \cup^\# ys \sqsubseteq xs$

$\langle \text{proof} \rangle$

lemma *upper-plus-below2*: $xs \cup^\# ys \sqsubseteq ys$

$\langle \text{proof} \rangle$

lemma *upper-plus-greatest*: $\llbracket xs \sqsubseteq ys; xs \sqsubseteq zs \rrbracket \Longrightarrow xs \sqsubseteq ys \cup^\# zs$

$\langle \text{proof} \rangle$

lemma *upper-below-plus-iff* [simp]:

$xs \sqsubseteq ys \cup^\# zs \longleftrightarrow xs \sqsubseteq ys \wedge xs \sqsubseteq zs$
 $\langle proof \rangle$

lemma *upper-plus-below-unit-iff* [simp]:
 $xs \cup^\# ys \sqsubseteq \{z\}^\# \longleftrightarrow xs \sqsubseteq \{z\}^\# \vee ys \sqsubseteq \{z\}^\#$
 $\langle proof \rangle$

lemma *upper-unit-below-iff* [simp]: $\{x\}^\# \sqsubseteq \{y\}^\# \longleftrightarrow x \sqsubseteq y$
 $\langle proof \rangle$

lemmas *upper-pd-below-simps* =
upper-unit-below-iff
upper-below-plus-iff
upper-plus-below-unit-iff

lemma *upper-unit-eq-iff* [simp]: $\{x\}^\# = \{y\}^\# \longleftrightarrow x = y$
 $\langle proof \rangle$

lemma *upper-unit-strict* [simp]: $\{\perp\}^\# = \perp$
 $\langle proof \rangle$

lemma *upper-plus-strict1* [simp]: $\perp \cup^\# ys = \perp$
 $\langle proof \rangle$

lemma *upper-plus-strict2* [simp]: $xs \cup^\# \perp = \perp$
 $\langle proof \rangle$

lemma *upper-unit-bottom-iff* [simp]: $\{x\}^\# = \perp \longleftrightarrow x = \perp$
 $\langle proof \rangle$

lemma *upper-plus-bottom-iff* [simp]:
 $xs \cup^\# ys = \perp \longleftrightarrow xs = \perp \vee ys = \perp$
 $\langle proof \rangle$

lemma *compact-upper-unit*: $compact\ x \implies compact\ \{x\}^\#$
 $\langle proof \rangle$

lemma *compact-upper-unit-iff* [simp]: $compact\ \{x\}^\# \longleftrightarrow compact\ x$
 $\langle proof \rangle$

lemma *compact-upper-plus* [simp]:
 $\llbracket compact\ xs; compact\ ys \rrbracket \implies compact\ (xs \cup^\# ys)$
 $\langle proof \rangle$

29.4 Induction rules

lemma *upper-pd-induct1*:
assumes P : $adm\ P$
assumes *unit*: $\bigwedge x. P\ \{x\}^\#$

assumes *insert*: $\bigwedge x \text{ } ys. \llbracket P \{x\}^\#; P \text{ } ys \rrbracket \implies P (\{x\}^\# \cup^\# ys)$
shows $P (xs::'a::bifinite \text{ } upper\text{-}pd)$
 $\langle proof \rangle$

lemma *upper-pd-induct* [*case-names adm upper-unit upper-plus, induct type: upper-pd*]:

assumes $P: adm \text{ } P$
assumes *unit*: $\bigwedge x. P \{x\}^\#$
assumes *plus*: $\bigwedge xs \text{ } ys. \llbracket P \text{ } xs; P \text{ } ys \rrbracket \implies P (xs \cup^\# ys)$
shows $P (xs::'a::bifinite \text{ } upper\text{-}pd)$
 $\langle proof \rangle$

29.5 Monadic bind

definition

upper-bind-basis ::
 $'a::bifinite \text{ } pd\text{-}basis \Rightarrow ('a \rightarrow 'b \text{ } upper\text{-}pd) \rightarrow 'b::bifinite \text{ } upper\text{-}pd$ **where**
upper-bind-basis = *fold-pd*
 $(\lambda a. \Lambda f. f \cdot (Rep\text{-}compact\text{-}basis \text{ } a))$
 $(\lambda x \text{ } y. \Lambda f. x \cdot f \cup^\# y \cdot f)$

lemma *ACI-upper-bind*:

semilattice $(\lambda x \text{ } y. \Lambda f. x \cdot f \cup^\# y \cdot f)$
 $\langle proof \rangle$

lemma *upper-bind-basis-simps* [*simp*]:

upper-bind-basis (*PDUnit* *a*) =
 $(\Lambda f. f \cdot (Rep\text{-}compact\text{-}basis \text{ } a))$
upper-bind-basis (*PDPlus* *t u*) =
 $(\Lambda f. upper\text{-}bind\text{-}basis \text{ } t \cdot f \cup^\# upper\text{-}bind\text{-}basis \text{ } u \cdot f)$
 $\langle proof \rangle$

lemma *upper-bind-basis-mono*:

$t \leq^\# u \implies upper\text{-}bind\text{-}basis \text{ } t \sqsubseteq upper\text{-}bind\text{-}basis \text{ } u$
 $\langle proof \rangle$

definition

upper-bind :: $'a::bifinite \text{ } upper\text{-}pd \rightarrow ('a \rightarrow 'b \text{ } upper\text{-}pd) \rightarrow 'b::bifinite \text{ } upper\text{-}pd$
where
upper-bind = *upper-pd.extension upper-bind-basis*

syntax

-upper-bind :: [*logic, logic, logic*] $\Rightarrow logic$
 $(\langle (\langle indent=3 \text{ } notation=\langle binder \text{ } upper\text{-}bind \rangle \cup^\# \text{-} \text{ } \text{ } \rangle [0, 0, 10] \text{ } 10)$

translations

$\bigcup^\# x \in xs. e == CONST \text{ } upper\text{-}bind \cdot xs \cdot (\Lambda x. e)$

lemma *upper-bind-principal* [*simp*]:

$upper\text{-}bind.(upper\text{-}principal\ t) = upper\text{-}bind\text{-}basis\ t$
 $\langle proof \rangle$

lemma *upper-bind-unit* [simp]:
 $upper\text{-}bind.\{x\}\#.f = f.x$
 $\langle proof \rangle$

lemma *upper-bind-plus* [simp]:
 $upper\text{-}bind.(xs \cup\# ys).f = upper\text{-}bind.xs.f \cup\# upper\text{-}bind.ys.f$
 $\langle proof \rangle$

lemma *upper-bind-strict* [simp]: $upper\text{-}bind.\perp.f = f.\perp$
 $\langle proof \rangle$

lemma *upper-bind-bind*:
 $upper\text{-}bind.(upper\text{-}bind.xs.f).g = upper\text{-}bind.xs.(\Lambda\ x.\ upper\text{-}bind.(f.x).g)$
 $\langle proof \rangle$

29.6 Map

definition
 $upper\text{-}map :: ('a::bifinite \rightarrow 'b::bifinite) \rightarrow 'a\ upper\text{-}pd \rightarrow 'b\ upper\text{-}pd$ **where**
 $upper\text{-}map = (\Lambda\ f\ xs.\ upper\text{-}bind.xs.(\Lambda\ x.\ \{f.x\}\#))$

lemma *upper-map-unit* [simp]:
 $upper\text{-}map.f.\{x\}\# = \{f.x\}\#$
 $\langle proof \rangle$

lemma *upper-map-plus* [simp]:
 $upper\text{-}map.f.(xs \cup\# ys) = upper\text{-}map.f.xs \cup\# upper\text{-}map.f.ys$
 $\langle proof \rangle$

lemma *upper-map-bottom* [simp]: $upper\text{-}map.f.\perp = \{f.\perp\}\#$
 $\langle proof \rangle$

lemma *upper-map-ident*: $upper\text{-}map.(\Lambda\ x.\ x).xs = xs$
 $\langle proof \rangle$

lemma *upper-map-ID*: $upper\text{-}map.ID = ID$
 $\langle proof \rangle$

lemma *upper-map-map*:
 $upper\text{-}map.f.(upper\text{-}map.g.xs) = upper\text{-}map.(\Lambda\ x.\ f.(g.x)).xs$
 $\langle proof \rangle$

lemma *upper-bind-map*:
 $upper\text{-}bind.(upper\text{-}map.f.xs).g = upper\text{-}bind.xs.(\Lambda\ x.\ g.(f.x))$
 $\langle proof \rangle$

lemma *upper-map-bind*:

$upper-map.f.(upper-bind.xs.g) = upper-bind.xs.(\Lambda x. upper-map.f.(g.x))$
 $\langle proof \rangle$

lemma *ep-pair-upper-map*: $ep-pair\ e\ p \implies ep-pair\ (upper-map.e)\ (upper-map.p)$

$\langle proof \rangle$

lemma *deflation-upper-map*: $deflation\ d \implies deflation\ (upper-map.d)$

$\langle proof \rangle$

lemma *finite-deflation-upper-map*:

assumes *finite-deflation d* **shows** *finite-deflation (upper-map.d)*
 $\langle proof \rangle$

29.7 Upper powerdomain is bifinite

lemma *approx-chain-upper-map*:

assumes *approx-chain a*
shows *approx-chain* $(\lambda i. upper-map.(a\ i))$
 $\langle proof \rangle$

instance *upper-pd* :: *(bifinite) bifinite*

$\langle proof \rangle$

29.8 Join

definition

$upper-join :: 'a :: bifinite\ upper-pd\ upper-pd \rightarrow 'a\ upper-pd$ **where**
 $upper-join = (\Lambda\ xss. upper-bind.xss.(\Lambda\ xs. xs))$

lemma *upper-join-unit* [*simp*]:

$upper-join.\{xs\}^\# = xs$
 $\langle proof \rangle$

lemma *upper-join-plus* [*simp*]:

$upper-join.(xss \cup^\# yss) = upper-join.xss \cup^\# upper-join.yss$
 $\langle proof \rangle$

lemma *upper-join-bottom* [*simp*]: $upper-join.\bot = \bot$

$\langle proof \rangle$

lemma *upper-join-map-unit*:

$upper-join.(upper-map.upper-unit.xs) = xs$
 $\langle proof \rangle$

lemma *upper-join-map-join*:

$upper-join.(upper-map.upper-join.xsss) = upper-join.(upper-join.xsss)$
 $\langle proof \rangle$

lemma *upper-join-map-map*:
 $upper\text{-}join.(upper\text{-}map.(upper\text{-}map.f).xss) =$
 $upper\text{-}map.f.(upper\text{-}join.xss)$
 $\langle proof \rangle$

end

30 Lower powerdomain

theory *LowerPD*
imports *Compact-Basis*
begin

30.1 Basis preorder

definition
 $lower\text{-}le :: 'a::bifinite\ pd\text{-}basis \Rightarrow 'a\ pd\text{-}basis \Rightarrow bool$ (**infix** \leq_b 50) **where**
 $lower\text{-}le = (\lambda u\ v. \forall x \in Rep\text{-}pd\text{-}basis\ u. \exists y \in Rep\text{-}pd\text{-}basis\ v. x \sqsubseteq y)$

lemma *lower-le-refl* [simp]: $t \leq_b t$
 $\langle proof \rangle$

lemma *lower-le-trans*: $\llbracket t \leq_b u; u \leq_b v \rrbracket \Longrightarrow t \leq_b v$
 $\langle proof \rangle$

interpretation *lower-le*: *preorder lower-le*
 $\langle proof \rangle$

lemma *lower-le-minimal* [simp]: $PDUnit\ compact\text{-}bot \leq_b t$
 $\langle proof \rangle$

lemma *PDUnit-lower-mono*: $x \sqsubseteq y \Longrightarrow PDUnit\ x \leq_b PDUnit\ y$
 $\langle proof \rangle$

lemma *PDPlus-lower-mono*: $\llbracket s \leq_b t; u \leq_b v \rrbracket \Longrightarrow PDPlus\ s\ u \leq_b PDPlus\ t\ v$
 $\langle proof \rangle$

lemma *PDPlus-lower-le*: $t \leq_b PDPlus\ t\ u$
 $\langle proof \rangle$

lemma *lower-le-PDUnit-PDUnit-iff* [simp]:
 $(PDUnit\ a \leq_b PDUnit\ b) = (a \sqsubseteq b)$
 $\langle proof \rangle$

lemma *lower-le-PDUnit-PDPlus-iff*:
 $(PDUnit\ a \leq_b PDPlus\ t\ u) = (PDUnit\ a \leq_b t \vee PDUnit\ a \leq_b u)$
 $\langle proof \rangle$

lemma *lower-le-PDPlus-iff*: $(PDPlus\ t\ u \leq_b v) = (t \leq_b v \wedge u \leq_b v)$

⟨proof⟩

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 $P t u$

⟨proof⟩

30.2 Type definition

typedef *'a::bifinite lower-pd* ($\langle \langle notation = \langle postfix lower-pd \rangle \rangle'(-)b \rangle \rangle =$
 $\{S :: 'a \text{ pd-basis set. lower-le.ideal } S\}$

⟨proof⟩

instantiation *lower-pd* :: (bifinite) below

begin

definition

$x \sqsubseteq y \longleftrightarrow Rep\text{-}lower\text{-}pd\ x \subseteq Rep\text{-}lower\text{-}pd\ y$

instance ⟨proof⟩

end

instance *lower-pd* :: (bifinite) po

⟨proof⟩

instance *lower-pd* :: (bifinite) cpo

⟨proof⟩

definition

lower-principal :: *'a::bifinite pd-basis* \Rightarrow *'a lower-pd* **where**

lower-principal $t = Abs\text{-}lower\text{-}pd\ \{u. u \leq b t\}$

interpretation *lower-pd*:

ideal-completion lower-le lower-principal Rep-lower-pd

⟨proof⟩

Lower powerdomain is pointed

lemma *lower-pd-minimal*: *lower-principal* (*PDUnit compact-bot*) $\sqsubseteq ys$

⟨proof⟩

instance *lower-pd* :: (bifinite) pcpo

⟨proof⟩

lemma *inst-lower-pd-pcpo*: $\perp = \text{lower-principal } (PDUnit \text{ compact-bot})$

⟨proof⟩

30.3 Monadic unit and plus

definition

lower-unit :: 'a::bifinite \rightarrow 'a lower-pd **where**
lower-unit = compact-basis.extension ($\lambda a.$ lower-principal (PDUnit a))

definition

lower-plus :: 'a::bifinite lower-pd \rightarrow 'a lower-pd \rightarrow 'a lower-pd **where**
lower-plus = lower-pd.extension ($\lambda t.$ lower-pd.extension ($\lambda u.$
 lower-principal (PDPlus t u)))

abbreviation

lower-add :: 'a::bifinite lower-pd \Rightarrow 'a lower-pd \Rightarrow 'a lower-pd
 (infixl $\langle \cup \rangle$ 65) **where**
xs \cup *ys* == *lower-plus*.*xs*.*ys*

syntax

-lower-pd :: args \Rightarrow logic ($\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix lower-pd enumeration} \rangle \{ \} \rangle) \rangle$)

translations

$\{x, xs\} \flat == \{x\} \flat \cup \{xs\} \flat$
 $\{x\} \flat == \text{CONST lower-unit} \cdot x$

lemma *lower-unit-Rep-compact-basis* [simp]:

$\{\text{Rep-compact-basis } a\} \flat = \text{lower-principal (PDUnit } a)$
 $\langle \text{proof} \rangle$

lemma *lower-plus-principal* [simp]:

$\text{lower-principal } t \cup \text{lower-principal } u = \text{lower-principal (PDPlus } t \ u)$
 $\langle \text{proof} \rangle$

interpretation *lower-add*: semilattice *lower-add* $\langle \text{proof} \rangle$

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*: $xs \sqsubseteq xs \cup ys$

$\langle \text{proof} \rangle$

lemma *lower-plus-below2*: $ys \sqsubseteq xs \cup b \ ys$
 $\langle proof \rangle$

lemma *lower-plus-least*: $\llbracket xs \sqsubseteq zs; ys \sqsubseteq zs \rrbracket \implies xs \cup b \ ys \sqsubseteq zs$
 $\langle proof \rangle$

lemma *lower-plus-below-iff* [simp]:
 $xs \cup b \ ys \sqsubseteq zs \iff xs \sqsubseteq zs \wedge ys \sqsubseteq zs$
 $\langle proof \rangle$

lemma *lower-unit-below-plus-iff* [simp]:
 $\{x\}b \sqsubseteq ys \cup b \ zs \iff \{x\}b \sqsubseteq ys \vee \{x\}b \sqsubseteq zs$
 $\langle proof \rangle$

lemma *lower-unit-below-iff* [simp]: $\{x\}b \sqsubseteq \{y\}b \iff x \sqsubseteq y$
 $\langle proof \rangle$

lemmas *lower-pd-below-simps* =
lower-unit-below-iff
lower-plus-below-iff
lower-unit-below-plus-iff

lemma *lower-unit-eq-iff* [simp]: $\{x\}b = \{y\}b \iff x = y$
 $\langle proof \rangle$

lemma *lower-unit-strict* [simp]: $\{\perp\}b = \perp$
 $\langle proof \rangle$

lemma *lower-unit-bottom-iff* [simp]: $\{x\}b = \perp \iff x = \perp$
 $\langle proof \rangle$

lemma *lower-plus-bottom-iff* [simp]:
 $xs \cup b \ ys = \perp \iff xs = \perp \wedge ys = \perp$
 $\langle proof \rangle$

lemma *lower-plus-strict1* [simp]: $\perp \cup b \ ys = ys$
 $\langle proof \rangle$

lemma *lower-plus-strict2* [simp]: $xs \cup b \ \perp = xs$
 $\langle proof \rangle$

lemma *compact-lower-unit*: $compact \ x \implies compact \ \{x\}b$
 $\langle proof \rangle$

lemma *compact-lower-unit-iff* [simp]: $compact \ \{x\}b \iff compact \ x$
 $\langle proof \rangle$

lemma *compact-lower-plus* [simp]:
 $\llbracket compact \ xs; compact \ ys \rrbracket \implies compact \ (xs \cup b \ ys)$

$\langle \text{proof} \rangle$

30.4 Induction rules

lemma *lower-pd-induct1*:

assumes *P*: *adm P*

assumes *unit*: $\bigwedge x. P \{x\} \flat$

assumes *insert*: $\bigwedge x \text{ } ys. \llbracket P \{x\} \flat; P \text{ } ys \rrbracket \implies P (\{x\} \flat \cup \flat ys)$

shows $P (xs :: 'a :: \text{bifinite lower-pd})$

$\langle \text{proof} \rangle$

lemma *lower-pd-induct* [*case-names adm lower-unit lower-plus, induct type: lower-pd*]:

assumes *P*: *adm P*

assumes *unit*: $\bigwedge x. P \{x\} \flat$

assumes *plus*: $\bigwedge xs \text{ } ys. \llbracket P \text{ } xs; P \text{ } ys \rrbracket \implies P (xs \cup \flat ys)$

shows $P (xs :: 'a :: \text{bifinite lower-pd})$

$\langle \text{proof} \rangle$

30.5 Monadic bind

definition

lower-bind-basis ::

$'a :: \text{bifinite pd-basis} \Rightarrow ('a \rightarrow 'b \text{ lower-pd}) \rightarrow 'b :: \text{bifinite lower-pd}$ **where**

lower-bind-basis = *fold-pd*

$(\lambda a. \Lambda f. f \cdot (\text{Rep-compact-basis } a))$

$(\lambda x \text{ } y. \Lambda f. x \cdot f \cup \flat y \cdot f)$

lemma *ACT-lower-bind*:

semilattice $(\lambda x \text{ } y. \Lambda f. x \cdot f \cup \flat y \cdot f)$

$\langle \text{proof} \rangle$

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*) =

$(\Lambda f. \text{lower-bind-basis } t \cdot f \cup \flat \text{lower-bind-basis } u \cdot f)$

$\langle \text{proof} \rangle$

lemma *lower-bind-basis-mono*:

$t \leq \flat u \implies \text{lower-bind-basis } t \sqsubseteq \text{lower-bind-basis } u$

$\langle \text{proof} \rangle$

definition

lower-bind :: $'a :: \text{bifinite lower-pd} \rightarrow ('a \rightarrow 'b \text{ lower-pd}) \rightarrow 'b :: \text{bifinite lower-pd}$

where

lower-bind = *lower-pd.extension lower-bind-basis*

syntax

-lower-bind :: [*logic, logic, logic*] \Rightarrow *logic*

$(\langle (\langle \text{indent}=3 \text{ notation}=\langle \text{binder lower-bind} \rangle \rangle \cup \flat \cdot \in \cdot / \cdot) \rangle [0, 0, 10] 10)$

translations

$$\bigcup_{x \in xs}. e == \text{CONST } \text{lower-bind} \cdot xs \cdot (\Lambda x. e)$$

lemma *lower-bind-principal* [simp]:

$$\text{lower-bind} \cdot (\text{lower-principal } t) = \text{lower-bind-basis } t$$

⟨proof⟩

lemma *lower-bind-unit* [simp]:

$$\text{lower-bind} \cdot \{x\} \flat f = f \cdot x$$

⟨proof⟩

lemma *lower-bind-plus* [simp]:

$$\text{lower-bind} \cdot (xs \cup ys) \cdot f = \text{lower-bind} \cdot xs \cdot f \cup \text{lower-bind} \cdot ys \cdot f$$

⟨proof⟩

lemma *lower-bind-strict* [simp]: $\text{lower-bind} \cdot \perp \cdot f = f \cdot \perp$

⟨proof⟩

lemma *lower-bind-bind*:

$$\text{lower-bind} \cdot (\text{lower-bind} \cdot xs \cdot f) \cdot g = \text{lower-bind} \cdot xs \cdot (\Lambda x. \text{lower-bind} \cdot (f \cdot x) \cdot g)$$

⟨proof⟩

30.6 Map

definition

$$\text{lower-map} :: ('a :: \text{bifinite} \rightarrow 'b :: \text{bifinite}) \rightarrow 'a \text{ lower-pd} \rightarrow 'b \text{ lower-pd} \text{ where}$$

$$\text{lower-map} = (\Lambda f \ xs. \text{lower-bind} \cdot xs \cdot (\Lambda x. \{f \cdot x\} \flat))$$

lemma *lower-map-unit* [simp]:

$$\text{lower-map} \cdot f \cdot \{x\} \flat = \{f \cdot x\} \flat$$

⟨proof⟩

lemma *lower-map-plus* [simp]:

$$\text{lower-map} \cdot f \cdot (xs \cup ys) = \text{lower-map} \cdot f \cdot xs \cup \text{lower-map} \cdot f \cdot ys$$

⟨proof⟩

lemma *lower-map-bottom* [simp]: $\text{lower-map} \cdot f \cdot \perp = \{f \cdot \perp\} \flat$

⟨proof⟩

lemma *lower-map-ident*: $\text{lower-map} \cdot (\Lambda x. x) \cdot xs = xs$

⟨proof⟩

lemma *lower-map-ID*: $\text{lower-map} \cdot ID = ID$

⟨proof⟩

lemma *lower-map-map*:

$$\text{lower-map} \cdot f \cdot (\text{lower-map} \cdot g \cdot xs) = \text{lower-map} \cdot (\Lambda x. f \cdot (g \cdot x)) \cdot xs$$

⟨proof⟩

lemma *lower-bind-map*:

$lower\text{-}bind.(lower\text{-}map.f.xs).g = lower\text{-}bind.xs.(\Lambda x. g.(f.x))$
 $\langle proof \rangle$

lemma *lower-map-bind*:

$lower\text{-}map.f.(lower\text{-}bind.xs.g) = lower\text{-}bind.xs.(\Lambda x. lower\text{-}map.f.(g.x))$
 $\langle proof \rangle$

lemma *ep-pair-lower-map*: $ep\text{-}pair\ e\ p \implies ep\text{-}pair\ (lower\text{-}map.e)\ (lower\text{-}map.p)$

$\langle proof \rangle$

lemma *deflation-lower-map*: $deflation\ d \implies deflation\ (lower\text{-}map.d)$

$\langle proof \rangle$

lemma *finite-deflation-lower-map*:

assumes *finite-deflation d* **shows** *finite-deflation (lower-map.d)*
 $\langle proof \rangle$

30.7 Lower powerdomain is bifinite

lemma *approx-chain-lower-map*:

assumes *approx-chain a*
shows *approx-chain* $(\lambda i. lower\text{-}map.(a\ i))$
 $\langle proof \rangle$

instance *lower-pd* :: $(bifinite)\ bifinite$

$\langle proof \rangle$

30.8 Join

definition

$lower\text{-}join :: 'a::bifinite\ lower\text{-}pd\ lower\text{-}pd \rightarrow 'a\ lower\text{-}pd$ **where**
 $lower\text{-}join = (\Lambda\ xss. lower\text{-}bind.xss.(\Lambda\ xs. xs))$

lemma *lower-join-unit* [simp]:

$lower\text{-}join.\{xs\}^\flat = xs$
 $\langle proof \rangle$

lemma *lower-join-plus* [simp]:

$lower\text{-}join.(xss \sqcup yss) = lower\text{-}join.xss \sqcup lower\text{-}join.yss$
 $\langle proof \rangle$

lemma *lower-join-bottom* [simp]: $lower\text{-}join.\perp = \perp$

$\langle proof \rangle$

lemma *lower-join-map-unit*:

$lower\text{-}join.(lower\text{-}map.lower\text{-}unit.xs) = xs$
 $\langle proof \rangle$

lemma *lower-join-map-join*:

$lower-join \cdot (lower-map \cdot lower-join \cdot xss) = lower-join \cdot (lower-join \cdot xss)$
 $\langle proof \rangle$

lemma *lower-join-map-map*:

$lower-join \cdot (lower-map \cdot (lower-map \cdot f) \cdot xss) =$
 $lower-map \cdot f \cdot (lower-join \cdot xss)$
 $\langle proof \rangle$

end

31 Convex powerdomain

theory *ConvexPD*

imports *UpperPD LowerPD*

begin

31.1 Basis preorder

definition

$convex-le :: 'a :: bifinite \text{ pd-basis} \Rightarrow 'a \text{ pd-basis} \Rightarrow bool$ (**infix** $\leq_{\mathfrak{h}}$ 50) **where**
 $convex-le = (\lambda u \ v. u \leq_{\#} v \wedge u \leq_b v)$

lemma *convex-le-refl* [simp]: $t \leq_{\mathfrak{h}} t$

$\langle proof \rangle$

lemma *convex-le-trans*: $\llbracket t \leq_{\mathfrak{h}} u; u \leq_{\mathfrak{h}} v \rrbracket \Longrightarrow t \leq_{\mathfrak{h}} v$

$\langle proof \rangle$

interpretation *convex-le*: *preorder convex-le*

$\langle proof \rangle$

lemma *upper-le-minimal* [simp]: $PDU\text{Unit } compact_bot \leq_{\mathfrak{h}} t$

$\langle proof \rangle$

lemma *PDUnit-convex-mono*: $x \sqsubseteq y \Longrightarrow PDU\text{Unit } x \leq_{\mathfrak{h}} PDU\text{Unit } y$

$\langle proof \rangle$

lemma *PDPlus-convex-mono*: $\llbracket s \leq_{\mathfrak{h}} t; u \leq_{\mathfrak{h}} v \rrbracket \Longrightarrow PDPlus \ s \ u \leq_{\mathfrak{h}} PDPlus \ t \ v$

$\langle proof \rangle$

lemma *convex-le-PDUnit-PDUnit-iff* [simp]:

$(PDU\text{Unit } a \leq_{\mathfrak{h}} PDU\text{Unit } b) = (a \sqsubseteq b)$

$\langle proof \rangle$

lemma *convex-le-PDUnit-lemma1*:

$(PDU\text{Unit } a \leq_{\mathfrak{h}} t) = (\forall b \in Rep_pd_basis \ t. a \sqsubseteq b)$

$\langle proof \rangle$

lemma *convex-le-PDUnit-PDPlus-iff* [simp]:

$(PDUnit\ a \leq_{\mathfrak{h}} PDPlus\ t\ u) = (PDUnit\ a \leq_{\mathfrak{h}} t \wedge PDUnit\ a \leq_{\mathfrak{h}} u)$
 $\langle proof \rangle$

lemma *convex-le-PDUnit-lemma2*:

$(t \leq_{\mathfrak{h}} PDUnit\ b) = (\forall a \in Rep\text{-}pd\text{-}basis\ t. a \sqsubseteq b)$
 $\langle proof \rangle$

lemma *convex-le-PDPlus-PDUnit-iff* [simp]:

$(PDPlus\ t\ u \leq_{\mathfrak{h}} PDUnit\ a) = (t \leq_{\mathfrak{h}} PDUnit\ a \wedge u \leq_{\mathfrak{h}} PDUnit\ a)$
 $\langle proof \rangle$

lemma *convex-le-PDPlus-lemma*:

assumes $z: PDPlus\ t\ u \leq_{\mathfrak{h}} z$
shows $\exists v\ w. z = PDPlus\ v\ w \wedge t \leq_{\mathfrak{h}} v \wedge u \leq_{\mathfrak{h}} w$
 $\langle proof \rangle$

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 $P\ t\ u$
 $\langle proof \rangle$

31.2 Type definition

typedef $'a::bifinite\ convex\text{-}pd$ $(\langle \langle notation = \langle postfix\ convex\text{-}pd \rangle \rangle'(-)_{\mathfrak{h}} \rangle) =$
 $\{S::'a\ pd\text{-}basis\ set. convex\text{-}le.\ ideal\ S\}$
 $\langle proof \rangle$

instantiation *convex-pd* :: (bifinite) below
begin

definition

$x \sqsubseteq y \longleftrightarrow Rep\text{-}convex\text{-}pd\ x \subseteq Rep\text{-}convex\text{-}pd\ y$

instance $\langle proof \rangle$

end

instance *convex-pd* :: (bifinite) po

$\langle proof \rangle$

instance *convex-pd* :: (bifinite) cpo

$\langle proof \rangle$

definition

convex-principal :: $'a::bifinite\ pd\text{-}basis \Rightarrow 'a\ convex\text{-}pd$ **where**

convex-principal $t = \text{Abs-convex-pd } \{u. u \leq_{\mathfrak{h}} t\}$

interpretation *convex-pd*:

ideal-completion convex-le convex-principal Rep-convex-pd
 $\langle \text{proof} \rangle$

Convex powerdomain is pointed

lemma *convex-pd-minimal*: *convex-principal* (*PDUnit compact-bot*) \sqsubseteq *ys*
 $\langle \text{proof} \rangle$

instance *convex-pd* :: (*bifinite*) *pcpo*
 $\langle \text{proof} \rangle$

lemma *inst-convex-pd-pcpo*: $\perp = \text{convex-principal } (\text{PDUnit compact-bot})$
 $\langle \text{proof} \rangle$

31.3 Monadic unit and plus

definition

convex-unit :: '*a*::*bifinite* \rightarrow '*a* *convex-pd* **where**
convex-unit = *compact-basis.extension* ($\lambda a. \text{convex-principal } (\text{PDUnit } a)$)

definition

convex-plus :: '*a*::*bifinite* *convex-pd* \rightarrow '*a* *convex-pd* \rightarrow '*a* *convex-pd* **where**
convex-plus = *convex-pd.extension* ($\lambda t. \text{convex-pd.extension } (\lambda u. \text{convex-principal } (\text{PDPlus } t u))$)

abbreviation

convex-add :: '*a*::*bifinite* *convex-pd* \Rightarrow '*a* *convex-pd* \Rightarrow '*a* *convex-pd*
 (**infixl** $\langle \cup_{\mathfrak{h}} \rangle$ 65) **where**
 $xs \cup_{\mathfrak{h}} ys == \text{convex-plus} \cdot xs \cdot ys$

syntax

-convex-pd :: *args* \Rightarrow *logic* ($\langle (\langle \text{indent}=1 \text{ notation}=\langle \text{mixfix convex-pd enumeration} \rangle \{-\}_{\mathfrak{h}} \rangle) \rangle$)

translations

$\{x, xs\}_{\mathfrak{h}} == \{x\}_{\mathfrak{h}} \cup_{\mathfrak{h}} \{xs\}_{\mathfrak{h}}$
 $\{x\}_{\mathfrak{h}} == \text{CONST } \text{convex-unit} \cdot x$

lemma *convex-unit-Rep-compact-basis* [*simp*]:

$\{\text{Rep-compact-basis } a\}_{\mathfrak{h}} = \text{convex-principal } (\text{PDUnit } a)$
 $\langle \text{proof} \rangle$

lemma *convex-plus-principal* [*simp*]:

$\text{convex-principal } t \cup_{\mathfrak{h}} \text{convex-principal } u = \text{convex-principal } (\text{PDPlus } t u)$
 $\langle \text{proof} \rangle$

interpretation *convex-add*: *semilattice convex-add* $\langle \text{proof} \rangle$

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*]:
 $\{x\} \sqsubseteq ys \sqcup z \iff \{x\} \sqsubseteq ys \wedge \{x\} \sqsubseteq z$
<proof>

lemma *convex-plus-below-unit-iff* [*simp*]:
 $xs \sqcup \{z\} \sqsubseteq ys \iff xs \sqsubseteq \{z\} \wedge ys \sqsubseteq \{z\}$
<proof>

lemma *convex-unit-below-iff* [*simp*]: $\{x\} \sqsubseteq \{y\} \iff x \sqsubseteq y$
<proof>

lemma *convex-unit-eq-iff* [*simp*]: $\{x\} = \{y\} \iff x = y$
<proof>

lemma *convex-unit-strict* [*simp*]: $\{\perp\} = \perp$
<proof>

lemma *convex-unit-bottom-iff* [*simp*]: $\{x\} = \perp \iff x = \perp$
<proof>

lemma *compact-convex-unit*: *compact* $x \implies \text{compact } \{x\}$
<proof>

lemma *compact-convex-unit-iff* [*simp*]: *compact* $\{x\} \iff \text{compact } x$
<proof>

lemma *compact-convex-plus* [*simp*]:
 $\llbracket \text{compact } xs; \text{compact } ys \rrbracket \implies \text{compact } (xs \sqcup ys)$
<proof>

31.4 Induction rules

lemma *convex-pd-induct1*:
assumes P : *adm* P
assumes *unit*: $\bigwedge x. P \{x\}$

assumes *insert*: $\bigwedge x \text{ } ys. \llbracket P \{x\} \sqcup; P \text{ } ys \rrbracket \implies P (\{x\} \sqcup \sqcup ys)$
shows $P (xs :: 'a :: \text{bifinite convex-pd})$
 $\langle \text{proof} \rangle$

lemma *convex-pd-induct* [*case-names adm convex-unit convex-plus, induct type: convex-pd*]:

assumes $P: \text{adm } P$
assumes *unit*: $\bigwedge x. P \{x\} \sqcup$
assumes *plus*: $\bigwedge xs \text{ } ys. \llbracket P \text{ } xs; P \text{ } ys \rrbracket \implies P (xs \sqcup \sqcup ys)$
shows $P (xs :: 'a :: \text{bifinite convex-pd})$
 $\langle \text{proof} \rangle$

31.5 Monadic bind

definition

convex-bind-basis ::
 $'a :: \text{bifinite pd-basis} \Rightarrow ('a \rightarrow 'b \text{ convex-pd}) \rightarrow 'b :: \text{bifinite convex-pd}$ **where**
convex-bind-basis = *fold-pd*
 $(\lambda a. \Lambda f. f \cdot (\text{Rep-compact-basis } a))$
 $(\lambda x \text{ } y. \Lambda f. x \cdot f \sqcup \sqcup y \cdot f)$

lemma *ACI-convex-bind*:

semilattice $(\lambda x \text{ } y. \Lambda f. x \cdot f \sqcup \sqcup y \cdot f)$
 $\langle \text{proof} \rangle$

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*) =
 $(\Lambda f. \text{convex-bind-basis } t \cdot f \sqcup \sqcup \text{convex-bind-basis } u \cdot f)$
 $\langle \text{proof} \rangle$

lemma *convex-bind-basis-mono*:

$t \leq \sqcup u \implies \text{convex-bind-basis } t \sqsubseteq \text{convex-bind-basis } u$
 $\langle \text{proof} \rangle$

definition

convex-bind :: $'a :: \text{bifinite convex-pd} \rightarrow ('a \rightarrow 'b \text{ convex-pd}) \rightarrow 'b :: \text{bifinite convex-pd}$
where
convex-bind = *convex-pd.extension convex-bind-basis*

syntax

-convex-bind :: [*logic, logic, logic*] \Rightarrow *logic*
 $(\langle \langle \text{indent} = 3 \text{ notation} = \langle \text{binder convex-bind} \rangle \rangle \cup \sqcup \in \cdot / \cdot \rangle [0, 0, 10] 10)$

translations

$\bigcup \sqcup x \in xs. e == \text{CONST } \text{convex-bind} \cdot xs \cdot (\Lambda x. e)$

lemma *convex-bind-principal* [*simp*]:

$\text{convex-bind} \cdot (\text{convex-principal } t) = \text{convex-bind-basis } t$
 $\langle \text{proof} \rangle$

lemma *convex-bind-unit* [simp]:
 $\text{convex-bind} \cdot \{x\} \cdot f = f \cdot x$
 $\langle \text{proof} \rangle$

lemma *convex-bind-plus* [simp]:
 $\text{convex-bind} \cdot (xs \cup ys) \cdot f = \text{convex-bind} \cdot xs \cdot f \cup \text{convex-bind} \cdot ys \cdot f$
 $\langle \text{proof} \rangle$

lemma *convex-bind-strict* [simp]: $\text{convex-bind} \cdot \perp \cdot f = f \cdot \perp$
 $\langle \text{proof} \rangle$

lemma *convex-bind-bind*:
 $\text{convex-bind} \cdot (\text{convex-bind} \cdot xs \cdot f) \cdot g =$
 $\text{convex-bind} \cdot xs \cdot (\Lambda x. \text{convex-bind} \cdot (f \cdot x) \cdot g)$
 $\langle \text{proof} \rangle$

31.6 Map

definition
 $\text{convex-map} :: ('a :: \text{bifinite} \rightarrow 'b) \rightarrow 'a \text{ convex-pd} \rightarrow 'b :: \text{bifinite convex-pd}$ **where**
 $\text{convex-map} = (\Lambda f \ xs. \text{convex-bind} \cdot xs \cdot (\Lambda x. \{f \cdot x\}))$

lemma *convex-map-unit* [simp]:
 $\text{convex-map} \cdot f \cdot \{x\} = \{f \cdot x\}$
 $\langle \text{proof} \rangle$

lemma *convex-map-plus* [simp]:
 $\text{convex-map} \cdot f \cdot (xs \cup ys) = \text{convex-map} \cdot f \cdot xs \cup \text{convex-map} \cdot f \cdot ys$
 $\langle \text{proof} \rangle$

lemma *convex-map-bottom* [simp]: $\text{convex-map} \cdot f \cdot \perp = \{f \cdot \perp\}$
 $\langle \text{proof} \rangle$

lemma *convex-map-ident*: $\text{convex-map} \cdot (\Lambda x. x) \cdot xs = xs$
 $\langle \text{proof} \rangle$

lemma *convex-map-ID*: $\text{convex-map} \cdot ID = ID$
 $\langle \text{proof} \rangle$

lemma *convex-map-map*:
 $\text{convex-map} \cdot f \cdot (\text{convex-map} \cdot g \cdot xs) = \text{convex-map} \cdot (\Lambda x. f \cdot (g \cdot x)) \cdot xs$
 $\langle \text{proof} \rangle$

lemma *convex-bind-map*:
 $\text{convex-bind} \cdot (\text{convex-map} \cdot f \cdot xs) \cdot g = \text{convex-bind} \cdot xs \cdot (\Lambda x. g \cdot (f \cdot x))$
 $\langle \text{proof} \rangle$

lemma *convex-map-bind*:

$\text{convex-map} \cdot f \cdot (\text{convex-bind} \cdot xs \cdot g) = \text{convex-bind} \cdot xs \cdot (\Lambda x. \text{convex-map} \cdot f \cdot (g \cdot x))$
 $\langle \text{proof} \rangle$

lemma *ep-pair-convex-map*: $\text{ep-pair } e \ p \implies \text{ep-pair } (\text{convex-map} \cdot e) \ (\text{convex-map} \cdot p)$
 $\langle \text{proof} \rangle$

lemma *deflation-convex-map*: $\text{deflation } d \implies \text{deflation } (\text{convex-map} \cdot d)$
 $\langle \text{proof} \rangle$

lemma *finite-deflation-convex-map*:

assumes *finite-deflation* d **shows** *finite-deflation* $(\text{convex-map} \cdot d)$
 $\langle \text{proof} \rangle$

31.7 Convex powerdomain is bifinite

lemma *approx-chain-convex-map*:

assumes *approx-chain* a
shows *approx-chain* $(\lambda i. \text{convex-map} \cdot (a \ i))$
 $\langle \text{proof} \rangle$

instance *convex-pd* :: $(\text{bifinite}) \text{ bifinite}$
 $\langle \text{proof} \rangle$

31.8 Join

definition

$\text{convex-join} :: 'a :: \text{bifinite convex-pd convex-pd} \rightarrow 'a \text{ convex-pd}$ **where**
 $\text{convex-join} = (\Lambda xss. \text{convex-bind} \cdot xss \cdot (\Lambda xs. xs))$

lemma *convex-join-unit* [*simp*]:

$\text{convex-join} \cdot \{xs\} \Downarrow = xs$
 $\langle \text{proof} \rangle$

lemma *convex-join-plus* [*simp*]:

$\text{convex-join} \cdot (xss \cup \Downarrow yss) = \text{convex-join} \cdot xss \cup \Downarrow \text{convex-join} \cdot yss$
 $\langle \text{proof} \rangle$

lemma *convex-join-bottom* [*simp*]: $\text{convex-join} \cdot \perp = \perp$

$\langle \text{proof} \rangle$

lemma *convex-join-map-unit*:

$\text{convex-join} \cdot (\text{convex-map} \cdot \text{convex-unit} \cdot xs) = xs$
 $\langle \text{proof} \rangle$

lemma *convex-join-map-join*:

$\text{convex-join} \cdot (\text{convex-map} \cdot \text{convex-join} \cdot xsss) = \text{convex-join} \cdot (\text{convex-join} \cdot xsss)$
 $\langle \text{proof} \rangle$

lemma *convex-join-map-map*:
 $\text{convex-join} \cdot (\text{convex-map} \cdot (\text{convex-map} \cdot f) \cdot xss) =$
 $\text{convex-map} \cdot f \cdot (\text{convex-join} \cdot xss)$
 ⟨proof⟩

31.9 Conversions to other powerdomains

Convex to upper

lemma *convex-le-imp-upper-le*: $t \leq_{\natural} u \implies t \leq_{\sharp} u$
 ⟨proof⟩

definition
 $\text{convex-to-upper} :: 'a :: \text{bifinite convex-pd} \rightarrow 'a \text{ upper-pd}$ **where**
 $\text{convex-to-upper} = \text{convex-pd.extension upper-principal}$

lemma *convex-to-upper-principal* [simp]:
 $\text{convex-to-upper} \cdot (\text{convex-principal } t) = \text{upper-principal } t$
 ⟨proof⟩

lemma *convex-to-upper-unit* [simp]:
 $\text{convex-to-upper} \cdot \{x\}_{\natural} = \{x\}_{\sharp}$
 ⟨proof⟩

lemma *convex-to-upper-plus* [simp]:
 $\text{convex-to-upper} \cdot (xs \cup_{\natural} ys) = \text{convex-to-upper} \cdot xs \cup_{\sharp} \text{convex-to-upper} \cdot ys$
 ⟨proof⟩

lemma *convex-to-upper-bind* [simp]:
 $\text{convex-to-upper} \cdot (\text{convex-bind} \cdot xs \cdot f) =$
 $\text{upper-bind} \cdot (\text{convex-to-upper} \cdot xs) \cdot (\text{convex-to-upper} \text{ oo } f)$
 ⟨proof⟩

lemma *convex-to-upper-map* [simp]:
 $\text{convex-to-upper} \cdot (\text{convex-map} \cdot f \cdot xs) = \text{upper-map} \cdot f \cdot (\text{convex-to-upper} \cdot xs)$
 ⟨proof⟩

lemma *convex-to-upper-join* [simp]:
 $\text{convex-to-upper} \cdot (\text{convex-join} \cdot xss) =$
 $\text{upper-bind} \cdot (\text{convex-to-upper} \cdot xss) \cdot \text{convex-to-upper}$
 ⟨proof⟩

Convex to lower

lemma *convex-le-imp-lower-le*: $t \leq_{\natural} u \implies t \leq_{\flat} u$
 ⟨proof⟩

definition
 $\text{convex-to-lower} :: 'a :: \text{bifinite convex-pd} \rightarrow 'a \text{ lower-pd}$ **where**
 $\text{convex-to-lower} = \text{convex-pd.extension lower-principal}$

lemma *convex-to-lower-principal* [simp]:
 $\text{convex-to-lower} \cdot (\text{convex-principal } t) = \text{lower-principal } t$
 ⟨proof⟩

lemma *convex-to-lower-unit* [simp]:
 $\text{convex-to-lower} \cdot \{x\}^\natural = \{x\}^\flat$
 ⟨proof⟩

lemma *convex-to-lower-plus* [simp]:
 $\text{convex-to-lower} \cdot (xs \cup^\natural ys) = \text{convex-to-lower} \cdot xs \cup^\flat \text{convex-to-lower} \cdot ys$
 ⟨proof⟩

lemma *convex-to-lower-bind* [simp]:
 $\text{convex-to-lower} \cdot (\text{convex-bind} \cdot xs \cdot f) =$
 $\text{lower-bind} \cdot (\text{convex-to-lower} \cdot xs) \cdot (\text{convex-to-lower } oo f)$
 ⟨proof⟩

lemma *convex-to-lower-map* [simp]:
 $\text{convex-to-lower} \cdot (\text{convex-map} \cdot f \cdot xs) = \text{lower-map} \cdot f \cdot (\text{convex-to-lower} \cdot xs)$
 ⟨proof⟩

lemma *convex-to-lower-join* [simp]:
 $\text{convex-to-lower} \cdot (\text{convex-join} \cdot xss) =$
 $\text{lower-bind} \cdot (\text{convex-to-lower} \cdot xss) \cdot \text{convex-to-lower}$
 ⟨proof⟩

Ordering property

lemma *convex-pd-below-iff*:
 $(xs \sqsubseteq ys) =$
 $(\text{convex-to-upper} \cdot xs \sqsubseteq \text{convex-to-upper} \cdot ys \wedge$
 $\text{convex-to-lower} \cdot xs \sqsubseteq \text{convex-to-lower} \cdot ys)$
 ⟨proof⟩

lemmas *convex-plus-below-plus-iff* =
 $\text{convex-pd-below-iff}$ [where $xs = xs \cup^\natural ys$ and $ys = zs \cup^\natural ws$]
 for $xs \ ys \ zs \ ws$

lemmas *convex-pd-below-simps* =
 $\text{convex-unit-below-plus-iff}$
 $\text{convex-plus-below-unit-iff}$
 $\text{convex-plus-below-plus-iff}$
 $\text{convex-unit-below-iff}$
 $\text{convex-to-upper-unit}$
 $\text{convex-to-upper-plus}$
 $\text{convex-to-lower-unit}$
 $\text{convex-to-lower-plus}$
 $\text{upper-pd-below-simps}$
 $\text{lower-pd-below-simps}$

end

32 Powerdomains

theory *Powerdomains*
imports *ConvexPD Domain*
begin

32.1 Universal domain embeddings

definition *upper-emb* = *udom-emb* ($\lambda i. \text{upper-map} \cdot (\text{udom-approx } i)$)

definition *upper-prj* = *udom-prj* ($\lambda i. \text{upper-map} \cdot (\text{udom-approx } i)$)

definition *lower-emb* = *udom-emb* ($\lambda i. \text{lower-map} \cdot (\text{udom-approx } i)$)

definition *lower-prj* = *udom-prj* ($\lambda i. \text{lower-map} \cdot (\text{udom-approx } i)$)

definition *convex-emb* = *udom-emb* ($\lambda i. \text{convex-map} \cdot (\text{udom-approx } i)$)

definition *convex-prj* = *udom-prj* ($\lambda i. \text{convex-map} \cdot (\text{udom-approx } i)$)

lemma *ep-pair-upper*: *ep-pair upper-emb upper-prj*
 ⟨*proof*⟩

lemma *ep-pair-lower*: *ep-pair lower-emb lower-prj*
 ⟨*proof*⟩

lemma *ep-pair-convex*: *ep-pair convex-emb convex-prj*
 ⟨*proof*⟩

32.2 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*:
 $\text{cast} \cdot (\text{upper-defl} \cdot A) = \text{upper-emb} \text{ oo } \text{upper-map} \cdot (\text{cast} \cdot A) \text{ oo } \text{upper-prj}$
 ⟨*proof*⟩

lemma *cast-lower-defl*:
 $\text{cast} \cdot (\text{lower-defl} \cdot A) = \text{lower-emb} \text{ oo } \text{lower-map} \cdot (\text{cast} \cdot A) \text{ oo } \text{lower-prj}$
 ⟨*proof*⟩

lemma *cast-convex-defl*:

$cast.(convex-defl.A) = convex-emb \text{ oo } convex-map.(cast.A) \text{ oo } convex-prj$
 $\langle proof \rangle$

32.3 Domain class instances

instantiation *upper-pd* :: (domain) domain
begin

definition

$emb = upper-emb \text{ oo } upper-map.emb$

definition

$prj = upper-map.prj \text{ oo } upper-prj$

definition

$defl (t::'a \text{ upper-pd } itself) = upper-defl.DEFL('a)$

definition

$(liftemb :: 'a \text{ upper-pd } u \rightarrow udom \ u) = u-map.emb$

definition

$(liftprj :: udom \ u \rightarrow 'a \text{ upper-pd } u) = u-map.prj$

definition

$liftdefl (t::'a \text{ upper-pd } itself) = liftdefl-of.DEFL('a \text{ upper-pd})$

instance $\langle proof \rangle$

end

instantiation *lower-pd* :: (domain) domain
begin

definition

$emb = lower-emb \text{ oo } lower-map.emb$

definition

$prj = lower-map.prj \text{ oo } lower-prj$

definition

$defl (t::'a \text{ lower-pd } itself) = lower-defl.DEFL('a)$

definition

$(liftemb :: 'a \text{ lower-pd } u \rightarrow udom \ u) = u-map.emb$

definition

$(liftprj :: udom \ u \rightarrow 'a \text{ lower-pd } u) = u-map.prj$

definition

$\text{liftdefl } (t :: 'a \text{ lower-pd itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \text{ lower-pd})$

instance $\langle \text{proof} \rangle$

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
 $\text{defl } (t :: 'a \text{ convex-pd itself}) = \text{convex-defl} \cdot \text{DEFL}('a)$

definition
 $(\text{liftemb} :: 'a \text{ convex-pd } u \rightarrow \text{udom } u) = u\text{-map} \cdot \text{emb}$

definition
 $(\text{liftprj} :: \text{udom } u \rightarrow 'a \text{ convex-pd } u) = u\text{-map} \cdot \text{prj}$

definition
 $\text{liftdefl } (t :: 'a \text{ convex-pd itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \text{ convex-pd})$

instance $\langle \text{proof} \rangle$

end

lemma DEFL-upper : $\text{DEFL}('a :: \text{domain upper-pd}) = \text{upper-defl} \cdot \text{DEFL}('a)$
 $\langle \text{proof} \rangle$

lemma DEFL-lower : $\text{DEFL}('a :: \text{domain lower-pd}) = \text{lower-defl} \cdot \text{DEFL}('a)$
 $\langle \text{proof} \rangle$

lemma DEFL-convex : $\text{DEFL}('a :: \text{domain convex-pd}) = \text{convex-defl} \cdot \text{DEFL}('a)$
 $\langle \text{proof} \rangle$

32.4 Isomorphic deflations

lemma isodefl-upper :
 $\text{isodefl } d \ t \implies \text{isodefl } (\text{upper-map} \cdot d) (\text{upper-defl} \cdot t)$
 $\langle \text{proof} \rangle$

lemma isodefl-lower :
 $\text{isodefl } d \ t \implies \text{isodefl } (\text{lower-map} \cdot d) (\text{lower-defl} \cdot t)$
 $\langle \text{proof} \rangle$

lemma *isodeft-convex*:

isodeft d t \implies isodeft (convex-map·d) (convex-defl·t)
<proof>

32.5 Domain package setup for powerdomains

lemmas [*domain-defl-simps*] = *DEFL-upper DEFL-lower DEFL-convex*

lemmas [*domain-map-ID*] = *upper-map-ID lower-map-ID convex-map-ID*

lemmas [*domain-isodeft*] = *isodeft-upper isodeft-lower isodeft-convex*

lemmas [*domain-deflation*] =

deflation-upper-map deflation-lower-map deflation-convex-map

<ML>

end

theory *HOLCF*

imports

Main

Domain

Powerdomains

begin

default-sort *domain*

end