Isabelle/HOLCF — Higher-Order Logic of Computable Functions

June 8, 2008

Contents

1 Porder: Partial orders **7**
- 1.1 Type class for partial orders . . . 7
- 1.2 Upper bounds . . . 8
- 1.3 Least upper bounds . . . 8
- 1.4 Countable chains . . . 10
- 1.5 Totally ordered sets . . . 11
- 1.6 Finite chains . . . 12
- 1.7 Directed sets . . . 13

2 Pcpo: Classes cpo and pcpo **15**
- 2.1 Complete partial orders . . . 15
- 2.2 Pointed cpos . . . 18
- 2.3 Chain-finite and flat cpos . . . 20

3 Cont: Continuity and monotonicity **22**
- 3.1 Definitions . . . 23
- 3.2 *monofun f* $\wedge$ *contlub f* $\equiv$ *cont f* . . . 23
- 3.3 Continuity of basic functions . . . 25
- 3.4 Finite chains and flat pcpos . . . 26

4 Ffun: Class instances for the full function space **27**
- 4.1 Full function space is a partial order . . . 27
- 4.2 Full function space is chain complete . . . 28
- 4.3 Full function space is pointed . . . 30
- 4.4 Propagation of monotonicity and continuity . . . 30

5 Adm: Admissibility and compactness **33**
- 5.1 Definitions . . . 33
- 5.2 Admissibility on chain-finite types . . . 33
- 5.3 Admissibility of special formulae and propagation . . . 34
- 5.4 Compactness . . . 36

6 Pcpodef: Subtypes of pcpos **37**
6.1 Proving a subtype is a partial order . . . 37
6.2 Proving a subtype is finite . . . 37
6.3 Proving a subtype is chain-finite . . . 38
6.4 Proving a subtype is complete . . . 39
6.4.1 Continuity of *Rep* and *Abs* . . . 40
6.5 Proving subtype elements are compact . . . 40
6.6 Proving a subtype is pointed . . . 41
6.6.1 Strictness of *Rep* and *Abs* . . . 41
6.7 Proving a subtype is flat . . . 43
6.8 HOLCF type definition package . . . 43

7 Cfun: The type of continuous functions **43**
7.1 Definition of continuous function type . . . 43
7.2 Syntax for continuous lambda abstraction . . . 44
7.3 Continuous function space is pointed . . . 45
7.4 Basic properties of continuous functions . . . 45
7.5 Continuity of application . . . 46
7.6 Continuity simplification procedure . . . 49
7.7 Miscellaneous . . . 49
7.8 Continuous injection-retraction pairs . . . 50
7.9 Identity and composition . . . 51
7.10 Strictified functions . . . 52
7.11 Continuous let-bindings . . . 53

8 Bifinite: Bifinite domains and approximation **53**
8.1 Omega-profinite and bifinite domains . . . 54
8.2 Instance for continuous function space . . . 56

9 Cprod: The cpo of cartesian products **57**
9.1 Type *unit* is a pcpo . . . 58
9.2 Product type is a partial order . . . 58
9.3 Monotonicity of (-,-), *fst*, *snd* . . . 59
9.4 Product type is a cpo . . . 59
9.5 Product type is pointed . . . 60
9.6 Continuity of (-,-), *fst*, *snd* . . . 61
9.7 Continuous versions of constants . . . 61
9.8 Convert all lemmas to the continuous versions . . . 62
9.9 Product type is a bifinite domain . . . 64

10 Sprod: The type of strict products **65**
10.1 Definition of strict product type . . . 65
10.2 Definitions of constants . . . 66
10.3 Case analysis . . . 66

10.4 Properties of *spair* . . . 67
10.5 Properties of *sfst* and *ssnd* . . . 68
10.6 Compactness . . . 69
10.7 Properties of *ssplit* . . . 69
10.8 Strict product preserves flatness . . . 69
10.9 Strict product is a bifinite domain . . . 70

11 Discrete: Discrete cpo types **70**
11.1 Type *'a discr* is a discrete cpo . . . 71
11.2 Type *'a discr* is a cpo . . . 71
11.3 *undiscr* . . . 72

12 Up: The type of lifted values **72**
12.1 Definition of new type for lifting . . . 72
12.2 Ordering on lifted cpo . . . 72
12.3 Lifted cpo is a partial order . . . 73
12.4 Lifted cpo is a cpo . . . 73
12.5 Lifted cpo is pointed . . . 75
12.6 Continuity of *Iup* and *Ifup* . . . 76
12.7 Continuous versions of constants . . . 76
12.8 Lifted cpo is a bifinite domain . . . 78

13 Countable: Encoding (almost) everything into natural numbers **79**
13.1 The class of countable types . . . 79
13.2 Conversion functions . . . 79
13.3 Countable types . . . 80

14 Lift: Lifting types of class type to flat pcpo's **82**
14.1 Lift as a datatype . . . 83
14.2 Lift is flat . . . 84
14.3 Further operations . . . 84
14.4 Continuity Proofs for flift1, flift2 . . . 85
14.5 Lifted countable types are bifinite . . . 86

15 Tr: The type of lifted booleans **87**
15.1 Rewriting of HOLCF operations to HOL functions . . . 90
15.2 Compactness . . . 91

16 Ssum: The type of strict sums **91**
16.1 Definition of strict sum type . . . 91
16.2 Definitions of constructors . . . 91
16.3 Properties of *sinl* and *sinr* . . . 92
16.4 Case analysis . . . 94
16.5 Case analysis combinator . . . 94

16.6 Strict sum preserves flatness . . . 95
16.7 Strict sum is a bifinite domain . . . 95

17 One: The unit domain **96**

18 Fix: Fixed point operator and admissibility **97**
18.1 Iteration . . . 97
18.2 Least fixed point operator . . . 98
18.3 Fixed point induction . . . 100
18.4 Recursive let bindings . . . 100
18.5 Weak admissibility . . . 101

19 Fixrec: Package for defining recursive functions in HOLCF **102**
19.1 Maybe monad type . . . 102
19.1.1 Monadic bind operator . . . 103
19.1.2 Run operator . . . 103
19.1.3 Monad plus operator . . . 104
19.1.4 Fatbar combinator . . . 104
19.2 Case branch combinator . . . 105
19.3 Case syntax . . . 105
19.4 Pattern combinators for data constructors . . . 107
19.5 Wildcards, as-patterns, and lazy patterns . . . 110
19.6 Match functions for built-in types . . . 111
19.7 Mutual recursion . . . 113
19.8 Initializing the fixrec package . . . 113

20 Domain: Domain package **113**
20.1 Continuous isomorphisms . . . 113
20.2 Casedist . . . 115

21 SetPcpo: Set as a pointed cpo **117**
21.1 Admissibility of set predicates . . . 118
21.2 Compactness . . . 118

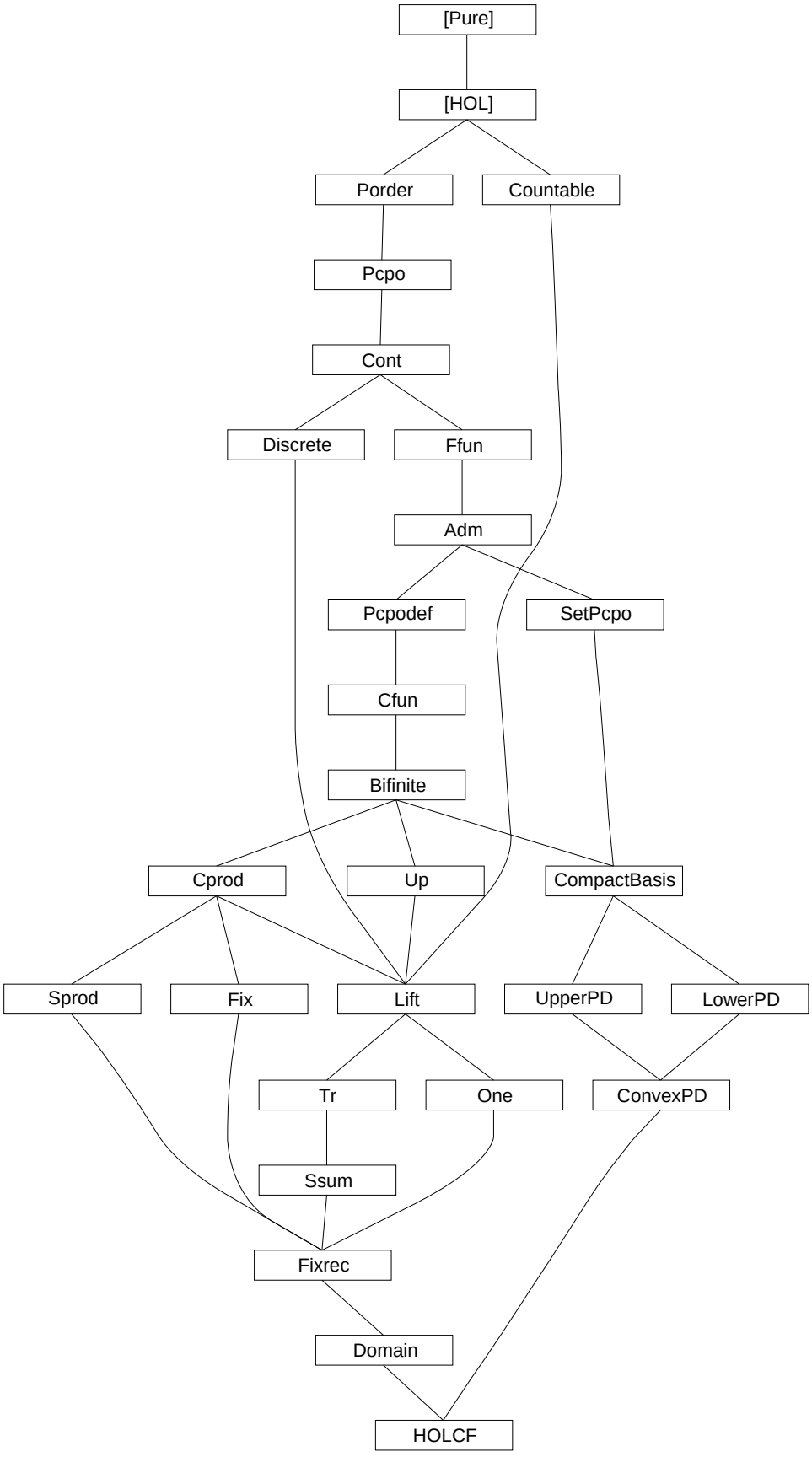
22 CompactBasis: Compact bases of domains **119**
22.1 Ideals over a preorder . . . 119
22.2 Defining functions in terms of basis elements . . . 120
22.3 Compact bases of bifinite domains . . . 126
22.4 A compact basis for powerdomains . . . 130

23 UpperPD: Upper powerdomain **133**
23.1 Basis preorder . . . 133
23.2 Type definition . . . 135
23.3 Approximation . . . 136
23.4 Monadic unit and plus . . . 137

23.5 Induction rules . . . 140
23.6 Monadic bind . . . 141
23.7 Map and join . . . 142

24 LowerPD: Lower powerdomain **143**
24.1 Basis preorder . . . 143
24.2 Type definition . . . 145
24.3 Approximation . . . 146
24.4 Monadic unit and plus . . . 147
24.5 Induction rules . . . 151
24.6 Monadic bind . . . 151
24.7 Map and join . . . 152

25 ConvexPD: Convex powerdomain **154**
25.1 Basis preorder . . . 154
25.2 Type definition . . . 156
25.3 Approximation . . . 158
25.4 Monadic unit and plus . . . 159
25.5 Induction rules . . . 162
25.6 Monadic bind . . . 162
25.7 Map and join . . . 163
25.8 Conversions to other powerdomains . . . 164


[Pure]
[HOL]
Porder
Countable
Pcpo
Cont
Discrete
Ffun
Adm
Pcpodef
SetPcpo
Cfun
Bifinite
Cprod
Up
CompactBasis
Sprod
Fix
Lift
UpperPD
LowerPD
Tr
One
ConvexPD
Ssum
Fixrec
Domain
HOLCF

1 Porder: Partial orders

theory *Porder*
imports *Datatype Finite-Set*
begin

1.1 Type class for partial orders

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

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

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

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

class *po* = *preorder* +
  assumes *antisym-less*: $[\![x \sqsubseteq y;\ y \sqsubseteq x]\!] \Longrightarrow x = y$

minimal fixes least element

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

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

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

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

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

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

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

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

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

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

1.2 Upper bounds

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

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

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

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

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

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

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

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

lemma *is-ub-insert* [*simp*]: (*insert x A*) <| *y* = (*x* ⊑ *y* ∧ *A* <| *y*)
unfolding *is-ub-def* **by** *fast*

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

1.3 Least upper bounds

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

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

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

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

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

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

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

access to some definition as inference rule

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

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

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

lubs are unique

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

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

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

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

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

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

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

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

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

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

1.4 Countable chains

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

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

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

chains are monotone functions

lemma *chain-mono*:
assumes Y: *chain* Y
shows $i \leq j \Longrightarrow Y\ i \sqsubseteq Y\ j$
apply (*induct j*)
apply *simp*
apply (*erule le-SucE*)
apply (*rule trans-less* [*OF - chainE* [*OF Y*]])
apply *simp*
apply *simp*
done

lemma *chain-mono-less*: $[\![chain\ Y; i < j]\!] \Longrightarrow Y\ i \sqsubseteq Y\ j$
by (*erule chain-mono*, *simp*)

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

technical lemmas about (least) upper bounds of chains

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

lemma *is-ub-range-shift*:

chain S $\Longrightarrow$ *range* $(\lambda i.\ S\ (i + j)) <|\ x =$ *range S* $<|\ x$
apply (*rule iffI*)
apply (*rule ub-rangeI*)
apply (*rule-tac y=S* $(i + j)$ **in** *trans-less*)
apply (*erule chain-mono*)
apply (*rule le-add1*)
apply (*erule ub-rangeD*)
apply (*rule ub-rangeI*)
apply (*erule ub-rangeD*)
done

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

the lub of a constant chain is the constant

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

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

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

1.5 Totally ordered sets

definition
— Arbitrary chains are total orders
tord :: *'a::po set* $\Rightarrow$ *bool* **where**
tord S $= (\forall\, x\ y.\ x \in S \wedge y \in S \longrightarrow (x \sqsubseteq y \vee y \sqsubseteq x))$

The range of a chain is a totally ordered

lemma *chain-tord*: *chain Y* $\Longrightarrow$ *tord* (*range Y*)
unfolding *tord-def*
apply (*clarify*, *rename-tac i j*)
apply (*rule-tac x=i* **and** *y=j* **in** *linorder-le-cases*)
apply (*fast intro*: *chain-mono*)+
done

lemma *finite-tord-has-max*:
⟦*finite S*; *S* $\neq \{\}$; *tord S*⟧ $\Longrightarrow \exists\, y{\in}S.\ \forall\, x{\in}S.\ x \sqsubseteq y$
apply (*induct S rule*: *finite-ne-induct*)
apply *simp*
apply (*drule meta-mp*, *simp add*: *tord-def*)
apply (*erule bexE*, *rename-tac z*)
apply (*subgoal-tac x* $\sqsubseteq z \vee z \sqsubseteq x$)
apply (*erule disjE*)
apply (*rule-tac x=z* **in** *bexI*, *simp*, *simp*)

apply (*rule-tac x=x* **in** *bexI*)
apply (*clarsimp elim!*: *rev-trans-less*)
apply *simp*
apply (*simp add*: *tord-def*)
done

1.6 Finite chains

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

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

results about finite chains

lemma *max-in-chainI*: (⋀*j*. *i* ≤ *j* ⟹ *Y i* = *Y j*) ⟹ *max-in-chain i Y*
unfolding *max-in-chain-def* **by** *fast*

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

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

lemma *lub-finch2*:
finite-chain C ⟹ *range C* <<| *C* (*LEAST i*. *max-in-chain i C*)
apply (*unfold finite-chain-def*)
apply (*erule conjE*)
apply (*erule LeastI2-ex*)
apply (*erule* (*1*) *lub-finch1*)
done

lemma *finch-imp-finite-range*: *finite-chain Y* ⟹ *finite* (*range Y*)
apply (*unfold finite-chain-def*, *clarify*)
apply (*rule-tac f=Y* **and** *n=Suc i* **in** *nat-seg-image-imp-finite*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule rangeE*, *rename-tac j*)
apply (*rule-tac x=i* **and** *y=j* **in** *linorder-le-cases*)
apply (*subgoal-tac Y j* = *Y i*, *simp*)

apply (*simp add*: *max-in-chain-def*)
apply *simp*
apply *fast*
done

lemma *finite-range-imp-finch*:
⟦*chain Y*; *finite* (*range Y*)⟧ ⟹ *finite-chain Y*
apply (*subgoal-tac* ∃ *y*∈*range Y*. ∀ *x*∈*range Y*. *x* ⊑ *y*)
apply (*clarsimp*, *rename-tac i*)
apply (*subgoal-tac max-in-chain i Y*)
apply (*simp add*: *finite-chain-def exI*)
apply (*simp add*: *max-in-chain-def po-eq-conv chain-mono*)
apply (*erule finite-tord-has-max*, *simp*)
apply (*erule chain-tord*)
done

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

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

lemma *lub-bin-chain*:
x ⊑ *y* ⟹ *range* (λ*i*::*nat*. *if i=0 then x else y*) <<| *y*
apply (*frule bin-chain*)
apply (*drule bin-chainmax*)
apply (*drule* (*1*) *lub-finch1*)
apply *simp*
done

the maximal element in a chain is its lub

lemma *lub-chain-maxelem*: ⟦*Y i* = *c*; ∀ *i*. *Y i* ⊑ *c*⟧ ⟹ *lub* (*range Y*) = *c*
by (*blast dest*: *ub-rangeD intro*: *thelubI is-lubI ub-rangeI*)

1.7 Directed sets

definition
directed :: '*a*::*po set* ⇒ *bool* **where**
directed S = ((∃ *x*. *x* ∈ *S*) ∧ (∀ *x*∈*S*. ∀ *y*∈*S*. ∃ *z*∈*S*. *x* ⊑ *z* ∧ *y* ⊑ *z*))

lemma *directedI*:
assumes *1*: ∃ *z*. *z* ∈ *S*
assumes *2*: ⋀*x y*. ⟦*x* ∈ *S*; *y* ∈ *S*⟧ ⟹ ∃ *z*∈*S*. *x* ⊑ *z* ∧ *y* ⊑ *z*
shows *directed S*
unfolding *directed-def* **using** *prems* **by** *fast*

lemma *directedD1*: *directed S* ⟹ ∃ *z*. *z* ∈ *S*
unfolding *directed-def* **by** *fast*

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

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

lemma *directedE2*:
  assumes *S*: *directed S*
  assumes *x*: $x \in S$ **and** *y*: $y \in S$
  obtains *z* **where** $z \in S\ x \sqsubseteq z\ y \sqsubseteq z$
by (*insert directedD2* [*OF S x y*], *fast*)

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

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

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

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

lemma *directed-bin*: *x* ⊑ *y* ⟹ *directed* {*x*, *y*}
by (*rule directedI*, *auto*)

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

end

2 Pcpo: Classes cpo and pcpo

theory *Pcpo*
imports *Porder*
begin

2.1 Complete partial orders

The class cpo of chain complete partial orders

axclass *cpo* < *po*
— class axiom:
cpo: *chain S* ⟹ ∃ *x*. *range S* <<| *x*

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

lemma *cpo-lubI*: *chain* (*S*::*nat* ⇒ *'a*::*cpo*) ⟹ *range S* <<| *lub* (*range S*)
by (*fast dest*: *cpo elim*: *lubI*)

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

Properties of the lub

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

lemma *is-lub-thelub*:
⟦*chain* (*S*::*nat* ⇒ *'a*::*cpo*); *range S* <| *x*⟧ ⟹ (⨆ *i*. *S i*) ⊑ *x*

by (*blast dest*: *cpo intro*: *lubI* [*THEN is-lub-lub*])

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

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

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

the ⊑ relation between two chains is preserved by their lubs

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

apply (*erule is-ub-thelub*)
done

the = relation between two chains is preserved by their lubs

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

more results about mono and = of lubs of chains

lemma *lub-mono2*:
  ⟦∃ *j*. ∀ *i*>*j*. *X i* = *Y i*; *chain* (*X*::*nat* ⇒ *'a*::*cpo*); *chain Y*⟧
    ⟹ (⨆ *i*. *X i*) ⊑ (⨆ *i*. *Y i*)
apply (*erule exE*)
apply (*subgoal-tac* (⨆ *i*. *X* (*i* + *Suc j*)) ⊑ (⨆ *i*. *Y* (*i* + *Suc j*)))
apply (*thin-tac* ∀ *i*>*j*. *X i* = *Y i*)
apply (*simp only*: *lub-range-shift*)
apply *simp*
done

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

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

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

lemma *diag-lub*:
  fixes *Y* :: *nat* ⇒ *nat* ⇒ *'a*::*cpo*
  assumes *1*: ⋀*j*. *chain* (λ*i*. *Y i j*)

assumes *2*: $\bigwedge i.\ chain\ (\lambda j.\ Y\ i\ j)$
shows $(\bigsqcup i.\ \bigsqcup j.\ Y\ i\ j) = (\bigsqcup i.\ Y\ i\ i)$
proof (*rule antisym-less*)
have *3*: *chain* $(\lambda i.\ Y\ i\ i)$
apply (*rule chainI*)
apply (*rule trans-less*)
apply (*rule chainE* [*OF 1*])
apply (*rule chainE* [*OF 2*])
done
have *4*: *chain* $(\lambda i.\ \bigsqcup j.\ Y\ i\ j)$
by (*rule ch2ch-lub* [*OF 1 2*])
show $(\bigsqcup i.\ \bigsqcup j.\ Y\ i\ j) \sqsubseteq (\bigsqcup i.\ Y\ i\ i)$
apply (*rule is-lub-thelub* [*OF 4*])
apply (*rule ub-rangeI*)
apply (*rule lub-mono3* [*rule-format*, *OF 2 3*])
apply (*rule exI*)
apply (*rule trans-less*)
apply (*rule chain-mono* [*OF 1 le-maxI1*])
apply (*rule chain-mono* [*OF 2 le-maxI2*])
done
show $(\bigsqcup i.\ Y\ i\ i) \sqsubseteq (\bigsqcup i.\ \bigsqcup j.\ Y\ i\ j)$
apply (*rule lub-mono* [*OF 3 4*])
apply (*rule is-ub-thelub* [*OF 2*])
done
qed

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

2.2 Pointed cpos

The class pcpo of pointed cpos

axclass *pcpo* < *cpo*
least: $\exists x.\ \forall y.\ x \sqsubseteq y$

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

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

derive the old rule minimal

lemma *UU-least*: $\forall z.\ \bot \sqsubseteq z$
apply (*unfold UU-def*)

apply (*rule theI'*)
apply (*rule ex-ex1I*)
apply (*rule least*)
apply (*blast intro*: *antisym-less*)
done

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

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

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

Addsimprocs [*UU-reorient-simproc*];
⟫

useful lemmas about $\bot$

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

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

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

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

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

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

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

lemma *notUU-I*: $[\![x \sqsubseteq y;\ x \neq \bot]\!] \Longrightarrow y \neq \bot$
by (*blast intro*: *UU-I*)

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

2.3 Chain-finite and flat cpos

further useful classes for HOLCF domains

axclass *finite-po* < *finite*, *po*

axclass *chfin* < *po*
chfin: $chain\ Y \Longrightarrow \exists n.\ max\text{-}in\text{-}chain\ n\ Y$

axclass *flat* < *pcpo*
ax-flat: $x \sqsubseteq y \Longrightarrow (x = \bot) \vee (x = y)$

finite partial orders are chain-finite and directed-complete

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

instance *finite-po* < *cpo*
apply *intro-classes*
apply (*drule directed-chain*)
apply (*drule directed-finiteD* [*OF - finite subset-refl*])
apply (*erule bexE*, *rule exI*)
apply (*erule* (*1*) *is-lub-maximal*)
done

some properties for chfin and flat

chfin types are cpo

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

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

flat types are chfin

instance *flat* < *chfin*
apply *intro-classes*
apply (*unfold max-in-chain-def*)
apply (*case-tac* $\forall i.\ Y\ i = \bot$)
apply *simp*
apply *simp*
apply (*erule exE*)
apply (*rule-tac x*=*i* **in** *exI*)
apply *clarify*
apply (*blast dest*: *chain-mono ax-flat*)
done

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

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

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

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

Discrete cpos

axclass *discrete-cpo* < *sq-ord*
  discrete-cpo [*simp*]: $x \sqsubseteq y \longleftrightarrow x = y$

instance *discrete-cpo* < *po*
by (*intro-classes*, *simp-all*)

In a discrete cpo, every chain is constant

lemma *discrete-chain-const*:
  assumes *S*: *chain* $(S::nat \Rightarrow 'a::discrete\text{-}cpo)$
  shows $\exists x.\ S = (\lambda i.\ x)$
proof (*intro exI ext*)
  fix $i :: nat$
  have *S 0* $\sqsubseteq$ *S i* **using** *S le0* **by** (*rule chain-mono*)
  hence *S 0* = *S i* **by** *simp*
  thus *S i* = *S 0* **by** (*rule sym*)
qed

instance *discrete-cpo* < *cpo*

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

lemmata for improved admissibility introdution rule

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

lemma *increasing-chain-adm-lemma*:
  $[\![$*chain* Y; $\forall i.\ P\ (Y\ i)$; $\bigwedge Y.\ [\![$*chain* Y; $\forall i.\ P\ (Y\ i)$;
    $\forall i.\ \exists j{>}i.\ Y\ i \neq Y\ j \wedge Y\ i \sqsubseteq Y\ j]\!] \Longrightarrow P\ (\bigsqcup i.\ Y\ i)]\!]$
      $\Longrightarrow P\ (\bigsqcup i.\ Y\ i)$
apply (*erule infinite-chain-adm-lemma*)
apply *assumption*
apply (*erule thin-rl*)
apply (*unfold finite-chain-def*)
apply (*unfold max-in-chain-def*)
apply (*fast dest*: *le-imp-less-or-eq elim*: *chain-mono-less*)
done

end

3 Cont: Continuity and monotonicity

theory *Cont*
imports *Pcpo*
begin

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

defaultsort *po*

3.1 Definitions

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

definition
contlub :: $('a{::}cpo \Rightarrow 'b{::}cpo) \Rightarrow bool$ — first cont. def **where**
$contlub\ f = (\forall\ Y.\ chain\ Y \longrightarrow f\ (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f\ (Y\ i)))$

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

lemma *contlubI*:
$[\![\bigwedge Y.\ chain\ Y \Longrightarrow f\ (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f\ (Y\ i))]\!] \Longrightarrow contlub\ f$
by (*simp add*: *contlub-def*)

lemma *contlubE*:
$[\![contlub\ f;\ chain\ Y]\!] \Longrightarrow f\ (\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ f\ (Y\ i))$
by (*simp add*: *contlub-def*)

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

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

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

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

3.2 $monofun\ f \wedge contlub\ f \equiv cont\ f$

monotone functions map chains to chains

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

monotone functions map upper bound to upper bounds

lemma *ub2ub-monofun*:

⟦*monofun f*; *range Y <| u*⟧ ⟹ *range* (λ*i*. *f* (*Y i*)) <| *f u*
apply (*rule ub-rangeI*)
apply (*erule monofunE*)
apply (*erule ub-rangeD*)
done

lemma *ub2ub-monofun'*:
⟦*monofun f*; *S <| u*⟧ ⟹ *f ' S <| f u*
apply (*rule ub-imageI*)
apply (*erule monofunE*)
apply (*erule* (*1*) *is-ubD*)
done

monotone functions map directed sets to directed sets

lemma *dir2dir-monofun*:
assumes *f*: *monofun f*
assumes *S*: *directed S*
shows *directed* (*f ' S*)
proof (*rule directedI*)
from *directedD1* [*OF S*]
obtain *x* **where** *x* ∈ *S* **..**
hence *f x* ∈ *f ' S* **by** *simp*
thus ∃ *x*. *x* ∈ *f ' S* **..**
next
fix *x* **assume** *x* ∈ *f ' S*
then obtain *a* **where** *x*: *x* = *f a* **and** *a*: *a* ∈ *S* **..**
fix *y* **assume** *y* ∈ *f ' S*
then obtain *b* **where** *y*: *y* = *f b* **and** *b*: *b* ∈ *S* **..**
from *directedD2* [*OF S a b*]
obtain *c* **where** *c* ∈ *S* **and** *a* ⊑ *c* ∧ *b* ⊑ *c* **..**
hence *f c* ∈ *f ' S* **and** *x* ⊑ *f c* ∧ *y* ⊑ *f c*
using *monofunE* [*OF f*] *x y* **by** *simp-all*
thus ∃ *z*∈*f ' S*. *x* ⊑ *z* ∧ *y* ⊑ *z* **..**
qed

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

lemma *monocontlub2cont*: ⟦*monofun f*; *contlub f*⟧ ⟹ *cont f*
apply (*rule contI*)
apply (*rule thelubE*)
apply (*erule* (*1*) *ch2ch-monofun*)
apply (*erule* (*1*) *contlubE* [*symmetric*])
done

first a lemma about binary chains

lemma *binchain-cont*:
⟦*cont f*; *x* ⊑ *y*⟧ ⟹ *range* (λ*i*::*nat*. *f* (*if i* = *0 then x else y*)) <<| *f y*
apply (*subgoal-tac f* (⨆ *i*::*nat*. *if i* = *0 then x else y*) = *f y*)
apply (*erule subst*)
apply (*erule contE*)

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

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

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

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

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

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

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

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

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

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

3.3 Continuity of basic functions

The identity function is continuous

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

constant functions are continuous

lemma *cont-const*: *cont* ($\lambda x.\ c$)
apply (*rule contI*)
apply (*rule lub-const*)
done

if-then-else is continuous

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

3.4 Finite chains and flat pcpos

monotone functions map finite chains to finite chains

lemma *monofun-finch2finch*:
  $[\![$*monofun f*; *finite-chain Y*$]\!] \Longrightarrow$ *finite-chain* ($\lambda n.$ *f* (*Y n*))
apply (*unfold finite-chain-def*)
apply (*simp add*: *ch2ch-monofun*)
apply (*force simp add*: *max-in-chain-def*)
done

The same holds for continuous functions

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

lemma *chfindom-monofun2cont*: *monofun f* $\Longrightarrow$ *cont* (*f*::'*a*::*chfin* $\Rightarrow$ '*b*::*cpo*)
apply (*rule monocontlub2cont*)
apply *assumption*
apply (*rule contlubI*)
apply (*frule chfin2finch*)
apply (*clarsimp simp add*: *finite-chain-def*)
apply (*subgoal-tac max-in-chain i* ($\lambda i.$ *f* (*Y i*)))
apply (*simp add*: *maxinch-is-thelub ch2ch-monofun*)
apply (*force simp add*: *max-in-chain-def*)
done

some properties of flat

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

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

functions with discrete domain

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

end

4 Ffun: Class instances for the full function space

theory *Ffun*
imports *Cont*
begin

4.1 Full function space is a partial order

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

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

instance ..
end

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

make the symbol << accessible for type fun

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

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

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

4.2 Full function space is chain complete

function application is monotone

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

chains of functions yield chains in the po range

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

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

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

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

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

lemma *is-lub-lambda*:
assumes f: $\bigwedge x.$ *range* $(\lambda i.\ Y\ i\ x) <<|\ f\ x$
shows *range* $Y <<|\ f$
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*rule less-fun-ext*)
apply (*rule is-ub-lub* [*OF f*])
apply (*rule less-fun-ext*)
apply (*rule is-lub-lub* [*OF f*])
apply (*erule ub2ub-fun*)
done

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

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

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

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

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

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

chain-finite function spaces

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

lemma *maxinch-mono*:
⟦*max-in-chain i Y*; *i* ≤ *j*⟧ ⟹ *max-in-chain j Y*
unfolding *max-in-chain-def*
proof (*intro allI impI*)
fix *k*
assume *Y*: ∀ *n*≥*i*. *Y i* = *Y n*
assume *ij*: *i* ≤ *j*
assume *jk*: *j* ≤ *k*
from *ij jk* **have** *ik*: *i* ≤ *k* **by** *simp*
from *Y ij* **have** *Yij*: *Y i* = *Y j* **by** *simp*
from *Y ik* **have** *Yik*: *Y i* = *Y k* **by** *simp*
from *Yij Yik* **show** *Y j* = *Y k* **by** *auto*
qed

instance *fun* :: (*finite*, *chfin*) *chfin*
proof
fix *Y* :: *nat* ⇒ 'a ⇒ 'b
let *?n* = λ*x*. *LEAST n*. *max-in-chain n* (λ*i*. *Y i x*)
assume *chain Y*
hence ⋀*x*. *chain* (λ*i*. *Y i x*)
by (*rule ch2ch-fun*)
hence ⋀*x*. ∃ *n*. *max-in-chain n* (λ*i*. *Y i x*)
by (*rule chfin*)
hence ⋀*x*. *max-in-chain* (*?n x*) (λ*i*. *Y i x*)
by (*rule LeastI-ex*)
hence ⋀*x*. *max-in-chain* (*Max* (*range ?n*)) (λ*i*. *Y i x*)

by (*rule maxinch-mono* [*OF - Max-ge*], *simp-all*)
hence *max-in-chain* (*Max* (*range ?n*)) *Y*
by (*rule maxinch2maxinch-lambda*)
thus ∃ *n*. *max-in-chain n Y* **..**
qed

4.3 Full function space is pointed

lemma *minimal-fun*: (λ*x*. ⊥) ⊑ *f*
by (*simp add*: *less-fun-def*)

lemma *least-fun*: ∃ *x*::'*a*::*type* ⇒ '*b*::*pcpo*. ∀ *y*. *x* ⊑ *y*
apply (*rule-tac x* = λ*x*. ⊥ **in** *exI*)
apply (*rule minimal-fun* [*THEN allI*])
done

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

for compatibility with old HOLCF-Version

lemma *inst-fun-pcpo*: ⊥ = (λ*x*. ⊥)
by (*rule minimal-fun* [*THEN UU-I*, *symmetric*])

function application is strict in the left argument

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

The following results are about application for functions in '*a* ⇒ '*b*

lemma *monofun-fun-fun*: *f* ⊑ *g* ⟹ *f x* ⊑ *g x*
by (*simp add*: *less-fun-def*)

lemma *monofun-fun-arg*: ⟦*monofun f*; *x* ⊑ *y*⟧ ⟹ *f x* ⊑ *f y*
by (*rule monofunE*)

lemma *monofun-fun*: ⟦*monofun f*; *monofun g*; *f* ⊑ *g*; *x* ⊑ *y*⟧ ⟹ *f x* ⊑ *g y*
by (*rule trans-less* [*OF monofun-fun-arg monofun-fun-fun*])

4.4 Propagation of monotonicity and continuity

the lub of a chain of monotone functions is monotone

lemma *monofun-lub-fun*:
⟦*chain* (*F*::*nat* ⇒ '*a* ⇒ '*b*::*cpo*); ∀ *i*. *monofun* (*F i*)⟧
⟹ *monofun* (⨆ *i*. *F i*)
apply (*rule monofunI*)
apply (*simp add*: *thelub-fun*)
apply (*rule lub-mono*)
apply (*erule ch2ch-fun*)
apply (*erule ch2ch-fun*)

apply (*simp add*: *monofunE*)
done

the lub of a chain of continuous functions is continuous

declare *range-composition* [*simp del*]

lemma *contlub-lub-fun*:
⟦*chain F*; ∀ *i*. *cont* (*F i*)⟧ ⟹ *contlub* (⨆ *i*. *F i*)
apply (*rule contlubI*)
apply (*simp add*: *thelub-fun*)
apply (*simp add*: *cont2contlubE*)
apply (*rule ex-lub*)
apply (*erule ch2ch-fun*)
apply (*simp add*: *ch2ch-cont*)
done

lemma *cont-lub-fun*:
⟦*chain F*; ∀ *i*. *cont* (*F i*)⟧ ⟹ *cont* (⨆ *i*. *F i*)
apply (*rule monocontlub2cont*)
apply (*erule monofun-lub-fun*)
apply (*simp add*: *cont2mono*)
apply (*erule* (*1*) *contlub-lub-fun*)
done

lemma *cont2cont-lub*:
⟦*chain F*; ⋀*i*. *cont* (*F i*)⟧ ⟹ *cont* (λ*x*. ⨆ *i*. *F i x*)
by (*simp add*: *thelub-fun* [*symmetric*] *cont-lub-fun*)

lemma *mono2mono-fun*: *monofun f* ⟹ *monofun* (λ*x*. *f x y*)
apply (*rule monofunI*)
apply (*erule* (*1*) *monofun-fun-arg* [*THEN monofun-fun-fun*])
done

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

Note (λ*x*. λ*y*. *f x y*) = *f*

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

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

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

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

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

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

lemma *cont2contlub-app*:
  $[\![cont\ f;\ \forall x.\ cont\ (f\ x);\ cont\ t]\!] \Longrightarrow contlub\ (\lambda x.\ (f\ x)\ (t\ x))$
apply (*rule contlubI*)
apply (*subgoal-tac chain* $(\lambda i.\ f\ (Y\ i))$)
apply (*subgoal-tac chain* $(\lambda i.\ t\ (Y\ i))$)
apply (*simp add*: *cont2contlubE thelub-fun*)
apply (*rule diag-lub*)
apply (*erule ch2ch-fun*)
apply (*drule spec*)
apply (*erule* (*1*) *ch2ch-cont*)
apply (*erule* (*1*) *ch2ch-cont*)
apply (*erule* (*1*) *ch2ch-cont*)
done

lemma *cont2cont-app*:
  $[\![cont\ f;\ \forall x.\ cont\ (f\ x);\ cont\ t]\!] \Longrightarrow cont\ (\lambda x.\ (f\ x)\ (t\ x))$
by (*blast intro*: *monocontlub2cont mono2mono-app cont2mono cont2contlub-app*)

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

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

end

5 Adm: Admissibility and compactness

theory *Adm*
imports *Ffun*
begin

defaultsort *cpo*

5.1 Definitions

definition
adm :: ('*a*::*cpo* ⇒ *bool*) ⇒ *bool* **where**
adm P = (∀ *Y*. *chain Y* ⟶ (∀ *i*. *P* (*Y i*)) ⟶ *P* (⨆ *i*. *Y i*))

lemma *admI*:
(⋀*Y*. ⟦*chain Y*; ∀ *i*. *P* (*Y i*)⟧ ⟹ *P* (⨆ *i*. *Y i*)) ⟹ *adm P*
unfolding *adm-def* **by** *fast*

lemma *admD*: ⟦*adm P*; *chain Y*; ⋀*i*. *P* (*Y i*)⟧ ⟹ *P* (⨆ *i*. *Y i*)
unfolding *adm-def* **by** *fast*

lemma *triv-admI*: ∀ *x*. *P x* ⟹ *adm P*
by (*rule admI*, *erule spec*)

improved admissibility introduction

lemma *admI2*:
(⋀*Y*. ⟦*chain Y*; ∀ *i*. *P* (*Y i*); ∀ *i*. ∃ *j*>*i*. *Y i* ≠ *Y j* ∧ *Y i* ⊑ *Y j*⟧
⟹ *P* (⨆ *i*. *Y i*)) ⟹ *adm P*
apply (*rule admI*)
apply (*erule* (*1*) *increasing-chain-adm-lemma*)
apply *fast*
done

5.2 Admissibility on chain-finite types

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

lemma *adm-chfin*: *adm* (*P*::'*a*::*chfin* ⇒ *bool*)
by (*rule admI*, *frule chfin*, *auto simp add*: *maxinch-is-thelub*)

5.3 Admissibility of special formulae and propagation

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

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

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

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

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

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

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

lemma *adm-disj-lemma3*:
$[\![chain\ (Y{::}nat \Rightarrow 'a{::}cpo);\ \forall i.\ \exists j{\geq}i.\ P\ (Y\ j)]\!] \Longrightarrow$
$(\bigsqcup i.\ Y\ i) = (\bigsqcup i.\ Y\ (LEAST\ j.\ i \leq j \wedge P\ (Y\ j)))$
apply (*frule* (*1*) *adm-disj-lemma1*)
apply (*rule antisym-less*)
apply (*rule lub-mono*, *assumption+*)
apply (*erule chain-mono*)
apply (*simp add*: *adm-disj-lemma2*)
apply (*rule lub-range-mono*, *fast*, *assumption+*)
done

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

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

apply (*rule-tac x=max a b* **in** *exI*)
apply *simp*
done

lemma *adm-disj*: ⟦*adm P*; *adm Q*⟧ ⟹ *adm* (λ*x*. *P x* ∨ *Q x*)
apply (*rule admI*)
apply (*erule adm-disj-lemma5* [*THEN disjE*])
apply (*erule* (*2*) *adm-disj-lemma4* [*THEN disjI1*])
apply (*erule* (*2*) *adm-disj-lemma4* [*THEN disjI2*])
done

lemma *adm-imp*: ⟦*adm* (λ*x*. ¬ *P x*); *adm Q*⟧ ⟹ *adm* (λ*x*. *P x* ⟶ *Q x*)
by (*subst imp-conv-disj*, *rule adm-disj*)

lemma *adm-iff*:
⟦*adm* (λ*x*. *P x* ⟶ *Q x*); *adm* (λ*x*. *Q x* ⟶ *P x*)⟧
⟹ *adm* (λ*x*. *P x* = *Q x*)
by (*subst iff-conv-conj-imp*, *rule adm-conj*)

lemma *adm-not-conj*:
⟦*adm* (λ*x*. ¬ *P x*); *adm* (λ*x*. ¬ *Q x*)⟧ ⟹ *adm* (λ*x*. ¬ (*P x* ∧ *Q x*))
by (*simp add*: *adm-imp*)

admissibility and continuity

declare *range-composition* [*simp del*]

lemma *adm-less*: ⟦*cont u*; *cont v*⟧ ⟹ *adm* (λ*x*. *u x* ⊑ *v x*)
apply (*rule admI*)
apply (*simp add*: *cont2contlubE*)
apply (*rule lub-mono*)
apply (*erule* (*1*) *ch2ch-cont*)
apply (*erule* (*1*) *ch2ch-cont*)
apply (*erule spec*)
done

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

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

lemma *adm-not-less*: *cont t* ⟹ *adm* (λ*x*. ¬ *t x* ⊑ *u*)
apply (*rule admI*)
apply (*drule-tac x=0* **in** *spec*)

apply (*erule contrapos-nn*)
apply (*erule rev-trans-less*)
apply (*erule cont2mono* [*THEN monofunE*])
apply (*erule is-ub-thelub*)
done

5.4 Compactness

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

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

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

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

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

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

lemma *compact-imp-max-in-chain*:
⟦*chain Y*; *compact* (⨆ *i*. *Y i*)⟧ ⟹ ∃ *i*. *max-in-chain i Y*
apply (*drule* (*1*) *compactD2*, *simp*)
apply (*erule exE*, *rule-tac x*=*i* **in** *exI*)
apply (*rule max-in-chainI*)
apply (*rule antisym-less*)
apply (*erule* (*1*) *chain-mono*)
apply (*erule* (*1*) *trans-less* [*OF is-ub-thelub*])
done

admissibility and compactness

lemma *adm-compact-not-less*: ⟦*compact k*; *cont t*⟧ ⟹ *adm* (λ*x*. ¬ *k* ⊑ *t x*)
unfolding *compact-def* **by** (*rule adm-subst*)

lemma *adm-neq-compact*: ⟦*compact k*; *cont t*⟧ ⟹ *adm* (λ*x*. *t x* ≠ *k*)
by (*simp add*: *po-eq-conv adm-imp adm-not-less adm-compact-not-less*)

lemma *adm-compact-neq*: ⟦*compact k*; *cont t*⟧ ⟹ *adm* (λ*x*. *k* ≠ *t x*)
by (*simp add*: *po-eq-conv adm-imp adm-not-less adm-compact-not-less*)

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

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

Any upward-closed predicate is admissible.

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

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

end

6 Pcpodef: Subtypes of pcpos

theory *Pcpodef*
imports *Adm*
uses (*Tools/pcpodef-package.ML*)
begin

6.1 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 *typedef-po*:
fixes *Abs* :: $'a$::*po* $\Rightarrow$ $'b$::*sq-ord*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* $\sqsubseteq \equiv \lambda x\ y.$ *Rep x* $\sqsubseteq$ *Rep y*
shows *OFCLASS*($'b$, *po-class*)
apply (*intro-classes*, *unfold less*)
apply (*rule refl-less*)
apply (*erule* (*1*) *trans-less*)
apply (*rule type-definition.Rep-inject* [*OF type*, *THEN iffD1*])
apply (*erule* (*1*) *antisym-less*)
done

6.2 Proving a subtype is finite

context *type-definition*
begin

lemma *Abs-image*:
shows *Abs ‘ A = UNIV*
proof
show *Abs ‘ A <= UNIV* **by** *simp*
show *UNIV <= Abs ‘ A*
proof
fix *x*
have *x = Abs (Rep x)* **by** (*rule Rep-inverse* [*symmetric*])
thus *x : Abs ‘ A* **using** *Rep* **by** (*rule image-eqI*)
qed
qed

lemma *finite-UNIV*: *finite A ⟹ finite (UNIV :: 'b set)*
proof −
assume *finite A*
hence *finite (Abs ‘ A)* **by** (*rule finite-imageI*)
thus *finite (UNIV :: 'b set)* **by** (*simp only*: *Abs-image*)
qed

end

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

6.3 Proving a subtype is chain-finite

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

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

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

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

6.4 Proving a subtype is complete

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

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

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

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

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

6.4.1 Continuity of *Rep* and *Abs*

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

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

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

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

theorem *typedef-cont-Abs*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
fixes *f* :: *'c*::*cpo* ⇒ *'a*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
and *f-in-A*: ⋀*x*. *f x* ∈ *A*
and *cont-f*: *cont f*
shows *cont* (λ*x*. *Abs* (*f x*))
apply (*rule contI*)
apply (*rule typedef-is-lubI* [*OF less*])
apply (*simp only*: *type-definition.Abs-inverse* [*OF type f-in-A*])
apply (*erule cont-f* [*THEN contE*])
done

6.5 Proving subtype elements are compact

theorem *typedef-compact*:

fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *adm*: *adm* (λ*x*. *x* ∈ *A*)
shows *compact* (*Rep k*) ⟹ *compact k*
proof (*unfold compact-def*)
have *cont-Rep*: *cont Rep*
by (*rule typedef-cont-Rep* [*OF type less adm*])
assume *adm* (λ*x*. ¬ *Rep k* ⊑ *x*)
with *cont-Rep* **have** *adm* (λ*x*. ¬ *Rep k* ⊑ *Rep x*) **by** (*rule adm-subst*)
thus *adm* (λ*x*. ¬ *k* ⊑ *x*) **by** (*unfold less*)
qed

6.6 Proving a subtype is pointed

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

theorem *typedef-pcpo-generic*:
fixes *Abs* :: *'a*::*cpo* ⇒ *'b*::*cpo*
assumes *type*: *type-definition Rep Abs A*
and *less*: *op* ⊑ ≡ λ*x y*. *Rep x* ⊑ *Rep y*
and *z-in-A*: *z* ∈ *A*
and *z-least*: ⋀*x*. *x* ∈ *A* ⟹ *z* ⊑ *x*
shows *OFCLASS*(*'b*, *pcpo-class*)
apply (*intro-classes*)
apply (*rule-tac x*=*Abs z* **in** *exI*, *rule allI*)
apply (*unfold less*)
apply (*subst type-definition.Abs-inverse* [*OF type z-in-A*])
apply (*rule z-least* [*OF type-definition.Rep* [*OF type*]])
done

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

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

6.6.1 Strictness of *Rep* and *Abs*

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

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

and *less*: $op \sqsubseteq \equiv \lambda x\ y.\ Rep\ x \sqsubseteq Rep\ y$
and *UU-in-A*: $\bot \in A$
shows $Abs\ \bot = \bot$
apply (*rule UU-I*, *unfold less*)
apply (*simp add*: *type-definition.Abs-inverse* [*OF type UU-in-A*])
done

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

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

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

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

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

6.7 Proving a subtype is flat

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

6.8 HOLCF type definition package

use *Tools/pcpodef-package.ML*

end

7 Cfun: The type of continuous functions

theory *Cfun*
imports *Pcpodef Ffun*
uses (*Tools/cont-proc.ML*)
begin

defaultsort *cpo*

7.1 Definition of continuous function type

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

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

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

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

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

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

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

7.2 Syntax for continuous lambda abstraction

syntax *-cabs* :: *'a*

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

To avoid eta-contraction of body:

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

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

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

Syntax for nested abstractions

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

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

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

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

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

Dummy patterns for continuous abstraction

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

7.3 Continuous function space is pointed

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

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

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

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

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

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

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

function application is strict in its first argument

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

for compatibility with old HOLCF-Version

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

7.4 Basic properties of continuous functions

Beta-equality for continuous functions

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

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

Eta-equality for continuous functions

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

Extensionality for continuous functions

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

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

Extensionality wrt. ordering for continuous functions

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

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

Congruence for continuous function application

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

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

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

7.5 Continuity of application

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

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

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

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

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

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

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

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

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

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

monotonicity of application

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

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

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

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

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

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

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

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

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

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

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

lemma *cont-cfun*:
  $[\![chain\ F;\ chain\ Y]\!] \Longrightarrow range\ (\lambda i.\ F\ i{\cdot}(Y\ i))\ <<|\ (\bigsqcup i.\ F\ i){\cdot}(\bigsqcup i.\ Y\ i)$
apply (*rule thelubE*)
apply (*simp only*: *ch2ch-Rep-CFun*)
apply (*simp only*: *contlub-cfun*)
done

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

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

strictness

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

the lub of a chain of continous functions is monotone

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

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

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

the lub of a chain of cont. functions is continuous

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

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

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

lemma *thelub-cfun*: *chain F* $\Longrightarrow$ *lub* (*range F*) = (Λ *x*. $\bigsqcup$ *i*. *F i*·*x*)
by (*rule lub-cfun* [*THEN thelubI*])

7.6 Continuity simplification procedure

cont2cont lemma for *Rep-CFun*

lemma *cont2cont-Rep-CFun*:
⟦*cont f*; *cont t*⟧ $\Longrightarrow$ *cont* (λ*x*. (*f x*)·(*t x*))
by (*best intro*: *cont2cont-app2 cont-const cont-Rep-CFun cont-Rep-CFun2*)

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

lemma *cont2mono-LAM*:
assumes *p1*: !!*x*. *cont*(*c1 x*)
assumes *p2*: !!*y*. *monofun*(%*x*. *c1 x y*)
shows *monofun*(%*x*. *LAM y*. *c1 x y*)
apply (*rule monofunI*)
apply (*rule less-cfun-ext*)
apply (*simp add*: *p1*)
apply (*erule p2* [*THEN monofunE*])
done

cont2cont Lemma for $\lambda x.\ \Lambda\ y.\ c1\ x\ y$

lemma *cont2cont-LAM*:
assumes *p1*: !!*x*. *cont*(*c1 x*)
assumes *p2*: !!*y*. *cont*(%*x*. *c1 x y*)
shows *cont*(%*x*. *LAM y*. *c1 x y*)
apply (*rule cont-Abs-CFun*)
apply (*simp add*: *p1 CFun-def*)
apply (*simp add*: *p2 cont2cont-lambda*)
done

continuity simplification procedure

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

use *Tools/cont-proc.ML*
setup *ContProc.setup*

7.7 Miscellaneous

Monotonicity of *Abs-CFun*

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

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

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

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

7.8 Continuous injection-retraction pairs

Continuous retractions are strict.

lemma *retraction-strict*:
∀ *x*. *f*·(*g*·*x*) = *x* ⟹ *f*·⊥ = ⊥
apply (*rule UU-I*)
apply (*drule-tac x*=⊥ **in** *spec*)
apply (*erule subst*)
apply (*rule monofun-cfun-arg*)
apply (*rule minimal*)
done

lemma *injection-eq*:
∀ *x*. *f*·(*g*·*x*) = *x* ⟹ (*g*·*x* = *g*·*y*) = (*x* = *y*)
apply (*rule iffI*)
apply (*drule-tac f*=*f* **in** *cfun-arg-cong*)
apply *simp*
apply *simp*
done

lemma *injection-less*:
∀ *x*. *f*·(*g*·*x*) = *x* ⟹ (*g*·*x* ⊑ *g*·*y*) = (*x* ⊑ *y*)
apply (*rule iffI*)
apply (*drule-tac f*=*f* **in** *monofun-cfun-arg*)
apply *simp*
apply (*erule monofun-cfun-arg*)
done

lemma *injection-defined-rev*:
⟦∀ *x*. *f*·(*g*·*x*) = *x*; *g*·*z* = ⊥⟧ ⟹ *z* = ⊥
apply (*drule-tac f*=*f* **in** *cfun-arg-cong*)
apply (*simp add*: *retraction-strict*)
done

lemma *injection-defined*:
⟦∀ *x*. *f*·(*g*·*x*) = *x*; *z* ≠ ⊥⟧ ⟹ *g*·*z* ≠ ⊥

by (*erule contrapos-nn*, *rule injection-defined-rev*)

propagation of flatness and chain-finiteness by retractions

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

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

a result about functions with flat codomain

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

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

7.9 Identity and composition

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

definition
$cfcomp :: ('b \to 'c) \to ('a \to 'b) \to 'a \to 'c$ **where**

oo-def: $cfcomp = (\Lambda\ f\ g\ x.\ f{\cdot}(g{\cdot}x))$

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

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

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

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

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

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

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

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

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

7.10 Strictified functions

defaultsort *pcpo*

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

results about strictify

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

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

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

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

lemma *strictify-conv-if*: *strictify*$\cdot f \cdot x$ = (*if* $x = \bot$ *then* $\bot$ *else* $f{\cdot}x$)
by (*unfold strictify-def*, *simp add*: *cont-strictify1 cont-strictify2*)

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

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

7.11 Continuous let-bindings

definition
CLet :: $'a \to ('a \to 'b) \to 'b$ **where**
CLet = (Λ s f. $f{\cdot}s$)

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

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

end

8 Bifinite: Bifinite domains and approximation

theory *Bifinite*
imports *Cfun*
begin

8.1 Omega-profinite and bifinite domains

class *profinite* = *cpo* +
  fixes *approx* :: *nat* ⇒ *'a* → *'a*
  assumes *chain-approx-app*: *chain* (λ*i*. *approx i*·*x*)
  assumes *lub-approx-app* [*simp*]: (⨆ *i*. *approx i*·*x*) = *x*
  assumes *approx-idem*: *approx i*·(*approx i*·*x*) = *approx i*·*x*
  assumes *finite-fixes-approx*: *finite* {*x*. *approx i*·*x* = *x*}

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

lemma *finite-range-imp-finite-fixes*:
  finite {*x*. ∃ *y*. *x* = *f y*} ⟹ *finite* {*x*. *f x* = *x*}
apply (*subgoal-tac* {*x*. *f x* = *x*} ⊆ {*x*. ∃ *y*. *x* = *f y*})
apply (*erule* (*1*) *finite-subset*)
apply (*clarify*, *erule subst*, *rule exI*, *rule refl*)
done

lemma *chain-approx* [*simp*]:
  chain (*approx* :: *nat* ⇒ *'a*::*profinite* → *'a*)
apply (*rule chainI*)
apply (*rule less-cfun-ext*)
apply (*rule chainE*)
apply (*rule chain-approx-app*)
done

lemma *lub-approx* [*simp*]: (⨆ *i*. *approx i*) = (Λ(*x*::*'a*::*profinite*). *x*)
by (*rule ext-cfun*, *simp add*: *contlub-cfun-fun*)

lemma *approx-less*: *approx i*·*x* ⊑ (*x*::*'a*::*profinite*)
apply (*subgoal-tac approx i*·*x* ⊑ (⨆ *i*. *approx i*·*x*), *simp*)
apply (*rule is-ub-thelub*, *simp*)
done

lemma *approx-strict* [*simp*]: *approx i*·(⊥::*'a*::*bifinite*) = ⊥
by (*rule UU-I*, *rule approx-less*)

lemma *approx-approx1*:
  i ≤ *j* ⟹ *approx i*·(*approx j*·*x*) = *approx i*·(*x*::*'a*::*profinite*)
apply (*rule antisym-less*)
apply (*rule monofun-cfun-arg* [*OF approx-less*])
apply (*rule sq-ord-eq-less-trans* [*OF approx-idem* [*symmetric*]])
apply (*rule monofun-cfun-arg*)
apply (*rule monofun-cfun-fun*)
apply (*erule chain-mono* [*OF chain-approx*])
done

lemma *approx-approx2*:
  j ≤ *i* ⟹ *approx i*·(*approx j*·*x*) = *approx j*·(*x*::*'a*::*profinite*)
apply (*rule antisym-less*)

apply (*rule approx-less*)
apply (*rule sq-ord-eq-less-trans* [*OF approx-idem* [*symmetric*]])
apply (*rule monofun-cfun-fun*)
apply (*erule chain-mono* [*OF chain-approx*])
done

lemma *approx-approx* [*simp*]:
  approx i·(*approx j*·*x*) = *approx* (*min i j*)·(*x*::'*a*::*profinite*)
apply (*rule-tac x*=*i* **and** *y*=*j* **in** *linorder-le-cases*)
apply (*simp add*: *approx-approx1 min-def*)
apply (*simp add*: *approx-approx2 min-def*)
done

lemma *idem-fixes-eq-range*:
  ∀ *x*. *f* (*f x*) = *f x* ⟹ {*x*. *f x* = *x*} = {*y*. ∃ *x*. *y* = *f x*}
by (*auto simp add*: *eq-sym-conv*)

lemma *finite-approx*: *finite* {*y*::'*a*::*profinite*. ∃ *x*. *y* = *approx n*·*x*}
using *finite-fixes-approx* **by** (*simp add*: *idem-fixes-eq-range*)

lemma *finite-range-approx*:
  finite (*range* (λ*x*::'*a*::*profinite*. *approx n*·*x*))
by (*simp add*: *image-def finite-approx*)

lemma *compact-approx* [*simp*]:
  fixes *x* :: '*a*::*profinite*
  shows *compact* (*approx n*·*x*)
proof (*rule compactI2*)
  fix *Y*::*nat* ⇒ '*a*
  assume *Y*: *chain Y*
  have *finite-chain* (λ*i*. *approx n*·(*Y i*))
  proof (*rule finite-range-imp-finch*)
    show *chain* (λ*i*. *approx n*·(*Y i*))
      using *Y* **by** *simp*
    have *range* (λ*i*. *approx n*·(*Y i*)) ⊆ {*x*. *approx n*·*x* = *x*}
      by *clarsimp*
    thus *finite* (*range* (λ*i*. *approx n*·(*Y i*)))
      using *finite-fixes-approx* **by** (*rule finite-subset*)
  qed
  hence ∃ *j*. (⨆ *i*. *approx n*·(*Y i*)) = *approx n*·(*Y j*)
    by (*simp add*: *finite-chain-def maxinch-is-thelub Y*)
  then obtain *j* **where** *j*: (⨆ *i*. *approx n*·(*Y i*)) = *approx n*·(*Y j*) **..**

  assume *approx n*·*x* ⊑ (⨆ *i*. *Y i*)
  hence *approx n*·(*approx n*·*x*) ⊑ *approx n*·(⨆ *i*. *Y i*)
    by (*rule monofun-cfun-arg*)
  hence *approx n*·*x* ⊑ (⨆ *i*. *approx n*·(*Y i*))
    by (*simp add*: *contlub-cfun-arg Y*)
  hence *approx n*·*x* ⊑ *approx n*·(*Y j*)

```
    using j by simp
  hence approx n·x ⊑ Y j
    using approx-less by (rule trans-less)
  thus ∃j. approx n·x ⊑ Y j ..
qed

lemma bifinite-compact-eq-approx:
  fixes x :: 'a::profinite
  assumes x: compact x
  shows ∃i. approx i·x = x
proof −
  have chain: chain (λi. approx i·x) by simp
  have less: x ⊑ (⨆i. approx i·x) by simp
  obtain i where i: x ⊑ approx i·x
    using compactD2 [OF x chain less] ..
  with approx-less have approx i·x = x
    by (rule antisym-less)
  thus ∃i. approx i·x = x ..
qed

lemma bifinite-compact-iff:
  compact (x::'a::profinite) = (∃n. approx n·x = x)
 apply (rule iffI)
  apply (erule bifinite-compact-eq-approx)
 apply (erule exE)
 apply (erule subst)
 apply (rule compact-approx)
done

lemma approx-induct:
  assumes adm: adm P and P: ⋀n x. P (approx n·x)
  shows P (x::'a::profinite)
proof −
  have P (⨆n. approx n·x)
    by (rule admD [OF adm], simp, simp add: P)
  thus P x by simp
qed

lemma bifinite-less-ext:
  fixes x y :: 'a::profinite
  shows (⋀i. approx i·x ⊑ approx i·y) ⟹ x ⊑ y
apply (subgoal-tac (⨆i. approx i·x) ⊑ (⨆i. approx i·y), simp)
apply (rule lub-mono, simp, simp, simp)
done
```

8.2 Instance for continuous function space

```
lemma finite-range-lemma:
  fixes h :: 'a::cpo → 'b::cpo
```

```
  fixes k :: 'c::cpo → 'd::cpo
  shows ⟦finite {y. ∃x. y = h·x}; finite {y. ∃x. y = k·x}⟧
    ⟹ finite {g. ∃f. g = (Λ x. k·(f·(h·x)))}
 apply (rule-tac f=λg. {(h·x, y) |x y. y = g·x} in finite-imageD)
  apply (rule-tac B=Pow ({y. ∃x. y = h·x} × {y. ∃x. y = k·x})
         in finite-subset)
   apply (rule image-subsetI)
   apply (clarsimp, fast)
  apply simp
 apply (rule inj-onI)
 apply (clarsimp simp add: expand-set-eq)
 apply (rule ext-cfun, simp)
 apply (drule-tac x=h·x in spec)
 apply (drule-tac x=k·(f·(h·x)) in spec)
 apply (drule iffD1, fast)
 apply clarsimp
done

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

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

instance
 apply (intro-classes, unfold approx-cfun-def)
    apply simp
   apply (simp add: lub-distribs eta-cfun)
  apply simp
 apply simp
 apply (rule finite-range-imp-finite-fixes)
 apply (intro finite-range-lemma finite-approx)
done

end

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

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

end
```

9 Cprod: The cpo of cartesian products

```
theory Cprod
imports Bifinite
```

begin

defaultsort *cpo*

9.1 Type *unit* is a pcpo

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

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

instance ..
end

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

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

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

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

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

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

9.2 Product type is a partial order

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

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

instance ..
end

instance $*$ :: (*po*, *po*) *po*
proof
fix $x :: 'a \times 'b$
show $x \sqsubseteq x$
unfolding *less-cprod-def* **by** *simp*
next

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

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

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

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

Pair (-,-) is monotone in both arguments

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

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

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

fst and *snd* are monotone

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

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

9.4 Product type is a cpo

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

lemma *lub-cprod*:

fixes $S :: nat \Rightarrow ('a::cpo \times 'b::cpo)$
assumes S: *chain* S
shows *range* $S <<| (\bigsqcup i.\ fst\ (S\ i), \bigsqcup i.\ snd\ (S\ i))$
proof −
have *chain* $(\lambda i.\ fst\ (S\ i))$
using *monofun-fst* S **by** (*rule ch2ch-monofun*)
hence *1*: *range* $(\lambda i.\ fst\ (S\ i)) <<| (\bigsqcup i.\ fst\ (S\ i))$
by (*rule cpo-lubI*)
have *chain* $(\lambda i.\ snd\ (S\ i))$
using *monofun-snd* S **by** (*rule ch2ch-monofun*)
hence *2*: *range* $(\lambda i.\ snd\ (S\ i)) <<| (\bigsqcup i.\ snd\ (S\ i))$
by (*rule cpo-lubI*)
show *range* $S <<| (\bigsqcup i.\ fst\ (S\ i), \bigsqcup i.\ snd\ (S\ i))$
using *is-lub-Pair* [*OF 1 2*] **by** *simp*
qed

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

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

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

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

9.5 Product type is pointed

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

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

lemma *inst-cprod-pcpo*: $\bot = (\bot, \bot)$

by (*rule minimal-cprod* [*THEN UU-I*, *symmetric*])

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

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

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

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

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

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

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

9.7 Continuous versions of constants

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

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

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

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

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

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

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

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

9.8 Convert all lemmas to the continuous versions

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

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

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

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

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

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

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

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

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

by *simp*

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

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

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

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

lemma *cfst-strict* [*simp*]: $cfst{\cdot}\bot = \bot$
unfolding *inst-cprod-pcpo2* **by** (*rule cfst-cpair*)

lemma *csnd-strict* [*simp*]: $csnd{\cdot}\bot = \bot$
unfolding *inst-cprod-pcpo2* **by** (*rule csnd-cpair*)

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

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

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

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

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

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

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

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

lemma *compact-cpair*: $[\![compact\ x;\ compact\ y]\!] \Longrightarrow compact <x,\ y>$
by (*rule compactI*, *simp add*: *less-cprod*)

lemma *compact-cpair-iff* [*simp*]: *compact* $<x,\ y> = (compact\ x \wedge compact\ y)$
apply (*safe intro*!: *compact-cpair*)
apply (*drule compact-cfst*, *simp*)

apply (*drule compact-csnd*, *simp*)
done

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

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

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

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

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

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

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

9.9 Product type is a bifinite domain

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

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

instance proof
fix i :: *nat* **and** x :: $'a \times 'b$
show *chain* ($\lambda i.\ approx\ i{\cdot}x$)
unfolding *approx-cprod-def* **by** *simp*
show ($\bigsqcup i.\ approx\ i{\cdot}x$) = x
unfolding *approx-cprod-def*
by (*simp add*: *lub-distribs eta-cfun*)
show $approx\ i{\cdot}(approx\ i{\cdot}x) = approx\ i{\cdot}x$
unfolding *approx-cprod-def csplit-def* **by** *simp*
have $\{x::'a \times 'b.\ approx\ i{\cdot}x = x\} \subseteq$

```
      {x::'a. approx i·x = x} × {x::'b. approx i·x = x}
  unfolding approx-cprod-def
  by (clarsimp simp add: pair-eq-cpair)
thus finite {x::'a × 'b. approx i·x = x}
  by (rule finite-subset,
      intro finite-cartesian-product finite-fixes-approx)
qed

end

instance ∗ :: (bifinite, bifinite) bifinite ..

lemma approx-cpair [simp]:
  approx i·⟨x, y⟩ = ⟨approx i·x, approx i·y⟩
unfolding approx-cprod-def by simp

lemma cfst-approx: cfst·(approx i·p) = approx i·(cfst·p)
by (cases p rule: cprodE, simp)

lemma csnd-approx: csnd·(approx i·p) = approx i·(csnd·p)
by (cases p rule: cprodE, simp)

end
```

10 Sprod: The type of strict products

```
theory Sprod
imports Cprod
begin

defaultsort pcpo
```

10.1 Definition of strict product type

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

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

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

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

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

10.2 Definitions of constants

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

definition
ssnd :: ('*a* ⊗ '*b*) → '*b* **where**
ssnd = (Λ *p*. *csnd*·(*Rep-Sprod p*))

definition
spair :: '*a* → '*b* → ('*a* ⊗ '*b*) **where**
spair = (Λ *a b*. *Abs-Sprod*
<*strictify*·(Λ *b*. *a*)·*b*, *strictify*·(Λ *a*. *b*)·*a*>)

definition
ssplit :: ('*a* → '*b* → '*c*) → ('*a* ⊗ '*b*) → '*c* **where**
ssplit = (Λ *f*. *strictify*·(Λ *p*. *f*·(*sfst*·*p*)·(*ssnd*·*p*)))

syntax
@*stuple* :: ['*a*, *args*] => '*a* ⊗ '*b* ((*1* '(:-,/ -:')))
translations
(:*x*, *y*, *z*:) == (:*x*, (:*y*, *z*:):)
(:*x*, *y*:) == *CONST spair*·*x*·*y*

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

10.3 Case analysis

lemma *Rep-Sprod-spair*:
Rep-Sprod (:*a*, *b*:) = <*strictify*·(Λ *b*. *a*)·*b*, *strictify*·(Λ *a*. *b*)·*a*>
unfolding *spair-def*
by (*simp add*: *cont-Abs-Sprod Abs-Sprod-inverse spair-lemma*)

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

lemma *Exh-Sprod2*:
z = ⊥ ∨ (∃ *a b*. *z* = (:*a*, *b*:) ∧ *a* ≠ ⊥ ∧ *b* ≠ ⊥)
apply (*insert Rep-Sprod* [*of z*])
apply (*simp add*: *Rep-Sprod-simps eq-cprod*)
apply (*simp add*: *Sprod-def*)
apply (*erule disjE*, *simp*)

apply (*simp add*: *strictify-conv-if*)
apply *fast*
done

lemma *sprodE* [*cases type*: $**$]:
$[\![p = \bot \Longrightarrow Q;\ \bigwedge x\ y.\ [\![p = (:x,\ y:);\ x \neq \bot;\ y \neq \bot]\!] \Longrightarrow Q]\!] \Longrightarrow Q$
by (*cut-tac z=p* **in** *Exh-Sprod2*, *auto*)

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

10.4 Properties of *spair*

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

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

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

lemma *spair-less-iff*:
$((:a,\ b:) \sqsubseteq (:c,\ d:)) = (a = \bot \vee b = \bot \vee (a \sqsubseteq c \wedge b \sqsubseteq d))$
by (*simp add*: *Rep-Sprod-simps strictify-conv-if*)

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

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

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

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

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

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

lemma *spair-inject*:

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

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

10.5 Properties of *sfst* and *ssnd*

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

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

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

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

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

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

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

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

lemma *surjective-pairing-Sprod2*: $(:sfst{\cdot}p,\ ssnd{\cdot}p:) = p$
by (*cases p*, *simp-all*)

lemma *less-sprod*: $x \sqsubseteq y = (sfst{\cdot}x \sqsubseteq sfst{\cdot}y \wedge ssnd{\cdot}x \sqsubseteq ssnd{\cdot}y)$
apply (*simp add*: *less-Sprod-def sfst-def ssnd-def cont-Rep-Sprod*)
apply (*rule less-cprod*)
done

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

lemma *spair-less*:
$[\![x \neq \bot;\ y \neq \bot]\!] \Longrightarrow (:x,\ y:) \sqsubseteq (:a,\ b:) = (x \sqsubseteq a \wedge y \sqsubseteq b)$
apply (*cases* $a = \bot$, *simp*)
apply (*cases* $b = \bot$, *simp*)
apply (*simp add*: *less-sprod*)
done

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

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

10.6 Compactness

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

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

lemma *compact-spair*: $[\![compact\ x;\ compact\ y]\!] \Longrightarrow compact\ (:x,\ y:)$
by (*rule compact-Sprod*, *simp add*: *Rep-Sprod-spair strictify-conv-if*)

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

10.7 Properties of *ssplit*

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

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

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

10.8 Strict product preserves flatness

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

10.9 Strict product is a bifinite domain

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

definition
approx-sprod-def:
approx = (λn. Λ(:x, y:). (:*approx* n·x, *approx* n·y:))

instance proof
fix i :: *nat* **and** x :: $'a \otimes 'b$
show *chain* (λi. *approx* i·x)
unfolding *approx-sprod-def* **by** *simp*
show ($\bigsqcup i$. *approx* i·x) = x
unfolding *approx-sprod-def*
by (*simp add*: *lub-distribs eta-cfun*)
show *approx* i·(*approx* i·x) = *approx* i·x
unfolding *approx-sprod-def*
by (*simp add*: *ssplit-def strictify-conv-if*)
have *Rep-Sprod* ' {x::$'a \otimes 'b$. *approx* i·x = x} $\subseteq$ {x. *approx* i·x = x}
unfolding *approx-sprod-def*
apply (*clarify*, *rule-tac* p=x **in** *sprodE*)
apply (*simp add*: *Rep-Sprod-strict*)
apply (*simp add*: *Rep-Sprod-spair spair-eq-iff*)
done
hence *finite* (*Rep-Sprod* ' {x::$'a \otimes 'b$. *approx* i·x = x})
using *finite-fixes-approx* **by** (*rule finite-subset*)
thus *finite* {x::$'a \otimes 'b$. *approx* i·x = x}
by (*rule finite-imageD*, *simp add*: *inj-on-def Rep-Sprod-inject*)
qed

end

lemma *approx-spair* [*simp*]:
approx i·(:x, y:) = (:*approx* i·x, *approx* i·y:)
unfolding *approx-sprod-def*
by (*simp add*: *ssplit-def strictify-conv-if*)

end

11 Discrete: Discrete cpo types

theory *Discrete*
imports *Cont*
begin

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

11.1 Type *'a discr* is a discrete cpo

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

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

instance ..
end

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

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

11.2 Type *'a discr* is a cpo

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

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

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

instance *discr* :: (*type*) *chfin*
apply *intro-classes*
apply (*rule-tac x=0* **in** *exI*)
apply (*unfold max-in-chain-def*)
apply (*clarify*, *erule discr-chain0* [*symmetric*])
done

11.3 *undiscr*

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

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

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

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

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

end

12 Up: The type of lifted values

theory *Up*
imports *Bifinite*
begin

defaultsort *cpo*

12.1 Definition of new type for lifting

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

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

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

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

12.2 Ordering on lifted cpo

instantiation *u* :: (*cpo*) *sq-ord*
begin

definition

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

instance ..
end

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

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

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

12.3 Lifted cpo is a partial order

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

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

instance *u* :: (*finite-po*) *finite-po*
by (*intro-classes*, *simp add*: *u-UNIV*)

12.4 Lifted cpo is a cpo

lemma *is-lub-Iup*:
range S <<| *x* $\Longrightarrow$ *range* ($\lambda i.$ *Iup* (*S i*)) <<| *Iup x*
apply (*rule is-lubI*)
apply (*rule ub-rangeI*)
apply (*subst Iup-less*)
apply (*erule is-ub-lub*)
apply (*case-tac u*)

apply (*drule ub-rangeD*)
apply *simp*
apply *simp*
apply (*erule is-lub-lub*)
apply (*rule ub-rangeI*)
apply (*drule-tac i=i* **in** *ub-rangeD*)
apply *simp*
done

lemma *is-lub-Iup'*: $[\![directed\ S;\ S <<|\ x]\!] \Longrightarrow (Iup\ `\ S) <<|\ Iup\ x$
apply (*rule is-lubI*)
apply (*rule ub-imageI*)
apply (*subst Iup-less*)
apply (*erule (1) is-ubD* [*OF is-lubD1*])
apply (*case-tac u*)
apply (*drule directedD1*, *erule exE*)
apply (*drule (1) ub-imageD*)
apply *simp*
apply *simp*
apply (*erule is-lub-lub*)
apply (*rule is-ubI*)
apply (*drule (1) ub-imageD*)
apply *simp*
done

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

lemma *up-lemma1*: $z \neq Ibottom \Longrightarrow Iup\ (THE\ a.\ Iup\ a = z) = z$
by (*case-tac z*, *simp-all*)

lemma *up-lemma2*:
$[\![chain\ Y;\ Y\ j \neq Ibottom]\!] \Longrightarrow Y\ (i + j) \neq Ibottom$
apply (*erule contrapos-nn*)
apply (*drule-tac i=j* **and** *j=i + j* **in** *chain-mono*)
apply (*rule le-add2*)
apply (*case-tac Y j*)
apply *assumption*
apply *simp*
done

lemma *up-lemma3*:
$[\![chain\ Y;\ Y\ j \neq Ibottom]\!] \Longrightarrow Iup\ (THE\ a.\ Iup\ a = Y\ (i + j)) = Y\ (i + j)$
by (*rule up-lemma1* [*OF up-lemma2*])

lemma *up-lemma4*:
$[\![chain\ Y;\ Y\ j \neq Ibottom]\!] \Longrightarrow chain\ (\lambda i.\ THE\ a.\ Iup\ a = Y\ (i + j))$
apply (*rule chainI*)
apply (*rule Iup-less* [*THEN iffD1*])
apply (*subst up-lemma3*, *assumption*+)+
apply (*simp add*: *chainE*)

done

lemma *up-lemma5*:
  $[\![chain\ Y;\ Y\ j \neq Ibottom]\!] \Longrightarrow$
    $(\lambda i.\ Y\ (i + j)) = (\lambda i.\ Iup\ (THE\ a.\ Iup\ a = Y\ (i + j)))$
by (*rule ext*, *rule up-lemma3* [*symmetric*])

lemma *up-lemma6*:
  $[\![chain\ Y;\ Y\ j \neq Ibottom]\!]$
    $\Longrightarrow range\ Y <<|\ Iup\ (\bigsqcup i.\ THE\ a.\ Iup\ a = Y(i + j))$
apply (*rule-tac j1 = j* **in** *is-lub-range-shift* [*THEN iffD1*])
apply *assumption*
apply (*subst up-lemma5*, *assumption+*)
apply (*rule is-lub-Iup*)
apply (*rule cpo-lubI*)
apply (*erule* (*1*) *up-lemma4*)
done

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

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

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

12.5 Lifted cpo is pointed

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

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

for compatibility with old HOLCF-Version

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

12.6 Continuity of *Iup* and *Ifup*

continuity for *Iup*

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

continuity for *Ifup*

lemma *cont-Ifup1*: *cont* ($\lambda f.$ *Ifup f x*)
by (*induct x*, *simp-all*)

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

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

12.7 Continuous versions of constants

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

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

translations

case l of XCONST up·x ⇒ t == CONST fup·(Λ x. t)·l
Λ(XCONST up·x). t == CONST fup·(Λ x. t)

continuous versions of lemmas for $'a_\perp$

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

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

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

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

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

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

lemma *upE* [*cases type*: *u*]: $[\![p = \perp \Longrightarrow Q;\ \bigwedge x.\ p = up{\cdot}x \Longrightarrow Q]\!] \Longrightarrow Q$
apply (*cases p*)
apply (*simp add*: *inst-up-pcpo*)
apply (*simp add*: *up-def cont-Iup*)
done

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

lifting preserves chain-finiteness

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

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

lemma *compact-upD*: *compact* (*up*·*x*) ⟹ *compact x*
unfolding *compact-def*
by (*drule adm-subst* [*OF cont-Rep-CFun2* [**where** *f*=*up*]], *simp*)

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

instance *u* :: (*chfin*) *chfin*
apply *intro-classes*
apply (*erule compact-imp-max-in-chain*)
apply (*rule-tac p*=⨆ *i*. *Y i* **in** *upE*, *simp-all*)
done

properties of fup

lemma *fup1* [*simp*]: *fup*·*f*·⊥ = ⊥
by (*simp add*: *fup-def cont-Ifup1 cont-Ifup2 inst-up-pcpo*)

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

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

12.8 Lifted cpo is a bifinite domain

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

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

instance proof
fix *i* :: *nat* **and** *x* :: *'a u*
show *chain* (λ*i*. *approx i*·*x*)
unfolding *approx-up-def* **by** *simp*
show (⨆ *i*. *approx i*·*x*) = *x*
unfolding *approx-up-def*
by (*simp add*: *lub-distribs eta-cfun*)
show *approx i*·(*approx i*·*x*) = *approx i*·*x*
unfolding *approx-up-def*
by (*induct x*, *simp*, *simp*)
have {*x*::*'a u*. *approx i*·*x* = *x*} ⊆
insert ⊥ ((λ*x*. *up*·*x*) ‘ {*x*::*'a*. *approx i*·*x* = *x*})
unfolding *approx-up-def*
by (*rule subsetI*, *rule-tac p*=*x* **in** *upE*, *simp-all*)
thus *finite* {*x*::*'a u*. *approx i*·*x* = *x*}
by (*rule finite-subset*, *simp add*: *finite-fixes-approx*)

qed

end

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

end

13 Countable: Encoding (almost) everything into natural numbers

theory *Countable*
imports *Finite-Set List Hilbert-Choice*
begin

13.1 The class of countable types

class *countable* = *itself* +
  assumes *ex-inj*: ∃ *to-nat* :: *'a* ⇒ *nat*. *inj to-nat*

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

13.2 Conversion functions

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

definition *from-nat* :: *nat* ⇒ *'a*::*countable* **where**
  from-nat = *inv* (*to-nat* :: *'a* ⇒ *nat*)

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

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

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

13.3 Countable types

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

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

Pairs

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

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

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

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

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

```
    from sum-mono[OF this] eq
    show ?thesis by auto
  qed
qed

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

Sums

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

Integers

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

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

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

Options

instance *option* :: (*countable*) *countable*
by (*rule countable-classI*[*of* λ*x*. *case x of None* ⇒ *0*
| *Some y* ⇒ *Suc* (*to-nat y*)])
(*auto split*:*option.splits*)

Lists

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

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

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

Functions

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

end

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

theory *Lift*
imports *Discrete Up Cprod Countable*

begin

defaultsort *type*

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

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

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

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

14.1 Lift as a datatype

lemma *lift-distinct1*: ⊥ ≠ *Def x*
by (*simp add*: *Def-def Abs-lift-inject lift-def inst-lift-pcpo*)

lemma *lift-distinct2*: *Def x* ≠ ⊥
by (*simp add*: *Def-def Abs-lift-inject lift-def inst-lift-pcpo*)

lemma *Def-inject*: (*Def x = Def y*) = (*x = y*)
by (*simp add*: *Def-def Abs-lift-inject lift-def*)

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

rep-datatype *lift*
distinct *lift-distinct1 lift-distinct2*
inject *Def-inject*
induction *lift-induct*

lemma *Def-not-UU*: *Def a* ≠ *UU*
by *simp*

⊥ and *Def*

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

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

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

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

For $x \neq \bot$ in assumptions *def-tac* replaces x by *Def a* in conclusion.

ML ⟪
local val lift-definedE = thm lift-definedE
in val def-tac = SIMPSET' (fn ss =>
etac lift-definedE THEN' asm-simp-tac ss)
end;
⟫

lemma *DefE*: $Def\ x = \bot \Longrightarrow R$
by *simp*

lemma *DefE2*: $[\![x = Def\ s;\ x = \bot]\!] \Longrightarrow R$
by *simp*

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

lemma *Def-less-is-eq* [*simp*]: $Def\ x \sqsubseteq y = (Def\ x = y)$
apply (*induct y*)
apply *simp*
apply (*simp add*: *Def-inject-less-eq*)
done

14.2 Lift is flat

lemma *less-lift*: $(x::'a\ lift) \sqsubseteq y = (x = y \vee x = \bot)$
by (*induct x*, *simp-all*)

instance *lift* :: (*type*) *flat*
by (*intro-classes*, *auto simp add*: *less-lift*)

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

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

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

14.3 Further operations

definition

flift1 :: $('a \Rightarrow 'b{::}pcpo) \Rightarrow ('a\ lift \rightarrow 'b)$ (**binder** *FLIFT* 10) **where**
flift1 = $(\lambda f.\ (\Lambda\ x.\ lift\text{-}case\ \bot\ f\ x))$

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

definition
liftpair :: $'a\ lift \times 'b\ lift \Rightarrow ('a \times 'b)\ lift$ **where**
liftpair x = *csplit*·$(FLIFT\ x\ y.\ Def\ (x,\ y))$·*x*

14.4 Continuity Proofs for flift1, flift2

Need the instance of *flat*.

lemma *cont-lift-case1*: *cont* $(\lambda f.\ lift\text{-}case\ a\ f\ x)$
apply (*induct x*)
apply *simp*
apply *simp*
apply (*rule cont-id* [*THEN cont2cont-fun*])
done

lemma *cont-lift-case2*: *cont* $(\lambda x.\ lift\text{-}case\ \bot\ f\ x)$
apply (*rule flatdom-strict2cont*)
apply *simp*
done

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

lemma *cont2cont-flift1* [*simp*]:
$[\![\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)]\!] \Longrightarrow cont\ (\lambda x.\ FLIFT\ y.\ f\ x\ y)$
apply (*rule cont-flift1* [*THEN cont2cont-app3*])
apply *simp*
done

lemma *cont2cont-lift-case* [*simp*]:
$[\![\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y);\ cont\ g]\!] \Longrightarrow cont\ (\lambda x.\ lift\text{-}case\ UU\ (f\ x)\ (g\ x))$
apply (*subgoal-tac cont* $(\lambda x.\ (FLIFT\ y.\ f\ x\ y)$·$(g\ x)))$
apply (*simp add*: *flift1-def cont-lift-case2*)
apply *simp*
done

rewrites for *flift1*, *flift2*

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

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

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

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

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

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

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

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

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

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

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

14.5 Lifted countable types are bifinite

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

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

instance proof
fix *x* :: *'a lift*
show *chain* (λ*i*. *approx i*·*x*)
unfolding *approx-lift-def*

```
    by (rule chainI, cases x, simp-all)
next
  fix x :: 'a lift
  show (⨆ i. approx i·x) = x
    unfolding approx-lift-def
    apply (cases x, simp)
    apply (rule thelubI)
    apply (rule is-lubI)
     apply (rule ub-rangeI, simp)
    apply (drule ub-rangeD)
    apply (erule rev-trans-less)
    apply simp
    apply (rule lessI)
    done
next
  fix i :: nat and x :: 'a lift
  show approx i·(approx i·x) = approx i·x
    unfolding approx-lift-def
    by (cases x, simp, simp)
next
  fix i :: nat
  show finite {x::'a lift. approx i·x = x}
  proof (rule finite-subset)
    let ?S = insert (⊥::'a lift) (Def ` to-nat -` {..<i})
    show {x::'a lift. approx i·x = x} ⊆ ?S
      unfolding approx-lift-def
      by (rule subsetI, case-tac x, simp, simp split: split-if-asm)
    show finite ?S
      by (simp add: finite-vimageI)
  qed
qed

end

end
```

15 Tr: The type of lifted booleans

```
theory Tr
imports Lift
begin

defaultsort pcpo

types
  tr = bool lift

translations
```

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

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

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

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

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

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

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

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

translations
Λ (*CONST TT*). *t* == *CONST trifte*·*t*·⊥
Λ (*CONST FF*). *t* == *CONST trifte*·⊥·*t*

Exhaustion and Elimination for type *tr*

lemma *Exh-tr*: *t* = ⊥ ∨ *t* = *TT* ∨ *t* = *FF*
apply (*unfold FF-def TT-def*)
apply (*induct t*)
apply *fast*
apply *fast*

done

lemma *trE*: $[\![p = \bot \Longrightarrow Q;\ p = TT \Longrightarrow Q;\ p = FF \Longrightarrow Q]\!] \Longrightarrow Q$
apply (*rule Exh-tr* [*THEN disjE*])
apply *fast*
apply (*erule disjE*)
apply *fast*
apply *fast*
done

tactic for tr-thms with case split

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

distinctness for type *tr*

lemma *dist-less-tr* [*simp*]:
$\neg\ TT \sqsubseteq \bot\ \neg\ FF \sqsubseteq \bot\ \neg\ TT \sqsubseteq FF\ \neg\ FF \sqsubseteq TT$
by (*simp-all add*: *tr-defs*)

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

lemmas about andalso, orelse, neg and if

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

lemma *andalso-thms* [*simp*]:
$(TT\ andalso\ y) = y$
$(FF\ andalso\ y) = FF$
$(\bot\ andalso\ y) = \bot$
$(y\ andalso\ TT) = y$
$(y\ andalso\ y) = y$
apply (*unfold andalso-def*, *simp-all*)
apply (*rule-tac p=y* **in** *trE*, *simp-all*)
apply (*rule-tac p=y* **in** *trE*, *simp-all*)
done

lemma *orelse-thms* [*simp*]:
$(TT\ orelse\ y) = TT$
$(FF\ orelse\ y) = y$
$(\bot\ orelse\ y) = \bot$
$(y\ orelse\ FF) = y$
$(y\ orelse\ y) = y$
apply (*unfold orelse-def*, *simp-all*)
apply (*rule-tac p=y* **in** *trE*, *simp-all*)
apply (*rule-tac p=y* **in** *trE*, *simp-all*)

done

lemma *neg-thms* [*simp*]:
neg·*TT* = *FF*
neg·*FF* = *TT*
neg·⊥ = ⊥
by (*simp-all add*: *neg-def TT-def FF-def*)

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

lemma *split-If2*:
P (*If2 Q x y*) = ((*Q* = ⊥ ⟶ *P* ⊥) ∧ (*Q* = *TT* ⟶ *P x*) ∧ (*Q* = *FF* ⟶ *P y*))
apply (*unfold If2-def*)
apply (*rule-tac p* = *Q* **in** *trE*)
apply (*simp-all*)
done

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

15.1 Rewriting of HOLCF operations to HOL functions

lemma *andalso-or*:
t ≠ ⊥ ⟹ ((*t andalso s*) = *FF*) = (*t* = *FF* ∨ *s* = *FF*)
apply (*rule-tac p* = *t* **in** *trE*)
apply *simp-all*
done

lemma *andalso-and*:
t ≠ ⊥ ⟹ ((*t andalso s*) ≠ *FF*) = (*t* ≠ *FF* ∧ *s* ≠ *FF*)
apply (*rule-tac p* = *t* **in** *trE*)
apply *simp-all*
done

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

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

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

lemma *Def-bool4* [*simp*]: (*Def x* ≠ *TT*) = (¬ *x*)
by (*simp add*: *TT-def*)

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

15.2 Compactness

lemma *compact-TT* [*simp*]: *compact TT*
by (*rule compact-chfin*)

lemma *compact-FF* [*simp*]: *compact FF*
by (*rule compact-chfin*)

end

16 Ssum: The type of strict sums

theory *Ssum*
imports *Cprod Tr*
begin

defaultsort *pcpo*

16.1 Definition of strict sum type

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

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

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

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

16.2 Definitions of constructors

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

definition
sinr :: $'b \to ('a \mathbin{++} 'b)$ **where**
sinr = $(\Lambda\ b.\ \mathit{Abs\text{-}Ssum} <\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{FF})\cdot b, \bot, b>)$

lemma *sinl-Ssum*: $<\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{TT})\cdot a, a, \bot> \in \mathit{Ssum}$
by (*simp add*: *Ssum-def strictify-conv-if*)

lemma *sinr-Ssum*: $<\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{FF})\cdot b, \bot, b> \in \mathit{Ssum}$
by (*simp add*: *Ssum-def strictify-conv-if*)

lemma *sinl-Abs-Ssum*: $\mathit{sinl}\cdot a = \mathit{Abs\text{-}Ssum} <\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{TT})\cdot a, a, \bot>$
by (*unfold sinl-def*, *simp add*: *cont-Abs-Ssum sinl-Ssum*)

lemma *sinr-Abs-Ssum*: $\mathit{sinr}\cdot b = \mathit{Abs\text{-}Ssum} <\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{FF})\cdot b, \bot, b>$
by (*unfold sinr-def*, *simp add*: *cont-Abs-Ssum sinr-Ssum*)

lemma *Rep-Ssum-sinl*: $\mathit{Rep\text{-}Ssum}\ (\mathit{sinl}\cdot a) = <\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{TT})\cdot a, a, \bot>$
by (*simp add*: *sinl-Abs-Ssum Abs-Ssum-inverse sinl-Ssum*)

lemma *Rep-Ssum-sinr*: $\mathit{Rep\text{-}Ssum}\ (\mathit{sinr}\cdot b) = <\mathit{strictify}\cdot(\Lambda\ \text{-}.\ \mathit{FF})\cdot b, \bot, b>$
by (*simp add*: *sinr-Abs-Ssum Abs-Ssum-inverse sinr-Ssum*)

16.3 Properties of *sinl* and *sinr*

Ordering

lemma *sinl-less* [*simp*]: $(\mathit{sinl}\cdot x \sqsubseteq \mathit{sinl}\cdot y) = (x \sqsubseteq y)$
by (*simp add*: *less-Ssum-def Rep-Ssum-sinl strictify-conv-if*)

lemma *sinr-less* [*simp*]: $(\mathit{sinr}\cdot x \sqsubseteq \mathit{sinr}\cdot y) = (x \sqsubseteq y)$
by (*simp add*: *less-Ssum-def Rep-Ssum-sinr strictify-conv-if*)

lemma *sinl-less-sinr* [*simp*]: $(\mathit{sinl}\cdot x \sqsubseteq \mathit{sinr}\cdot y) = (x = \bot)$
by (*simp add*: *less-Ssum-def Rep-Ssum-sinl Rep-Ssum-sinr strictify-conv-if*)

lemma *sinr-less-sinl* [*simp*]: $(\mathit{sinr}\cdot x \sqsubseteq \mathit{sinl}\cdot y) = (x = \bot)$
by (*simp add*: *less-Ssum-def Rep-Ssum-sinl Rep-Ssum-sinr strictify-conv-if*)

Equality

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

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

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

lemma *sinr-eq-sinl* [*simp*]: $(\mathit{sinr}\cdot x = \mathit{sinl}\cdot y) = (x = \bot \wedge y = \bot)$

by (*subst po-eq-conv*, *simp*)

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

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

Strictness

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

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

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

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

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

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

Compactness

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

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

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

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

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

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

16.4 Case analysis

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

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

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

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

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

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

16.5 Case analysis combinator

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

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

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

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

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

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

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

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

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

16.6 Strict sum preserves flatness

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

16.7 Strict sum is a bifinite domain

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

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

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

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

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

by (*cases x*, *simp add*: *approx-ssum-def*, *simp*, *simp*)
have $\{x::'a \oplus 'b.\ approx\ i{\cdot}x = x\} \subseteq$
$(\lambda x.\ sinl{\cdot}x)$ ` $\{x.\ approx\ i{\cdot}x = x\} \cup$
$(\lambda x.\ sinr{\cdot}x)$ ` $\{x.\ approx\ i{\cdot}x = x\}$
by (*rule subsetI*, *rule-tac p=x* **in** *ssumE2*, *simp*, *simp*)
thus *finite* $\{x::'a \oplus 'b.\ approx\ i{\cdot}x = x\}$
by (*rule finite-subset*,
intro finite-UnI finite-imageI finite-fixes-approx)
qed

end

end

17 One: The unit domain

theory *One*
imports *Lift*
begin

types *one* = *unit lift*
translations
one <= (*type*) *unit lift*

constdefs
ONE :: *one*
ONE == *Def* ()

Exhaustion and Elimination for type *one*

lemma *Exh-one*: $t = \bot \vee t = ONE$
apply (*unfold ONE-def*)
apply (*induct t*)
apply *simp*
apply *simp*
done

lemma *oneE*: $[\![p = \bot \Longrightarrow Q;\ p = ONE \Longrightarrow Q]\!] \Longrightarrow Q$
apply (*rule Exh-one* [*THEN disjE*])
apply *fast*
apply *fast*
done

lemma *dist-less-one* [*simp*]: $\neg\ ONE \sqsubseteq \bot$
apply (*unfold ONE-def*)
apply *simp*
done

lemma *dist-eq-one* [*simp*]: $ONE \neq \bot\ \bot \neq ONE$

apply (*unfold ONE-def*)
apply *simp-all*
done

lemma *compact-ONE* [*simp*]: *compact ONE*
by (*rule compact-chfin*)

Case analysis function for type *one*

definition
one-when :: *'a::pcpo* → *one* → *'a* **where**
one-when = (Λ *a*. *strictify*·(Λ -. *a*))

translations
case x of CONST ONE ⇒ *t* == *CONST one-when*·*t*·*x*
Λ (*CONST ONE*). *t* == *CONST one-when*·*t*

lemma *one-when1* [*simp*]: (*case* ⊥ *of ONE* ⇒ *t*) = ⊥
by (*simp add*: *one-when-def*)

lemma *one-when2* [*simp*]: (*case ONE of ONE* ⇒ *t*) = *t*
by (*simp add*: *one-when-def*)

lemma *one-when3* [*simp*]: (*case x of ONE* ⇒ *ONE*) = *x*
by (*rule-tac p*=*x* **in** *oneE*, *simp-all*)

end

18 Fix: Fixed point operator and admissibility

theory *Fix*
imports *Cfun Cprod Adm*
begin

defaultsort *pcpo*

18.1 Iteration

consts
iterate :: *nat* ⇒ (*'a::cpo* → *'a*) → (*'a* → *'a*)

primrec
iterate 0 = (Λ *F x*. *x*)
iterate (*Suc n*) = (Λ *F x*. *F*·(*iterate n*·*F*·*x*))

Derive inductive properties of iterate from primitive recursion

lemma *iterate-0* [*simp*]: *iterate 0*·*F*·*x* = *x*
by *simp*

lemma *iterate-Suc* [*simp*]: *iterate* (*Suc n*)·*F*·*x* = *F*·(*iterate n*·*F*·*x*)
by *simp*

declare *iterate.simps* [*simp del*]

lemma *iterate-Suc2*: *iterate* (*Suc n*)·*F*·*x* = *iterate n*·*F*·(*F*·*x*)
by (*induct-tac n*, *auto*)

The sequence of function iterations is a chain. This property is essential since monotonicity of iterate makes no sense.

lemma *chain-iterate2*: *x* ⊑ *F*·*x* ⟹ *chain* (λ*i*. *iterate i*·*F*·*x*)
by (*rule chainI*, *induct-tac i*, *auto elim*: *monofun-cfun-arg*)

lemma *chain-iterate* [*simp*]: *chain* (λ*i*. *iterate i*·*F*·⊥)
by (*rule chain-iterate2* [*OF minimal*])

18.2 Least fixed point operator

definition
fix :: ('*a* → '*a*) → '*a* **where**
fix = (Λ *F*. ⨆ *i*. *iterate i*·*F*·⊥)

Binder syntax for *fix*

syntax
-FIX :: ['*a*, '*a*] ⇒ '*a* ((*3FIX -./ -*) [*1000*, *10*] *10*)

syntax (*xsymbols*)
-FIX :: ['*a*, '*a*] ⇒ '*a* ((*3*μ *-./ -*) [*1000*, *10*] *10*)

translations
μ *x*. *t* == *CONST fix*·(Λ *x*. *t*)

Properties of *fix*

direct connection between *fix* and iteration

lemma *fix-def2*: *fix*·*F* = (⨆ *i*. *iterate i*·*F*·⊥)
apply (*unfold fix-def*)
apply (*rule beta-cfun*)
apply (*rule cont2cont-lub*)
apply (*rule ch2ch-lambda*)
apply (*rule chain-iterate*)
apply *simp*
done

Kleene's fixed point theorems for continuous functions in pointed omega cpo's

lemma *fix-eq*: *fix*·*F* = *F*·(*fix*·*F*)

apply (*simp add*: *fix-def2*)
apply (*subst lub-range-shift* [*of - 1*, *symmetric*])
apply (*rule chain-iterate*)
apply (*subst contlub-cfun-arg*)
apply (*rule chain-iterate*)
apply *simp*
done

lemma *fix-least-less*: $F{\cdot}x \sqsubseteq x \Longrightarrow fix{\cdot}F \sqsubseteq x$
apply (*simp add*: *fix-def2*)
apply (*rule is-lub-thelub*)
apply (*rule chain-iterate*)
apply (*rule ub-rangeI*)
apply (*induct-tac i*)
apply *simp*
apply *simp*
apply (*erule rev-trans-less*)
apply (*erule monofun-cfun-arg*)
done

lemma *fix-least*: $F{\cdot}x = x \Longrightarrow fix{\cdot}F \sqsubseteq x$
by (*rule fix-least-less*, *simp*)

lemma *fix-eqI*: $[\![F{\cdot}x = x; \forall z.\ F{\cdot}z = z \longrightarrow x \sqsubseteq z]\!] \Longrightarrow x = fix{\cdot}F$
apply (*rule antisym-less*)
apply (*simp add*: *fix-eq* [*symmetric*])
apply (*erule fix-least*)
done

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

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

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

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

strictness of *fix*

lemma *fix-defined-iff*: $(fix{\cdot}F = \bot) = (F{\cdot}\bot = \bot)$
apply (*rule iffI*)
apply (*erule subst*)
apply (*rule fix-eq* [*symmetric*])
apply (*erule fix-least* [*THEN UU-I*])

done

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

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

fix applied to identity and constant functions

lemma *fix-id*: $(\mu\ x.\ x) = \bot$
by (*simp add*: *fix-strict*)

lemma *fix-const*: $(\mu\ x.\ c) = c$
by (*subst fix-eq*, *simp*)

18.3 Fixed point induction

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

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

18.4 Recursive let bindings

definition
CLetrec :: $('a \to 'a \times 'b) \to 'b$ **where**
CLetrec = $(\Lambda\ F.\ \mathit{csnd}{\cdot}(F{\cdot}(\mu\ x.\ \mathit{cfst}{\cdot}(F{\cdot}x))))$

nonterminals
recbinds recbindt recbind

syntax
-recbind :: $['a, 'a] \Rightarrow \mathit{recbind}$ ((*2- =/ -*) *10*)
:: *recbind* $\Rightarrow$ *recbindt* (-)
-recbindt :: [*recbind*, *recbindt*] $\Rightarrow$ *recbindt* (-,/ -)
:: *recbindt* $\Rightarrow$ *recbinds* (-)
-recbinds :: [*recbindt*, *recbinds*] $\Rightarrow$ *recbinds* (-;/ -)
-Letrec :: [*recbinds*, $'a$] $\Rightarrow$ $'a$ ((*Letrec* (-)/ *in* (-)) *10*)

translations
(*recbindt*) *x* = *a*, ⟨*y*,*ys*⟩ = ⟨*b*,*bs*⟩ == (*recbindt*) ⟨*x*,*y*,*ys*⟩ = ⟨*a*,*b*,*bs*⟩
(*recbindt*) *x* = *a*, *y* = *b* == (*recbindt*) ⟨*x*,*y*⟩ = ⟨*a*,*b*⟩

translations
-Letrec (-recbinds b bs) e == -Letrec b (-Letrec bs e)
Letrec xs = a in ⟨e,es⟩ == CONST CLetrec·(Λ xs. ⟨a,e,es⟩)
Letrec xs = a in e == CONST CLetrec·(Λ xs. ⟨a,e⟩)

Bekic's Theorem: Simultaneous fixed points over pairs can be written in terms of separate fixed points.

lemma *fix-cprod*:
fix·(F::'a × 'b → 'a × 'b) =
⟨μ x. cfst·(F·⟨x, μ y. csnd·(F·⟨x, y⟩)⟩),
μ y. csnd·(F·⟨μ x. cfst·(F·⟨x, μ y. csnd·(F·⟨x, y⟩)⟩), y⟩)⟩
(**is** *fix·F = ⟨?x, ?y⟩*)
proof (*rule fix-eqI* [*rule-format*, *symmetric*])
have *1*: *cfst·(F·⟨?x, ?y⟩) = ?x*
by (*rule trans* [*symmetric*, *OF fix-eq*], *simp*)
have *2*: *csnd·(F·⟨?x, ?y⟩) = ?y*
by (*rule trans* [*symmetric*, *OF fix-eq*], *simp*)
from *1 2* **show** *F·⟨?x, ?y⟩ = ⟨?x, ?y⟩* **by** (*simp add*: *eq-cprod*)
next
fix *z* **assume** *F-z*: *F·z = z*
then obtain *x y* **where** *z*: *z = ⟨x,y⟩* **by** (*rule-tac p=z* **in** *cprodE*)
from *F-z z* **have** *F-x*: *cfst·(F·⟨x, y⟩) = x* **by** *simp*
from *F-z z* **have** *F-y*: *csnd·(F·⟨x, y⟩) = y* **by** *simp*
let *?y1 = μ y. csnd·(F·⟨x, y⟩)*
have *?y1 ⊑ y* **by** (*rule fix-least*, *simp add*: *F-y*)
hence *cfst·(F·⟨x, ?y1⟩) ⊑ cfst·(F·⟨x, y⟩)* **by** (*simp add*: *monofun-cfun*)
hence *cfst·(F·⟨x, ?y1⟩) ⊑ x* **using** *F-x* **by** *simp*
hence *1*: *?x ⊑ x* **by** (*simp add*: *fix-least-less*)
hence *csnd·(F·⟨?x, y⟩) ⊑ csnd·(F·⟨x, y⟩)* **by** (*simp add*: *monofun-cfun*)
hence *csnd·(F·⟨?x, y⟩) ⊑ y* **using** *F-y* **by** *simp*
hence *2*: *?y ⊑ y* **by** (*simp add*: *fix-least-less*)
show *⟨?x, ?y⟩ ⊑ z* **using** *z 1 2* **by** *simp*
qed

18.5 Weak admissibility

definition
admw :: (*'a* ⇒ *bool*) ⇒ *bool* **where**
admw P = (∀ F. (∀ n. P (iterate n·F·⊥)) ⟶ P (⨆ i. iterate i·F·⊥))

an admissible formula is also weak admissible

lemma *adm-impl-admw*: *adm P ⟹ admw P*
apply (*unfold admw-def*)
apply (*intro strip*)
apply (*erule admD*)
apply (*rule chain-iterate*)
apply (*erule spec*)
done

computational induction for weak admissible formulae

lemma *wfix-ind*: $[\![admw\ P; \forall n.\ P\ (iterate\ n{\cdot}F{\cdot}\bot)]\!] \Longrightarrow P\ (fix{\cdot}F)$
by (*simp add*: *fix-def2 admw-def*)

lemma *def-wfix-ind*:
$[\![f \equiv fix{\cdot}F;\ admw\ P; \forall n.\ P\ (iterate\ n{\cdot}F{\cdot}\bot)]\!] \Longrightarrow P\ f$
by (*simp*, *rule wfix-ind*)

end

19 Fixrec: Package for defining recursive functions in HOLCF

theory *Fixrec*
imports *Sprod Ssum Up One Tr Fix*
uses (*Tools/fixrec-package.ML*)
begin

19.1 Maybe monad type

defaultsort *cpo*

pcpodef (**open**) *'a maybe* = *UNIV*::(*one* ++ *'a u*) *set*
by *simp*

constdefs
fail :: *'a maybe*
fail ≡ *Abs-maybe* (*sinl*·*ONE*)

constdefs
return :: *'a* → *'a maybe* **where**
return ≡ Λ *x*. *Abs-maybe* (*sinr*·(*up*·*x*))

definition
maybe-when :: *'b* → (*'a* → *'b*) → *'a maybe* → *'b*::*pcpo* **where**
maybe-when = (Λ *f r m*. *sscase*·(Λ *x*. *f*)·(*fup*·*r*)·(*Rep-maybe m*))

lemma *maybeE*:
$[\![p = \bot \Longrightarrow Q;\ p = fail \Longrightarrow Q;\ \bigwedge x.\ p = return{\cdot}x \Longrightarrow Q]\!] \Longrightarrow Q$
apply (*unfold fail-def return-def*)
apply (*cases p*, *rename-tac r*)
apply (*rule-tac p*=*r* **in** *ssumE*, *simp add*: *Abs-maybe-strict*)
apply (*rule-tac p*=*x* **in** *oneE*, *simp*, *simp*)
apply (*rule-tac p*=*y* **in** *upE*, *simp*, *simp add*: *cont-Abs-maybe*)
done

lemma *return-defined* [*simp*]: *return*·*x* ≠ ⊥
by (*simp add*: *return-def cont-Abs-maybe Abs-maybe-defined*)

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

lemma *return-eq* [*simp*]: $(return{\cdot}x = return{\cdot}y) = (x = y)$
by (*simp add*: *return-def cont-Abs-maybe Abs-maybe-inject*)

lemma *return-neq-fail* [*simp*]:
$return{\cdot}x \neq fail$ $fail \neq return{\cdot}x$
by (*simp-all add*: *return-def fail-def cont-Abs-maybe Abs-maybe-inject*)

lemma *maybe-when-rews* [*simp*]:
$maybe\text{-}when{\cdot}f{\cdot}r{\cdot}\bot = \bot$
$maybe\text{-}when{\cdot}f{\cdot}r{\cdot}fail = f$
$maybe\text{-}when{\cdot}f{\cdot}r{\cdot}(return{\cdot}x) = r{\cdot}x$
by (*simp-all add*: *return-def fail-def maybe-when-def cont-Rep-maybe*
cont-Abs-maybe Abs-maybe-inverse Rep-maybe-strict)

translations
$case\ m\ of\ fail \Rightarrow t1 \mid return{\cdot}x \Rightarrow t2 == CONST\ maybe\text{-}when{\cdot}t1{\cdot}(\Lambda\ x.\ t2){\cdot}m$

19.1.1 Monadic bind operator

definition
$bind :: 'a\ maybe \to ('a \to 'b\ maybe) \to 'b\ maybe$ **where**
$bind = (\Lambda\ m\ f.\ case\ m\ of\ fail \Rightarrow fail \mid return{\cdot}x \Rightarrow f{\cdot}x)$

monad laws

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

lemma *bind-fail* [*simp*]: $bind{\cdot}fail{\cdot}f = fail$
by (*simp add*: *bind-def*)

lemma *left-unit* [*simp*]: $bind{\cdot}(return{\cdot}a){\cdot}k = k{\cdot}a$
by (*simp add*: *bind-def*)

lemma *right-unit* [*simp*]: $bind{\cdot}m{\cdot}return = m$
by (*rule-tac p=m* **in** *maybeE*, *simp-all*)

lemma *bind-assoc*:
$bind{\cdot}(bind{\cdot}m{\cdot}k){\cdot}h = bind{\cdot}m{\cdot}(\Lambda\ a.\ bind{\cdot}(k{\cdot}a){\cdot}h)$
by (*rule-tac p=m* **in** *maybeE*, *simp-all*)

19.1.2 Run operator

definition
$run :: 'a\ maybe \to 'a :: pcpo$ **where**
$run = maybe\text{-}when{\cdot}\bot{\cdot}ID$

rewrite rules for run

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

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

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

19.1.3 Monad plus operator

definition
$mplus :: 'a\ maybe \to 'a\ maybe \to 'a\ maybe$ **where**
$mplus = (\Lambda\ m1\ m2.\ case\ m1\ of\ fail \Rightarrow m2 \mid return{\cdot}x \Rightarrow m1)$

abbreviation
$mplus\text{-}syn :: ['a\ maybe, 'a\ maybe] \Rightarrow 'a\ maybe$ (**infixr** $+\!+\!+$ *65*) **where**
$m1 +\!+\!+ m2 == mplus{\cdot}m1{\cdot}m2$

rewrite rules for mplus

lemma *mplus-strict* [*simp*]: $\bot +\!+\!+ m = \bot$
by (*simp add*: *mplus-def*)

lemma *mplus-fail* [*simp*]: $fail +\!+\!+ m = m$
by (*simp add*: *mplus-def*)

lemma *mplus-return* [*simp*]: $return{\cdot}x +\!+\!+ m = return{\cdot}x$
by (*simp add*: *mplus-def*)

lemma *mplus-fail2* [*simp*]: $m +\!+\!+ fail = m$
by (*rule-tac p=m* **in** *maybeE*, *simp-all*)

lemma *mplus-assoc*: $(x +\!+\!+ y) +\!+\!+ z = x +\!+\!+ (y +\!+\!+ z)$
by (*rule-tac p=x* **in** *maybeE*, *simp-all*)

19.1.4 Fatbar combinator

definition
$fatbar :: ('a \to 'b\ maybe) \to ('a \to 'b\ maybe) \to ('a \to 'b\ maybe)$ **where**
$fatbar = (\Lambda\ a\ b\ x.\ a{\cdot}x +\!+\!+ b{\cdot}x)$

abbreviation
$fatbar\text{-}syn :: ['a \to 'b\ maybe, 'a \to 'b\ maybe] \Rightarrow 'a \to 'b\ maybe$ (**infixr** $\|$ *60*)
where
$m1 \parallel m2 == fatbar{\cdot}m1{\cdot}m2$

lemma *fatbar1*: $m{\cdot}x = \bot \Longrightarrow (m \parallel ms){\cdot}x = \bot$
by (*simp add*: *fatbar-def*)

lemma *fatbar2*: $m{\cdot}x = \mathit{fail} \Longrightarrow (m \parallel ms){\cdot}x = ms{\cdot}x$
by (*simp add*: *fatbar-def*)

lemma *fatbar3*: $m{\cdot}x = \mathit{return}{\cdot}y \Longrightarrow (m \parallel ms){\cdot}x = \mathit{return}{\cdot}y$
by (*simp add*: *fatbar-def*)

lemmas *fatbar-simps* = *fatbar1 fatbar2 fatbar3*

lemma *run-fatbar1*: $m{\cdot}x = \bot \Longrightarrow \mathit{run}{\cdot}((m \parallel ms){\cdot}x) = \bot$
by (*simp add*: *fatbar-def*)

lemma *run-fatbar2*: $m{\cdot}x = \mathit{fail} \Longrightarrow \mathit{run}{\cdot}((m \parallel ms){\cdot}x) = \mathit{run}{\cdot}(ms{\cdot}x)$
by (*simp add*: *fatbar-def*)

lemma *run-fatbar3*: $m{\cdot}x = \mathit{return}{\cdot}y \Longrightarrow \mathit{run}{\cdot}((m \parallel ms){\cdot}x) = y$
by (*simp add*: *fatbar-def*)

lemmas *run-fatbar-simps* [*simp*] = *run-fatbar1 run-fatbar2 run-fatbar3*

19.2 Case branch combinator

constdefs
branch :: $('a \to 'b\ \mathit{maybe}) \Rightarrow ('b \to 'c) \to ('a \to 'c\ \mathit{maybe})$
branch $p \equiv \Lambda\ r\ x.\ \mathit{bind}{\cdot}(p{\cdot}x){\cdot}(\Lambda\ y.\ \mathit{return}{\cdot}(r{\cdot}y))$

lemma *branch-rews*:
$p{\cdot}x = \bot \Longrightarrow \mathit{branch}\ p{\cdot}r{\cdot}x = \bot$
$p{\cdot}x = \mathit{fail} \Longrightarrow \mathit{branch}\ p{\cdot}r{\cdot}x = \mathit{fail}$
$p{\cdot}x = \mathit{return}{\cdot}y \Longrightarrow \mathit{branch}\ p{\cdot}r{\cdot}x = \mathit{return}{\cdot}(r{\cdot}y)$
by (*simp-all add*: *branch-def*)

lemma *branch-return* [*simp*]: $\mathit{branch}\ \mathit{return}{\cdot}r{\cdot}x = \mathit{return}{\cdot}(r{\cdot}x)$
by (*simp add*: *branch-def*)

19.3 Case syntax

nonterminals
Case-syn *Cases-syn*

syntax
-Case-syntax:: $['a,\ \mathit{Cases\text{-}syn}] \Rightarrow 'b$ ((Case - of/ -) 10)
-Case1 :: $['a,\ 'b] \Rightarrow \mathit{Case\text{-}syn}$ ((2- =>/ -) 10)
:: $\mathit{Case\text{-}syn} \Rightarrow \mathit{Cases\text{-}syn}$ (-)
-Case2 :: $[\mathit{Case\text{-}syn},\ \mathit{Cases\text{-}syn}] \Rightarrow \mathit{Cases\text{-}syn}$ (-/ | -)

syntax (*xsymbols*)
-Case1 :: $['a,\ 'b] \Rightarrow \mathit{Case\text{-}syn}$ ((2- ⇒/ -) 10)

translations

-Case-syntax x ms == CONST Fixrec.run·(ms·x)
-Case2 m ms == m || ms

Parsing Case expressions

syntax
-pat :: 'a
-var :: 'a
-noargs :: 'a

translations
-Case1 p r => CONST branch (-pat p)·(-var p r)
-var (-args x y) r => CONST csplit·(-var x (-var y r))
-var -noargs r => CONST unit-when·r

parse-translation ⟪
(rewrites (-pat x) => (return) *)*
(rewrites (-var x t) => (Abs-CFun (%x. t)) *)*
[(-pat, K (Syntax.const Fixrec.return)),
mk-binder-tr (-var, Abs-CFun)];
⟫

Printing Case expressions

syntax
-match :: 'a

print-translation ⟪
let
fun dest-LAM (Const (@{const-syntax Rep-CFun},-) $ Const (@{const-syntax unit-when},-) $ t) =
(Syntax.const -noargs, t)
| dest-LAM (Const (@{const-syntax Rep-CFun},-) $ Const (@{const-syntax csplit},-) $ t) =
let
val (v1, t1) = dest-LAM t;
val (v2, t2) = dest-LAM t1;
in (Syntax.const -args $ v1 $ v2, t2) end
| dest-LAM (Const (@{const-syntax Abs-CFun},-) $ t) =
let
val abs = case t of Abs abs => abs
| - => (x, dummyT, incr-boundvars 1 t $ Bound 0);
val (x, t') = atomic-abs-tr' abs;
in (Syntax.const -var $ x, t') end
| dest-LAM - = raise Match; (too few vars: abort translation *)*

fun Case1-tr' [Const(@{const-syntax branch},-) $ p, r] =
let val (v, t) = dest-LAM r;
in Syntax.const -Case1 $ (Syntax.const -match $ p $ v) $ t end;

in [(@{const-syntax Rep-CFun}, Case1-tr')] end;

⟫

translations
x <= -match Fixrec.return (-var x)

19.4 Pattern combinators for data constructors

types *('a, 'b) pat = 'a → 'b maybe*

definition
cpair-pat :: ('a, 'c) pat ⇒ ('b, 'd) pat ⇒ ('a × 'b, 'c × 'd) pat **where**
cpair-pat p1 p2 = (Λ⟨x, y⟩.
bind·(p1·x)·(Λ a. bind·(p2·y)·(Λ b. return·⟨a, b⟩)))

definition
spair-pat ::
('a, 'c) pat ⇒ ('b, 'd) pat ⇒ ('a::pcpo ⊗ 'b::pcpo, 'c × 'd) pat **where**
spair-pat p1 p2 = (Λ(:x, y:). cpair-pat p1 p2·⟨x, y⟩)

definition
sinl-pat :: ('a, 'c) pat ⇒ ('a::pcpo ⊕ 'b::pcpo, 'c) pat **where**
sinl-pat p = sscase·p·(Λ x. fail)

definition
sinr-pat :: ('b, 'c) pat ⇒ ('a::pcpo ⊕ 'b::pcpo, 'c) pat **where**
sinr-pat p = sscase·(Λ x. fail)·p

definition
up-pat :: ('a, 'b) pat ⇒ ('a u, 'b) pat **where**
up-pat p = fup·p

definition
TT-pat :: (tr, unit) pat **where**
TT-pat = (Λ b. If b then return·() else fail fi)

definition
FF-pat :: (tr, unit) pat **where**
FF-pat = (Λ b. If b then fail else return·() fi)

definition
ONE-pat :: (one, unit) pat **where**
ONE-pat = (Λ ONE. return·())

Parse translations (patterns)

translations
-pat (XCONST cpair·x·y) => CONST cpair-pat (-pat x) (-pat y)
-pat (XCONST spair·x·y) => CONST spair-pat (-pat x) (-pat y)
-pat (XCONST sinl·x) => CONST sinl-pat (-pat x)
-pat (XCONST sinr·x) => CONST sinr-pat (-pat x)

-pat (XCONST up·x) => CONST up-pat (-pat x)
-pat (XCONST TT) => CONST TT-pat
-pat (XCONST FF) => CONST FF-pat
-pat (XCONST ONE) => CONST ONE-pat

CONST version is also needed for constructors with special syntax

translations
-pat (CONST cpair·x·y) => CONST cpair-pat (-pat x) (-pat y)
-pat (CONST spair·x·y) => CONST spair-pat (-pat x) (-pat y)

Parse translations (variables)

translations
-var (XCONST cpair·x·y) r => -var (-args x y) r
-var (XCONST spair·x·y) r => -var (-args x y) r
-var (XCONST sinl·x) r => -var x r
-var (XCONST sinr·x) r => -var x r
-var (XCONST up·x) r => -var x r
-var (XCONST TT) r => -var -noargs r
-var (XCONST FF) r => -var -noargs r
-var (XCONST ONE) r => -var -noargs r

translations
-var (CONST cpair·x·y) r => -var (-args x y) r
-var (CONST spair·x·y) r => -var (-args x y) r

Print translations

translations
CONST cpair·(-match p1 v1)·(-match p2 v2)
<= -match (CONST cpair-pat p1 p2) (-args v1 v2)
CONST spair·(-match p1 v1)·(-match p2 v2)
<= -match (CONST spair-pat p1 p2) (-args v1 v2)
CONST sinl·(-match p1 v1) <= -match (CONST sinl-pat p1) v1
CONST sinr·(-match p1 v1) <= -match (CONST sinr-pat p1) v1
CONST up·(-match p1 v1) <= -match (CONST up-pat p1) v1
CONST TT <= -match (CONST TT-pat) -noargs
CONST FF <= -match (CONST FF-pat) -noargs
CONST ONE <= -match (CONST ONE-pat) -noargs

lemma *cpair-pat1*:
branch p·r·x = ⊥ ⟹ branch (cpair-pat p q)·(csplit·r)·⟨x, y⟩ = ⊥
apply (*simp add*: *branch-def cpair-pat-def*)
apply (*rule-tac p=p·x* **in** *maybeE*, *simp-all*)
done

lemma *cpair-pat2*:
branch p·r·x = fail ⟹ branch (cpair-pat p q)·(csplit·r)·⟨x, y⟩ = fail
apply (*simp add*: *branch-def cpair-pat-def*)
apply (*rule-tac p=p·x* **in** *maybeE*, *simp-all*)
done

lemma *cpair-pat3*:
  $branch\ p{\cdot}r{\cdot}x = return{\cdot}s \Longrightarrow$
   $branch\ (cpair\text{-}pat\ p\ q){\cdot}(csplit{\cdot}r){\cdot}\langle x,\ y\rangle = branch\ q{\cdot}s{\cdot}y$
apply (*simp add*: *branch-def cpair-pat-def*)
apply (*rule-tac p*=$p{\cdot}x$ **in** *maybeE*, *simp-all*)
apply (*rule-tac p*=$q{\cdot}y$ **in** *maybeE*, *simp-all*)
done

lemmas *cpair-pat* [*simp*] =
  cpair-pat1 cpair-pat2 cpair-pat3

lemma *spair-pat* [*simp*]:
  $branch\ (spair\text{-}pat\ p1\ p2){\cdot}r{\cdot}\bot = \bot$
  $[\![x \neq \bot;\ y \neq \bot]\!]$
    $\Longrightarrow branch\ (spair\text{-}pat\ p1\ p2){\cdot}r{\cdot}(:x,\ y:) =$
        $branch\ (cpair\text{-}pat\ p1\ p2){\cdot}r{\cdot}\langle x,\ y\rangle$
by (*simp-all add*: *branch-def spair-pat-def*)

lemma *sinl-pat* [*simp*]:
  $branch\ (sinl\text{-}pat\ p){\cdot}r{\cdot}\bot = \bot$
  $x \neq \bot \Longrightarrow branch\ (sinl\text{-}pat\ p){\cdot}r{\cdot}(sinl{\cdot}x) = branch\ p{\cdot}r{\cdot}x$
  $y \neq \bot \Longrightarrow branch\ (sinl\text{-}pat\ p){\cdot}r{\cdot}(sinr{\cdot}y) = fail$
by (*simp-all add*: *branch-def sinl-pat-def*)

lemma *sinr-pat* [*simp*]:
  $branch\ (sinr\text{-}pat\ p){\cdot}r{\cdot}\bot = \bot$
  $x \neq \bot \Longrightarrow branch\ (sinr\text{-}pat\ p){\cdot}r{\cdot}(sinl{\cdot}x) = fail$
  $y \neq \bot \Longrightarrow branch\ (sinr\text{-}pat\ p){\cdot}r{\cdot}(sinr{\cdot}y) = branch\ p{\cdot}r{\cdot}y$
by (*simp-all add*: *branch-def sinr-pat-def*)

lemma *up-pat* [*simp*]:
  $branch\ (up\text{-}pat\ p){\cdot}r{\cdot}\bot = \bot$
  $branch\ (up\text{-}pat\ p){\cdot}r{\cdot}(up{\cdot}x) = branch\ p{\cdot}r{\cdot}x$
by (*simp-all add*: *branch-def up-pat-def*)

lemma *TT-pat* [*simp*]:
  $branch\ TT\text{-}pat{\cdot}(unit\text{-}when{\cdot}r){\cdot}\bot = \bot$
  $branch\ TT\text{-}pat{\cdot}(unit\text{-}when{\cdot}r){\cdot}TT = return{\cdot}r$
  $branch\ TT\text{-}pat{\cdot}(unit\text{-}when{\cdot}r){\cdot}FF = fail$
by (*simp-all add*: *branch-def TT-pat-def*)

lemma *FF-pat* [*simp*]:
  $branch\ FF\text{-}pat{\cdot}(unit\text{-}when{\cdot}r){\cdot}\bot = \bot$
  $branch\ FF\text{-}pat{\cdot}(unit\text{-}when{\cdot}r){\cdot}TT = fail$
  $branch\ FF\text{-}pat{\cdot}(unit\text{-}when{\cdot}r){\cdot}FF = return{\cdot}r$
by (*simp-all add*: *branch-def FF-pat-def*)

lemma *ONE-pat* [*simp*]:

branch ONE-pat·(unit-when·r)·⊥ = ⊥
branch ONE-pat·(unit-when·r)·ONE = return·r
by (*simp-all add*: *branch-def ONE-pat-def*)

19.5 Wildcards, as-patterns, and lazy patterns

syntax
-as-pat :: [*idt*, *'a*] ⇒ *'a* (**infixr** *as 10*)
-lazy-pat :: *'a* ⇒ *'a* (∼ - [*1000*] *1000*)

definition
wild-pat :: *'a* → *unit maybe* **where**
wild-pat = (Λ *x*. *return·*())

definition
as-pat :: (*'a* → *'b maybe*) ⇒ *'a* → (*'a* × *'b*) *maybe* **where**
as-pat p = (Λ *x*. *bind·*(*p·x*)·(Λ *a*. *return·*⟨*x*, *a*⟩))

definition
lazy-pat :: (*'a* → *'b*::*pcpo maybe*) ⇒ (*'a* → *'b maybe*) **where**
lazy-pat p = (Λ *x*. *return·*(*run·*(*p·x*)))

Parse translations (patterns)

translations
-pat - => *CONST wild-pat*
-pat (*-as-pat x y*) => *CONST as-pat* (*-pat y*)
-pat (*-lazy-pat x*) => *CONST lazy-pat* (*-pat x*)

Parse translations (variables)

translations
-var - *r* => *-var -noargs r*
-var (*-as-pat x y*) *r* => *-var* (*-args x y*) *r*
-var (*-lazy-pat x*) *r* => *-var x r*

Print translations

translations
- <= *-match* (*CONST wild-pat*) *-noargs*
-as-pat x (*-match p v*) <= *-match* (*CONST as-pat p*) (*-args* (*-var x*) *v*)
-lazy-pat (*-match p v*) <= *-match* (*CONST lazy-pat p*) *v*

Lazy patterns in lambda abstractions

translations
-cabs (*-lazy-pat p*) *r* == *CONST Fixrec.run oo* (*-Case1* (*-lazy-pat p*) *r*)

lemma *wild-pat* [*simp*]: *branch wild-pat·*(*unit-when·r*)·*x* = *return·r*
by (*simp add*: *branch-def wild-pat-def*)

lemma *as-pat* [*simp*]:
branch (*as-pat p*)·(*csplit·r*)·*x* = *branch p·*(*r·x*)·*x*

apply (*simp add*: *branch-def as-pat-def*)
apply (*rule-tac p=p·x* **in** *maybeE*, *simp-all*)
done

lemma *lazy-pat* [*simp*]:
branch p·r·x = ⊥ ⟹ branch (lazy-pat p)·r·x = return·(r·⊥)
branch p·r·x = fail ⟹ branch (lazy-pat p)·r·x = return·(r·⊥)
branch p·r·x = return·s ⟹ branch (lazy-pat p)·r·x = return·s
apply (*simp-all add*: *branch-def lazy-pat-def*)
apply (*rule-tac* [!] *p=p·x* **in** *maybeE*, *simp-all*)
done

19.6 Match functions for built-in types

defaultsort *pcpo*

definition
match-UU :: *'a → unit maybe* **where**
match-UU = (Λ *x*. *fail*)

definition
match-cpair :: *'a*::*cpo* × *'b*::*cpo* → (*'a* × *'b*) *maybe* **where**
match-cpair = *csplit*·(Λ *x y*. *return*·<*x*,*y*>)

definition
match-spair :: *'a* ⊗ *'b* → (*'a* × *'b*) *maybe* **where**
match-spair = *ssplit*·(Λ *x y*. *return*·<*x*,*y*>)

definition
match-sinl :: *'a* ⊕ *'b* → *'a maybe* **where**
match-sinl = *sscase*·*return*·(Λ *y*. *fail*)

definition
match-sinr :: *'a* ⊕ *'b* → *'b maybe* **where**
match-sinr = *sscase*·(Λ *x*. *fail*)·*return*

definition
match-up :: *'a*::*cpo u* → *'a maybe* **where**
match-up = *fup*·*return*

definition
match-ONE :: *one* → *unit maybe* **where**
match-ONE = (Λ *ONE*. *return*·())

definition
match-TT :: *tr* → *unit maybe* **where**
match-TT = (Λ *b*. *If b then return*·() *else fail fi*)

definition

match-FF :: *tr* → *unit maybe* **where**
match-FF = (Λ *b*. *If b then fail else return*·() *fi*)

lemma *match-UU-simps* [*simp*]:
match-UU·*x* = *fail*
by (*simp add*: *match-UU-def*)

lemma *match-cpair-simps* [*simp*]:
match-cpair·<*x*,*y*> = *return*·<*x*,*y*>
by (*simp add*: *match-cpair-def*)

lemma *match-spair-simps* [*simp*]:
⟦*x* ≠ ⊥; *y* ≠ ⊥⟧ ⟹ *match-spair*·(:*x*,*y*:) = *return*·<*x*,*y*>
match-spair·⊥ = ⊥
by (*simp-all add*: *match-spair-def*)

lemma *match-sinl-simps* [*simp*]:
x ≠ ⊥ ⟹ *match-sinl*·(*sinl*·*x*) = *return*·*x*
x ≠ ⊥ ⟹ *match-sinl*·(*sinr*·*x*) = *fail*
match-sinl·⊥ = ⊥
by (*simp-all add*: *match-sinl-def*)

lemma *match-sinr-simps* [*simp*]:
x ≠ ⊥ ⟹ *match-sinr*·(*sinr*·*x*) = *return*·*x*
x ≠ ⊥ ⟹ *match-sinr*·(*sinl*·*x*) = *fail*
match-sinr·⊥ = ⊥
by (*simp-all add*: *match-sinr-def*)

lemma *match-up-simps* [*simp*]:
match-up·(*up*·*x*) = *return*·*x*
match-up·⊥ = ⊥
by (*simp-all add*: *match-up-def*)

lemma *match-ONE-simps* [*simp*]:
match-ONE·*ONE* = *return*·()
match-ONE·⊥ = ⊥
by (*simp-all add*: *match-ONE-def*)

lemma *match-TT-simps* [*simp*]:
match-TT·*TT* = *return*·()
match-TT·*FF* = *fail*
match-TT·⊥ = ⊥
by (*simp-all add*: *match-TT-def*)

lemma *match-FF-simps* [*simp*]:
match-FF·*FF* = *return*·()
match-FF·*TT* = *fail*
match-FF·⊥ = ⊥
by (*simp-all add*: *match-FF-def*)

19.7 Mutual recursion

The following rules are used to prove unfolding theorems from fixed-point definitions of mutually recursive functions.

lemma *cpair-equalI*: $[\![x \equiv cfst{\cdot}p;\ y \equiv csnd{\cdot}p]\!] \Longrightarrow <x,y> \equiv p$
by (*simp add*: *surjective-pairing-Cprod2*)

lemma *cpair-eqD1*: $<x,y> = <x',y'> \Longrightarrow x = x'$
by *simp*

lemma *cpair-eqD2*: $<x,y> = <x',y'> \Longrightarrow y = y'$
by *simp*

lemma for proving rewrite rules

lemma *ssubst-lhs*: $[\![t = s;\ P\ s = Q]\!] \Longrightarrow P\ t = Q$
by *simp*

19.8 Initializing the fixrec package

use *Tools/fixrec-package.ML*

hide (**open**) *const return bind fail run*

end

20 Domain: Domain package

theory *Domain*
imports *Ssum Sprod Up One Tr Fixrec*
begin

defaultsort *pcpo*

20.1 Continuous isomorphisms

A locale for continuous isomorphisms

locale *iso* =
  fixes *abs* :: $'a \to 'b$
  fixes *rep* :: $'b \to 'a$
  assumes *abs-iso* [*simp*]: $rep{\cdot}(abs{\cdot}x) = x$
  assumes *rep-iso* [*simp*]: $abs{\cdot}(rep{\cdot}y) = y$
begin

lemma *swap*: *iso rep abs*
  by (*rule iso.intro* [*OF rep-iso abs-iso*])

lemma *abs-less*: $(abs{\cdot}x \sqsubseteq abs{\cdot}y) = (x \sqsubseteq y)$

proof
  assume $abs{\cdot}x \sqsubseteq abs{\cdot}y$
  then have $rep{\cdot}(abs{\cdot}x) \sqsubseteq rep{\cdot}(abs{\cdot}y)$ **by** (*rule monofun-cfun-arg*)
  then show $x \sqsubseteq y$ **by** *simp*
next
  assume $x \sqsubseteq y$
  then show $abs{\cdot}x \sqsubseteq abs{\cdot}y$ **by** (*rule monofun-cfun-arg*)
qed

lemma *rep-less*: $(rep{\cdot}x \sqsubseteq rep{\cdot}y) = (x \sqsubseteq y)$
  by (*rule iso.abs-less* [*OF swap*])

lemma *abs-eq*: $(abs{\cdot}x = abs{\cdot}y) = (x = y)$
  by (*simp add*: *po-eq-conv abs-less*)

lemma *rep-eq*: $(rep{\cdot}x = rep{\cdot}y) = (x = y)$
  by (*rule iso.abs-eq* [*OF swap*])

lemma *abs-strict*: $abs{\cdot}\bot = \bot$
proof −
  have $\bot \sqsubseteq rep{\cdot}\bot$ **..**
  then have $abs{\cdot}\bot \sqsubseteq abs{\cdot}(rep{\cdot}\bot)$ **by** (*rule monofun-cfun-arg*)
  then have $abs{\cdot}\bot \sqsubseteq \bot$ **by** *simp*
  then show *?thesis* **by** (*rule UU-I*)
qed

lemma *rep-strict*: $rep{\cdot}\bot = \bot$
  by (*rule iso.abs-strict* [*OF swap*])

lemma *abs-defin'*: $abs{\cdot}x = \bot \Longrightarrow x = \bot$
proof −
  have $x = rep{\cdot}(abs{\cdot}x)$ **by** *simp*
  also assume $abs{\cdot}x = \bot$
  also note *rep-strict*
  finally show $x = \bot$ **.**
qed

lemma *rep-defin'*: $rep{\cdot}z = \bot \Longrightarrow z = \bot$
  by (*rule iso.abs-defin'* [*OF swap*])

lemma *abs-defined*: $z \neq \bot \Longrightarrow abs{\cdot}z \neq \bot$
  by (*erule contrapos-nn*, *erule abs-defin'*)

lemma *rep-defined*: $z \neq \bot \Longrightarrow rep{\cdot}z \neq \bot$
  by (*rule iso.abs-defined* [*OF iso.swap*]) (*rule iso-axioms*)

lemma *abs-defined-iff*: $(abs{\cdot}x = \bot) = (x = \bot)$
  by (*auto elim*: *abs-defin' intro*: *abs-strict*)

lemma *rep-defined-iff*: $(rep{\cdot}x = \bot) = (x = \bot)$
by (*rule iso.abs-defined-iff* [*OF iso.swap*]) (*rule iso-axioms*)

lemma (**in** *iso*) *compact-abs-rev*: *compact* $(abs{\cdot}x) \Longrightarrow$ *compact* x
proof (*unfold compact-def*)
assume *adm* $(\lambda y.\ \neg\ abs{\cdot}x \sqsubseteq y)$
with *cont-Rep-CFun2*
have *adm* $(\lambda y.\ \neg\ abs{\cdot}x \sqsubseteq abs{\cdot}y)$ **by** (*rule adm-subst*)
then show *adm* $(\lambda y.\ \neg\ x \sqsubseteq y)$ **using** *abs-less* **by** *simp*
qed

lemma *compact-rep-rev*: *compact* $(rep{\cdot}x) \Longrightarrow$ *compact* x
by (*rule iso.compact-abs-rev* [*OF iso.swap*]) (*rule iso-axioms*)

lemma *compact-abs*: *compact* $x \Longrightarrow$ *compact* $(abs{\cdot}x)$
by (*rule compact-rep-rev*) *simp*

lemma *compact-rep*: *compact* $x \Longrightarrow$ *compact* $(rep{\cdot}x)$
by (*rule iso.compact-abs* [*OF iso.swap*]) (*rule iso-axioms*)

lemma *iso-swap*: $(x = abs{\cdot}y) = (rep{\cdot}x = y)$
proof
assume $x = abs{\cdot}y$
then have $rep{\cdot}x = rep{\cdot}(abs{\cdot}y)$ **by** *simp*
then show $rep{\cdot}x = y$ **by** *simp*
next
assume $rep{\cdot}x = y$
then have $abs{\cdot}(rep{\cdot}x) = abs{\cdot}y$ **by** *simp*
then show $x = abs{\cdot}y$ **by** *simp*
qed

end

20.2 Casedist

lemma *ex-one-defined-iff*:
$(\exists x.\ P\ x \wedge x \neq \bot) = P\ ONE$
apply *safe*
apply (*rule-tac p=x* **in** *oneE*)
apply *simp*
apply *simp*
apply *force*
done

lemma *ex-up-defined-iff*:
$(\exists x.\ P\ x \wedge x \neq \bot) = (\exists x.\ P\ (up{\cdot}x))$
apply *safe*
apply (*rule-tac p=x* **in** *upE*)
apply *simp*

```
 apply fast
apply (force intro!: up-defined)
done

lemma ex-sprod-defined-iff:
(∃ y. P y ∧ y ≠ ⊥) =
 (∃ x y. (P (:x, y:) ∧ x ≠ ⊥) ∧ y ≠ ⊥)
apply safe
 apply (rule-tac p=y in sprodE)
  apply simp
 apply fast
apply (force intro!: spair-defined)
done

lemma ex-sprod-up-defined-iff:
(∃ y. P y ∧ y ≠ ⊥) =
 (∃ x y. P (:up·x, y:) ∧ y ≠ ⊥)
apply safe
 apply (rule-tac p=y in sprodE)
  apply simp
 apply (rule-tac p=x in upE)
  apply simp
 apply fast
apply (force intro!: spair-defined)
done

lemma ex-ssum-defined-iff:
(∃ x. P x ∧ x ≠ ⊥) =
((∃ x. P (sinl·x) ∧ x ≠ ⊥) ∨
 (∃ x. P (sinr·x) ∧ x ≠ ⊥))
apply (rule iffI)
 apply (erule exE)
 apply (erule conjE)
 apply (rule-tac p=x in ssumE)
   apply simp
  apply (rule disjI1, fast)
 apply (rule disjI2, fast)
apply (erule disjE)
 apply force
apply force
done

lemma exh-start: p = ⊥ ∨ (∃ x. p = x ∧ x ≠ ⊥)
 by auto

lemmas ex-defined-iffs =
   ex-ssum-defined-iff
   ex-sprod-up-defined-iff
   ex-sprod-defined-iff
```

ex-up-defined-iff
ex-one-defined-iff

Rules for turning exh into casedist

lemma *exh-casedist0*: $[\![R;\ R \Longrightarrow P]\!] \Longrightarrow P$
by *auto*

lemma *exh-casedist1*: $((P \vee Q \Longrightarrow R) \Longrightarrow S) \equiv ([\![P \Longrightarrow R;\ Q \Longrightarrow R]\!] \Longrightarrow S)$
by *rule auto*

lemma *exh-casedist2*: $(\exists x.\ P\ x \Longrightarrow Q) \equiv (\bigwedge x.\ P\ x \Longrightarrow Q)$
by *rule auto*

lemma *exh-casedist3*: $(P \wedge Q \Longrightarrow R) \equiv (P \Longrightarrow Q \Longrightarrow R)$
by *rule auto*

lemmas *exh-casedists* = *exh-casedist1 exh-casedist2 exh-casedist3*

end

21 SetPcpo: Set as a pointed cpo

theory *SetPcpo*
imports *Adm*
begin

instantiation *bool* :: *po*
begin

definition
less-bool-def: $(op \sqsubseteq) = (op \longrightarrow)$

instance
by (*intro-classes*, *unfold less-bool-def*, *safe*)

end

lemma *less-set-eq*: $(op \sqsubseteq) = (op \subseteq)$
by (*simp add*: *less-fun-def less-bool-def le-fun-def le-bool-def expand-fun-eq*)

instance *bool* :: *finite-po* **..**

lemma *Union-is-lub*: $A <<| Union\ A$
unfolding *is-lub-def is-ub-def less-set-eq* **by** *fast*

instance *bool* :: *cpo* **..**

lemma *lub-eq-Union*: *lub* = *Union*

by (*rule ext*, *rule thelubI* [*OF Union-is-lub*])

instance *bool* :: *pcpo*
proof
  have $\forall\, y{::}bool.\ False \sqsubseteq y$ **unfolding** *less-bool-def* **by** *simp*
  thus $\exists\, x{::}bool.\ \forall\, y.\ x \sqsubseteq y$ **..**
qed

lemma *UU-eq-empty*: $\bot = \{\}$
by (*rule UU-I* [*symmetric*], *simp add*: *less-set-eq*)

lemmas *set-cpo-simps* = *less-set-eq lub-eq-Union UU-eq-empty*

21.1 Admissibility of set predicates

lemma *adm-nonempty*: *adm* $(\lambda A.\ \exists\, x.\ x \in A)$
by (*rule admI*, *force simp add*: *lub-eq-Union*)

lemma *adm-in*: *adm* $(\lambda A.\ x \in A)$
by (*rule admI*, *simp add*: *lub-eq-Union*)

lemma *adm-not-in*: *adm* $(\lambda A.\ x \notin A)$
by (*rule admI*, *simp add*: *lub-eq-Union*)

lemma *adm-Ball*: $(\bigwedge x.\ adm\ (\lambda A.\ P\ A\ x)) \Longrightarrow adm\ (\lambda A.\ \forall\, x{\in}A.\ P\ A\ x)$
unfolding *Ball-def* **by** (*simp add*: *adm-not-in*)

lemma *adm-Bex*: *adm* $(\lambda A.\ Bex\ A\ P)$
by (*rule admI*, *simp add*: *lub-eq-Union*)

lemma *adm-subset*: *adm* $(\lambda A.\ A \subseteq B)$
by (*rule admI*, *auto simp add*: *lub-eq-Union*)

lemma *adm-superset*: *adm* $(\lambda A.\ B \subseteq A)$
by (*rule admI*, *auto simp add*: *lub-eq-Union*)

lemmas *adm-set-lemmas* =
  adm-nonempty adm-in adm-not-in adm-Bex adm-Ball adm-subset adm-superset

21.2 Compactness

lemma *compact-empty*: *compact* $\{\}$
by (*fold UU-eq-empty*, *simp*)

lemma *compact-insert*: *compact* $A \Longrightarrow$ *compact* $(insert\ x\ A)$
unfolding *compact-def set-cpo-simps*
by (*simp add*: *adm-set-lemmas*)

lemma *finite-imp-compact*: *finite* $A \Longrightarrow$ *compact* A
by (*induct A set*: *finite*, *rule compact-empty*, *erule compact-insert*)

end

22 CompactBasis: Compact bases of domains

theory *CompactBasis*
imports *Bifinite SetPcpo*
begin

22.1 Ideals over a preorder

context *preorder*
begin

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

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

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

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

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

lemma *ideal-directed-finite*:
  assumes *A*: *ideal A*
  shows ⟦*finite U*; *U* ⊆ *A*⟧ ⟹ ∃ *z*∈*A*. ∀ *x*∈*U*. *x* ⊑ *z*
apply (*induct U set*: *finite*)
apply (*simp add*: *idealD1* [*OF A*])
apply (*simp*, *clarify*, *rename-tac y*)
apply (*drule* (*1*) *idealD2* [*OF A*])
apply (*clarify*, *erule-tac x*=*z* **in** *rev-bexI*)
apply (*fast intro*: *trans-less*)
done

lemma *ideal-principal*: *ideal* $\{x.\ x \sqsubseteq z\}$
apply (*rule idealI*)
apply (*rule-tac x=z* **in** *exI*)
apply *fast*
apply (*rule-tac x=z* **in** *bexI*, *fast*)
apply *fast*
apply (*fast intro*: *trans-less*)
done

lemma *directed-image-ideal*:
  assumes *A*: *ideal A*
  assumes *f*: $\bigwedge x\ y.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$
  shows *directed* (*f* ' *A*)
apply (*rule directedI*)
apply (*cut-tac idealD1* [*OF A*], *fast*)
apply (*clarify*, *rename-tac a b*)
apply (*drule* (*1*) *idealD2* [*OF A*])
apply (*clarify*, *rename-tac c*)
apply (*rule-tac x=f c* **in** *rev-bexI*)
apply (*erule imageI*)
apply (*simp add*: *f*)
done

lemma *adm-ideal*: *adm* ($\lambda A.$ *ideal A*)
unfolding *ideal-def* **by** (*intro adm-lemmas adm-set-lemmas*)

lemma *lub-image-principal*:
  assumes *f*: $\bigwedge x\ y.\ x \sqsubseteq y \Longrightarrow f\ x \sqsubseteq f\ y$
  shows $(\bigsqcup x{\in}\{x.\ x \sqsubseteq y\}.\ f\ x) = f\ y$
apply (*rule thelubI*)
apply (*rule is-lub-maximal*)
apply (*rule ub-imageI*)
apply (*simp add*: *f*)
apply (*rule imageI*)
apply *simp*
done

end

22.2 Defining functions in terms of basis elements

lemma *finite-directed-contains-lub*:
  $[\![$*finite S*; *directed S*$]\!] \Longrightarrow \exists u{\in}S.\ S <<|\ u$
apply (*drule* (*1*) *directed-finiteD*, *rule subset-refl*)
apply (*erule bexE*)
apply (*rule rev-bexI*, *assumption*)
apply (*erule* (*1*) *is-lub-maximal*)
done

lemma *lub-finite-directed-in-self*:
  $[\![\mathit{finite}\ S;\ \mathit{directed}\ S]\!] \Longrightarrow \mathit{lub}\ S \in S$
apply (*drule* (*1*) *finite-directed-contains-lub*, *clarify*)
apply (*drule thelubI*, *simp*)
done

lemma *finite-directed-has-lub*: $[\![\mathit{finite}\ S;\ \mathit{directed}\ S]\!] \Longrightarrow \exists\, u.\ S <<|\ u$
by (*drule* (*1*) *finite-directed-contains-lub*, *fast*)

lemma *is-ub-thelub0*: $[\![\exists\, u.\ S <<|\ u;\ x \in S]\!] \Longrightarrow x \sqsubseteq \mathit{lub}\ S$
apply (*erule exE*, *drule lubI*)
apply (*drule is-lubD1*)
apply (*erule* (*1*) *is-ubD*)
done

lemma *is-lub-thelub0*: $[\![\exists\, u.\ S <<|\ u;\ S <|\ x]\!] \Longrightarrow \mathit{lub}\ S \sqsubseteq x$
by (*erule exE*, *drule lubI*, *erule is-lub-lub*)

locale *basis-take* = *preorder r* +
  fixes *take* :: *nat* ⇒ *'a*::*type* ⇒ *'a*
  assumes *take-less*: *r* (*take n a*) *a*
  assumes *take-take*: *take n* (*take n a*) = *take n a*
  assumes *take-mono*: *r a b* ⟹ *r* (*take n a*) (*take n b*)
  assumes *take-chain*: *r* (*take n a*) (*take* (*Suc n*) *a*)
  assumes *finite-range-take*: *finite* (*range* (*take n*))
  assumes *take-covers*: ∃ *n*. *take n a* = *a*

locale *ideal-completion* = *basis-take r* +
  fixes *principal* :: *'a*::*type* ⇒ *'b*::*cpo*
  fixes *rep* :: *'b*::*cpo* ⇒ *'a*::*type set*
  assumes *ideal-rep*: ⋀*x*. *preorder.ideal r* (*rep x*)
  assumes *cont-rep*: *cont rep*
  assumes *rep-principal*: ⋀*a*. *rep* (*principal a*) = {*b*. *r b a*}
  assumes *subset-repD*: ⋀*x y*. *rep x* ⊆ *rep y* ⟹ *x* ⊑ *y*
begin

lemma *finite-take-rep*: *finite* (*take n* ‘ *rep x*)
by (*rule finite-subset* [*OF image-mono* [*OF subset-UNIV*] *finite-range-take*])

lemma *basis-fun-lemma0*:
  fixes *f* :: *'a*::*type* ⇒ *'c*::*cpo*
  assumes *f-mono*: ⋀*a b*. *r a b* ⟹ *f a* ⊑ *f b*
  shows ∃ *u*. *f* ‘ *take i* ‘ *rep x* <<| *u*
apply (*rule finite-directed-has-lub*)
apply (*rule finite-imageI*)
apply (*rule finite-take-rep*)
apply (*subst image-image*)
apply (*rule directed-image-ideal*)

apply (*rule ideal-rep*)
apply (*rule f-mono*)
apply (*erule take-mono*)
done

lemma *basis-fun-lemma1*:
  fixes f :: $'a$::*type* ⇒ $'c$::*cpo*
  assumes *f-mono*: ⋀a b. r a b ⟹ f a ⊑ f b
  shows *chain* (λi. *lub* (f ‘ *take* i ‘ *rep* x))
 apply (*rule chainI*)
 apply (*rule is-lub-thelub0*)
  apply (*rule basis-fun-lemma0*, *erule f-mono*)
 apply (*rule is-ubI*, *clarsimp*, *rename-tac a*)
 apply (*rule sq-le.trans-less* [*OF f-mono* [*OF take-chain*]])
 apply (*rule is-ub-thelub0*)
  apply (*rule basis-fun-lemma0*, *erule f-mono*)
 apply *simp*
done

lemma *basis-fun-lemma2*:
  fixes f :: $'a$::*type* ⇒ $'c$::*cpo*
  assumes *f-mono*: ⋀a b. r a b ⟹ f a ⊑ f b
  shows f ‘ *rep* x <<| (⨆ i. *lub* (f ‘ *take* i ‘ *rep* x))
 apply (*rule is-lubI*)
 apply (*rule ub-imageI*, *rename-tac a*)
  apply (*cut-tac a=a* **in** *take-covers*, *erule exE*, *rename-tac i*)
  apply (*erule subst*)
  apply (*rule rev-trans-less*)
   apply (*rule-tac x=i* **in** *is-ub-thelub*)
   apply (*rule basis-fun-lemma1*, *erule f-mono*)
  apply (*rule is-ub-thelub0*)
   apply (*rule basis-fun-lemma0*, *erule f-mono*)
  apply *simp*
 apply (*rule is-lub-thelub* [*OF - ub-rangeI*])
  apply (*rule basis-fun-lemma1*, *erule f-mono*)
 apply (*rule is-lub-thelub0*)
  apply (*rule basis-fun-lemma0*, *erule f-mono*)
 apply (*rule is-ubI*, *clarsimp*, *rename-tac a*)
 apply (*rule sq-le.trans-less* [*OF f-mono* [*OF take-less*]])
 apply (*erule* (*1*) *ub-imageD*)
done

lemma *basis-fun-lemma*:
  fixes f :: $'a$::*type* ⇒ $'c$::*cpo*
  assumes *f-mono*: ⋀a b. r a b ⟹ f a ⊑ f b
  shows ∃ u. f ‘ *rep* x <<| u
by (*rule exI*, *rule basis-fun-lemma2*, *erule f-mono*)

lemma *rep-mono*: x ⊑ y ⟹ *rep* x ⊆ *rep* y

apply (*drule cont-rep* [*THEN cont2mono*, *THEN monofunE*])
apply (*simp add*: *set-cpo-simps*)
done

lemma *rep-contlub*:
  chain $Y \Longrightarrow$ *rep* $(\bigsqcup i.\ Y\ i) = (\bigcup i.\ rep\ (Y\ i))$
by (*simp add*: *cont2contlubE* [*OF cont-rep*] *set-cpo-simps*)

lemma *less-def*: $x \sqsubseteq y \longleftrightarrow rep\ x \subseteq rep\ y$
by (*rule iffI* [*OF rep-mono subset-repD*])

lemma *rep-eq*: $rep\ x = \{a.\ principal\ a \sqsubseteq x\}$
unfolding *less-def rep-principal*
apply *safe*
apply (*erule* (*1*) *idealD3* [*OF ideal-rep*])
apply (*erule subsetD*, *simp add*: *refl*)
done

lemma *mem-rep-iff-principal-less*: $a \in rep\ x \longleftrightarrow principal\ a \sqsubseteq x$
by (*simp add*: *rep-eq*)

lemma *principal-less-iff-mem-rep*: $principal\ a \sqsubseteq x \longleftrightarrow a \in rep\ x$
by (*simp add*: *rep-eq*)

lemma *principal-less-iff*: $principal\ a \sqsubseteq principal\ b \longleftrightarrow r\ a\ b$
by (*simp add*: *principal-less-iff-mem-rep rep-principal*)

lemma *principal-eq-iff*: $principal\ a = principal\ b \longleftrightarrow r\ a\ b \wedge r\ b\ a$
unfolding *po-eq-conv* [**where** $'a='b$] *principal-less-iff* **..**

lemma *repD*: $a \in rep\ x \Longrightarrow principal\ a \sqsubseteq x$
by (*simp add*: *rep-eq*)

lemma *principal-mono*: $r\ a\ b \Longrightarrow principal\ a \sqsubseteq principal\ b$
by (*simp add*: *principal-less-iff*)

lemma *lessI*: $(\bigwedge a.\ principal\ a \sqsubseteq x \Longrightarrow principal\ a \sqsubseteq u) \Longrightarrow x \sqsubseteq u$
unfolding *principal-less-iff-mem-rep*
by (*simp add*: *less-def subset-eq*)

lemma *lub-principal-rep*: *principal* ‘ $rep\ x <<|\ x$
apply (*rule is-lubI*)
apply (*rule ub-imageI*)
apply (*erule repD*)
apply (*subst less-def*)
apply (*rule subsetI*)
apply (*drule* (*1*) *ub-imageD*)
apply (*simp add*: *rep-eq*)
done

definition
basis-fun :: (*'a*::*type* ⇒ *'c*::*cpo*) ⇒ *'b* → *'c* **where**
basis-fun = (λ*f*. (Λ *x*. *lub* (*f* ‘ *rep* *x*)))

lemma *basis-fun-beta*:
fixes *f* :: *'a*::*type* ⇒ *'c*::*cpo*
assumes *f-mono*: ⋀*a b*. *r a b* ⟹ *f a* ⊑ *f b*
shows *basis-fun f*·*x* = *lub* (*f* ‘ *rep* *x*)
unfolding *basis-fun-def*
proof (*rule beta-cfun*)
have *lub*: ⋀*x*. ∃ *u*. *f* ‘ *rep* *x* <<| *u*
using *f-mono* **by** (*rule basis-fun-lemma*)
show *cont*: *cont* (λ*x*. *lub* (*f* ‘ *rep* *x*))
apply (*rule contI2*)
apply (*rule monofunI*)
apply (*rule is-lub-thelub0* [*OF lub ub-imageI*])
apply (*rule is-ub-thelub0* [*OF lub imageI*])
apply (*erule* (*1*) *subsetD* [*OF rep-mono*])
apply (*rule is-lub-thelub0* [*OF lub ub-imageI*])
apply (*simp add*: *rep-contlub*, *clarify*)
apply (*erule rev-trans-less* [*OF is-ub-thelub*])
apply (*erule is-ub-thelub0* [*OF lub imageI*])
done
qed

lemma *basis-fun-principal*:
fixes *f* :: *'a*::*type* ⇒ *'c*::*cpo*
assumes *f-mono*: ⋀*a b*. *r a b* ⟹ *f a* ⊑ *f b*
shows *basis-fun f*·(*principal a*) = *f a*
apply (*subst basis-fun-beta*, *erule f-mono*)
apply (*subst rep-principal*)
apply (*rule lub-image-principal*, *erule f-mono*)
done

lemma *basis-fun-mono*:
assumes *f-mono*: ⋀*a b*. *r a b* ⟹ *f a* ⊑ *f b*
assumes *g-mono*: ⋀*a b*. *r a b* ⟹ *g a* ⊑ *g b*
assumes *less*: ⋀*a*. *f a* ⊑ *g a*
shows *basis-fun f* ⊑ *basis-fun g*
apply (*rule less-cfun-ext*)
apply (*simp only*: *basis-fun-beta f-mono g-mono*)
apply (*rule is-lub-thelub0*)
apply (*rule basis-fun-lemma*, *erule f-mono*)
apply (*rule ub-imageI*, *rename-tac a*)
apply (*rule sq-le.trans-less* [*OF less*])
apply (*rule is-ub-thelub0*)
apply (*rule basis-fun-lemma*, *erule g-mono*)
apply (*erule imageI*)

done

lemma *compact-principal*: *compact* (*principal a*)
by (*rule compactI2*, *simp add*: *principal-less-iff-mem-rep rep-contlub*)

definition
completion-approx :: *nat* ⇒ *'b* → *'b* **where**
completion-approx = (λ*i*. *basis-fun* (λ*a*. *principal* (*take i a*)))

lemma *completion-approx-beta*:
completion-approx i·*x* = (⨆ *a*∈*rep x*. *principal* (*take i a*))
unfolding *completion-approx-def*
by (*simp add*: *basis-fun-beta principal-mono take-mono*)

lemma *completion-approx-principal*:
completion-approx i·(*principal a*) = *principal* (*take i a*)
unfolding *completion-approx-def*
by (*simp add*: *basis-fun-principal principal-mono take-mono*)

lemma *chain-completion-approx*: *chain completion-approx*
unfolding *completion-approx-def*
apply (*rule chainI*)
apply (*rule basis-fun-mono*)
apply (*erule principal-mono* [*OF take-mono*])
apply (*erule principal-mono* [*OF take-mono*])
apply (*rule principal-mono* [*OF take-chain*])
done

lemma *lub-completion-approx*: (⨆ *i*. *completion-approx i*·*x*) = *x*
unfolding *completion-approx-beta*
apply (*subst image-image* [**where** *f*=*principal*, *symmetric*])
apply (*rule unique-lub* [*OF* - *lub-principal-rep*])
apply (*rule basis-fun-lemma2*, *erule principal-mono*)
done

lemma *completion-approx-eq-principal*:
∃ *a*∈*rep x*. *completion-approx i*·*x* = *principal* (*take i a*)
unfolding *completion-approx-beta*
apply (*subst image-image* [**where** *f*=*principal*, *symmetric*])
apply (*subgoal-tac finite* (*principal* ‘ *take i* ‘ *rep x*))
apply (*subgoal-tac directed* (*principal* ‘ *take i* ‘ *rep x*))
apply (*drule* (*1*) *lub-finite-directed-in-self*, *fast*)
apply (*subst image-image*)
apply (*rule directed-image-ideal*)
apply (*rule ideal-rep*)
apply (*erule principal-mono* [*OF take-mono*])
apply (*rule finite-imageI*)
apply (*rule finite-take-rep*)
done

lemma *completion-approx-idem*:
completion-approx i·(completion-approx i·x) = completion-approx i·x
using *completion-approx-eq-principal* [**where** *i=i* **and** *x=x*]
by (*auto simp add*: *completion-approx-principal take-take*)

lemma *finite-fixes-completion-approx*:
finite {*x*. *completion-approx i·x = x*} (**is** *finite ?S*)
apply (*subgoal-tac ?S ⊆ principal ' range* (*take i*))
apply (*erule finite-subset*)
apply (*rule finite-imageI*)
apply (*rule finite-range-take*)
apply (*clarify*, *erule subst*)
apply (*cut-tac x=x* **and** *i=i* **in** *completion-approx-eq-principal*)
apply *fast*
done

lemma *principal-induct*:
assumes *adm*: *adm P*
assumes *P*: ⋀*a*. *P* (*principal a*)
shows *P x*
apply (*subgoal-tac P* (⨆ *i*. *completion-approx i·x*))
apply (*simp add*: *lub-completion-approx*)
apply (*rule admD* [*OF adm*])
apply (*simp add*: *chain-completion-approx*)
apply (*cut-tac x=x* **and** *i=i* **in** *completion-approx-eq-principal*)
apply (*clarify*, *simp add*: *P*)
done

end

22.3 Compact bases of bifinite domains

defaultsort *profinite*

typedef (**open**) *'a compact-basis* = {*x*::*'a*::*profinite*. *compact x*}
by (*fast intro*: *compact-approx*)

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

lemma *Rep-Abs-compact-basis-approx* [*simp*]:
Rep-compact-basis (*Abs-compact-basis* (*approx n·x*)) = *approx n·x*
by (*rule Abs-compact-basis-inverse*, *simp*)

lemma *compact-imp-Rep-compact-basis*:
compact x ⟹ ∃ *y*. *x = Rep-compact-basis y*
by (*rule exI*, *rule Abs-compact-basis-inverse* [*symmetric*], *simp*)

instantiation *compact-basis* :: (*profinite*) *sq-ord*
begin

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

instance ..

end

instance *compact-basis* :: (*profinite*) *po*
by (*rule typedef-po*
[*OF type-definition-compact-basis compact-le-def*])

minimal compact element

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

lemma *Rep-compact-bot*: *Rep-compact-basis compact-bot* = $\bot$
unfolding *compact-bot-def* **by** (*simp add*: *Abs-compact-basis-inverse*)

lemma *compact-minimal* [*simp*]: *compact-bot* $\sqsubseteq$ *a*
unfolding *compact-le-def Rep-compact-bot* **by** *simp*

compacts

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

lemma *ideal-compacts*: *preorder.ideal sq-le* (*compacts w*)
unfolding *compacts-def*
apply (*rule preorder.idealI*)
apply (*rule preorder-class.axioms*)
apply (*rule-tac x=Abs-compact-basis* (*approx 0*·*w*) **in** *exI*)
apply (*simp add*: *approx-less*)
apply *simp*
apply (*cut-tac a=x* **in** *compact-Rep-compact-basis*)
apply (*cut-tac a=y* **in** *compact-Rep-compact-basis*)
apply (*drule bifinite-compact-eq-approx*)
apply (*drule bifinite-compact-eq-approx*)
apply (*clarify*, *rename-tac i j*)
apply (*rule-tac x=Abs-compact-basis* (*approx* (*max i j*)·*w*) **in** *exI*)
apply (*simp add*: *approx-less compact-le-def*)
apply (*erule subst*, *erule subst*)
apply (*simp add*: *monofun-cfun chain-mono* [*OF chain-approx*])
apply (*simp add*: *compact-le-def*)
apply (*erule* (*1*) *trans-less*)

done

lemma *compacts-Rep-compact-basis*:
compacts (*Rep-compact-basis b*) = {*a*. *a* ⊑ *b*}
unfolding *compacts-def compact-le-def* **..**

lemma *cont-compacts*: *cont compacts*
unfolding *compacts-def*
apply (*rule contI2*)
apply (*rule monofunI*)
apply (*simp add*: *set-cpo-simps*)
apply (*fast intro*: *trans-less*)
apply (*simp add*: *set-cpo-simps*)
apply *clarify*
apply *simp*
apply (*erule* (*1*) *compactD2* [*OF compact-Rep-compact-basis*])
done

lemma *compacts-lessD*: *compacts x* ⊆ *compacts y* ⟹ *x* ⊑ *y*
apply (*subgoal-tac* (⨆ *i*. *approx i*·*x*) ⊑ *y*, *simp*)
apply (*rule admD*, *simp*, *simp*)
apply (*drule-tac c*=*Abs-compact-basis* (*approx i*·*x*) **in** *subsetD*)
apply (*simp add*: *compacts-def Abs-compact-basis-inverse approx-less*)
apply (*simp add*: *compacts-def Abs-compact-basis-inverse*)
done

lemma *compacts-mono*: *x* ⊑ *y* ⟹ *compacts x* ⊆ *compacts y*
unfolding *compacts-def* **by** (*fast intro*: *trans-less*)

lemma *less-compact-basis-iff*: (*x* ⊑ *y*) = (*compacts x* ⊆ *compacts y*)
by (*rule iffI* [*OF compacts-mono compacts-lessD*])

lemma *compact-basis-induct*:
⟦*adm P*; ⋀*a*. *P* (*Rep-compact-basis a*)⟧ ⟹ *P x*
apply (*erule approx-induct*)
apply (*drule-tac x*=*Abs-compact-basis* (*approx n*·*x*) **in** *meta-spec*)
apply (*simp add*: *Abs-compact-basis-inverse*)
done

approximation on compact bases

definition
compact-approx :: *nat* ⇒ *'a compact-basis* ⇒ *'a compact-basis* **where**
compact-approx = (λ*n a*. *Abs-compact-basis* (*approx n*·(*Rep-compact-basis a*)))

lemma *Rep-compact-approx*:
Rep-compact-basis (*compact-approx n a*) = *approx n*·(*Rep-compact-basis a*)
unfolding *compact-approx-def*
by (*simp add*: *Abs-compact-basis-inverse*)

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

lemma *compact-approx-le*: *compact-approx n a* $\sqsubseteq$ *a*
unfolding *compact-le-def*
by (*simp add*: *Rep-compact-approx approx-less*)

lemma *compact-approx-mono1*:
$i \leq j \Longrightarrow$ *compact-approx i a* $\sqsubseteq$ *compact-approx j a*
unfolding *compact-le-def*
apply (*simp add*: *Rep-compact-approx*)
apply (*rule chain-mono*, *simp*, *assumption*)
done

lemma *compact-approx-mono*:
a $\sqsubseteq$ *b* $\Longrightarrow$ *compact-approx n a* $\sqsubseteq$ *compact-approx n b*
unfolding *compact-le-def*
apply (*simp add*: *Rep-compact-approx*)
apply (*erule monofun-cfun-arg*)
done

lemma *ex-compact-approx-eq*: $\exists$ *n*. *compact-approx n a* = *a*
apply (*simp add*: *Rep-compact-basis-inject* [*symmetric*])
apply (*simp add*: *Rep-compact-approx*)
apply (*rule bifinite-compact-eq-approx*)
apply (*rule compact-Rep-compact-basis*)
done

lemma *compact-approx-idem*:
compact-approx n (*compact-approx n a*) = *compact-approx n a*
apply (*rule Rep-compact-basis-inject* [*THEN iffD1*])
apply (*simp add*: *Rep-compact-approx*)
done

lemma *finite-fixes-compact-approx*: *finite* {*a*. *compact-approx n a* = *a*}
apply (*subgoal-tac finite* (*Rep-compact-basis* ' {*a*. *compact-approx n a* = *a*}))
apply (*erule finite-imageD*)
apply (*rule inj-onI*, *simp add*: *Rep-compact-basis-inject*)
apply (*rule finite-subset* [*OF - finite-fixes-approx* [**where** *i*=*n*]])
apply (*rule subsetI*, *clarify*, *rename-tac a*)
apply (*simp add*: *Rep-compact-basis-inject* [*symmetric*])
apply (*simp add*: *Rep-compact-approx*)
done

lemma *finite-range-compact-approx*: *finite* (*range* (*compact-approx n*))
apply (*cut-tac n*=*n* **in** *finite-fixes-compact-approx*)
apply (*simp add*: *idem-fixes-eq-range compact-approx-idem*)
apply (*simp add*: *image-def*)
done

interpretation *compact-basis*:
  ideal-completion [*sq-le compact-approx Rep-compact-basis compacts*]
proof (*unfold-locales*)
  fix *n* :: *nat* **and** *a b* :: *'a compact-basis* **and** *x* :: *'a*
  show *compact-approx n a* ⊑ *a*
    by (*rule compact-approx-le*)
  show *compact-approx n* (*compact-approx n a*) = *compact-approx n a*
    by (*rule compact-approx-idem*)
  show *compact-approx n a* ⊑ *compact-approx* (*Suc n*) *a*
    by (*rule compact-approx-mono1*, *simp*)
  show *finite* (*range* (*compact-approx n*))
    by (*rule finite-range-compact-approx*)
  show ∃ *n*::*nat*. *compact-approx n a* = *a*
    by (*rule ex-compact-approx-eq*)
  show *preorder.ideal sq-le* (*compacts x*)
    by (*rule ideal-compacts*)
  show *cont compacts*
    by (*rule cont-compacts*)
  show *compacts* (*Rep-compact-basis a*) = {*b*. *b* ⊑ *a*}
    by (*rule compacts-Rep-compact-basis*)
next
  fix *n* :: *nat* **and** *a b* :: *'a compact-basis*
  assume *a* ⊑ *b*
  thus *compact-approx n a* ⊑ *compact-approx n b*
    by (*rule compact-approx-mono*)
next
  fix *x y* :: *'a*
  assume *compacts x* ⊆ *compacts y* **thus** *x* ⊑ *y*
    by (*rule compacts-lessD*)
qed

22.4 A compact basis for powerdomains

typedef *'a pd-basis* =
  {*S*::*'a*::*profinite compact-basis set*. *finite S* ∧ *S* ≠ {}}
by (*rule-tac x*={*arbitrary*} **in** *exI*, *simp*)

lemma *finite-Rep-pd-basis* [*simp*]: *finite* (*Rep-pd-basis u*)
by (*insert Rep-pd-basis* [*of u*, *unfolded pd-basis-def*]) *simp*

lemma *Rep-pd-basis-nonempty* [*simp*]: *Rep-pd-basis u* ≠ {}
by (*insert Rep-pd-basis* [*of u*, *unfolded pd-basis-def*]) *simp*

unit and plus

definition
  PDUnit :: *'a compact-basis* ⇒ *'a pd-basis* **where**
  PDUnit = (λ*x*. *Abs-pd-basis* {*x*})

definition

PDPlus :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *'a pd-basis* **where**
PDPlus t u = *Abs-pd-basis* (*Rep-pd-basis t* ∪ *Rep-pd-basis u*)

lemma *Rep-PDUnit*:
Rep-pd-basis (*PDUnit x*) = {*x*}
unfolding *PDUnit-def* **by** (*rule Abs-pd-basis-inverse*) (*simp add*: *pd-basis-def*)

lemma *Rep-PDPlus*:
Rep-pd-basis (*PDPlus u v*) = *Rep-pd-basis u* ∪ *Rep-pd-basis v*
unfolding *PDPlus-def* **by** (*rule Abs-pd-basis-inverse*) (*simp add*: *pd-basis-def*)

lemma *PDUnit-inject* [*simp*]: (*PDUnit a* = *PDUnit b*) = (*a* = *b*)
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDUnit* **by** *simp*

lemma *PDPlus-assoc*: *PDPlus* (*PDPlus t u*) *v* = *PDPlus t* (*PDPlus u v*)
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDPlus* **by** (*rule Un-assoc*)

lemma *PDPlus-commute*: *PDPlus t u* = *PDPlus u t*
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDPlus* **by** (*rule Un-commute*)

lemma *PDPlus-absorb*: *PDPlus t t* = *t*
unfolding *Rep-pd-basis-inject* [*symmetric*] *Rep-PDPlus* **by** (*rule Un-absorb*)

lemma *pd-basis-induct1*:
assumes *PDUnit*: ⋀*a*. *P* (*PDUnit a*)
assumes *PDPlus*: ⋀*a t*. *P t* ⟹ *P* (*PDPlus* (*PDUnit a*) *t*)
shows *P x*
apply (*induct x*, *unfold pd-basis-def*, *clarify*)
apply (*erule* (*1*) *finite-ne-induct*)
apply (*cut-tac a*=*x* **in** *PDUnit*)
apply (*simp add*: *PDUnit-def*)
apply (*drule-tac a*=*x* **in** *PDPlus*)
apply (*simp add*: *PDUnit-def PDPlus-def Abs-pd-basis-inverse* [*unfolded pd-basis-def*])
done

lemma *pd-basis-induct*:
assumes *PDUnit*: ⋀*a*. *P* (*PDUnit a*)
assumes *PDPlus*: ⋀*t u*. ⟦*P t*; *P u*⟧ ⟹ *P* (*PDPlus t u*)
shows *P x*
apply (*induct x rule*: *pd-basis-induct1*)
apply (*rule PDUnit*, *erule PDPlus* [*OF PDUnit*])
done

fold-pd

definition
fold-pd ::
(*'a compact-basis* ⇒ *'b*::*type*) ⇒ (*'b* ⇒ *'b* ⇒ *'b*) ⇒ *'a pd-basis* ⇒ *'b*
where *fold-pd g f t* = *fold1 f* (*g* ‘ *Rep-pd-basis t*)

lemma *fold-pd-PDUnit*:
includes *ab-semigroup-idem-mult f*
shows *fold-pd g f (PDUnit x) = g x*
unfolding *fold-pd-def Rep-PDUnit* **by** *simp*

lemma *fold-pd-PDPlus*:
includes *ab-semigroup-idem-mult f*
shows *fold-pd g f (PDPlus t u) = f (fold-pd g f t) (fold-pd g f u)*
unfolding *fold-pd-def Rep-PDPlus* **by** (*simp add*: *image-Un fold1-Un2*)

approx-pd

definition
approx-pd :: *nat* ⇒ *'a pd-basis* ⇒ *'a pd-basis* **where**
approx-pd n = (λ*t*. *Abs-pd-basis* (*compact-approx n ‘ Rep-pd-basis t*))

lemma *Rep-approx-pd*:
Rep-pd-basis (*approx-pd n t*) = *compact-approx n ‘ Rep-pd-basis t*
unfolding *approx-pd-def*
apply (*rule Abs-pd-basis-inverse*)
apply (*simp add*: *pd-basis-def*)
done

lemma *approx-pd-simps* [*simp*]:
approx-pd n (*PDUnit a*) = *PDUnit* (*compact-approx n a*)
approx-pd n (*PDPlus t u*) = *PDPlus* (*approx-pd n t*) (*approx-pd n u*)
apply (*simp-all add*: *Rep-pd-basis-inject* [*symmetric*])
apply (*simp-all add*: *Rep-approx-pd Rep-PDUnit Rep-PDPlus image-Un*)
done

lemma *approx-pd-idem*: *approx-pd n* (*approx-pd n t*) = *approx-pd n t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-approx-idem*)
apply *simp*
done

lemma *range-image-f*: *range* (*image f*) = *Pow* (*range f*)
apply (*safe*, *fast*)
apply (*rule-tac x*=*f* −‘ *x* **in** *range-eqI*)
apply (*simp*, *fast*)
done

lemma *finite-range-approx-pd*: *finite* (*range* (*approx-pd n*))
apply (*subgoal-tac finite* (*Rep-pd-basis ‘ range* (*approx-pd n*)))
apply (*erule finite-imageD*)
apply (*rule inj-onI*, *simp add*: *Rep-pd-basis-inject*)
apply (*subst image-image*)
apply (*subst Rep-approx-pd*)
apply (*simp only*: *range-composition*)
apply (*rule finite-subset* [*OF image-mono* [*OF subset-UNIV*]])

apply (*simp add*: *range-image-f*)
apply (*rule finite-range-compact-approx*)
done

lemma *ex-approx-pd-eq*: $\exists\, n.\ \mathit{approx\text{-}pd}\ n\ t = t$
apply (*subgoal-tac* $\exists\, n.\ \forall\, m{\geq}n.\ \mathit{approx\text{-}pd}\ m\ t = t$, *fast*)
apply (*induct t rule*: *pd-basis-induct*)
apply (*cut-tac a*=*a* **in** *ex-compact-approx-eq*)
apply (*clarify*, *rule-tac x*=*n* **in** *exI*)
apply (*clarify*, *simp*)
apply (*rule antisym-less*)
apply (*rule compact-approx-le*)
apply (*drule-tac a*=*a* **in** *compact-approx-mono1*)
apply *simp*
apply (*clarify*, *rename-tac i j*)
apply (*rule-tac x*=*max i j* **in** *exI*, *simp*)
done

end

23 UpperPD: Upper powerdomain

theory *UpperPD*
imports *CompactBasis*
begin

23.1 Basis preorder

definition
upper-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** $\leq\sharp$ *50*) **where**
upper-le = $(\lambda u\ v.\ \forall\, y{\in}\mathit{Rep\text{-}pd\text{-}basis}\ v.\ \exists\, x{\in}\mathit{Rep\text{-}pd\text{-}basis}\ u.\ x \sqsubseteq y)$

lemma *upper-le-refl* [*simp*]: $t \leq\sharp\ t$
unfolding *upper-le-def* **by** *fast*

lemma *upper-le-trans*: $[\![t \leq\sharp\ u;\ u \leq\sharp\ v]\!] \Longrightarrow t \leq\sharp\ v$
unfolding *upper-le-def*
apply (*rule ballI*)
apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*erule rev-bexI*)
apply (*erule* (*1*) *trans-less*)
done

interpretation *upper-le*: *preorder* [*upper-le*]
by (*rule preorder.intro*, *rule upper-le-refl*, *rule upper-le-trans*)

lemma *upper-le-minimal* [*simp*]: *PDUnit compact-bot* $\leq\sharp\ t$

unfolding *upper-le-def Rep-PDUnit* **by** *simp*

lemma *PDUnit-upper-mono*: $x \sqsubseteq y \Longrightarrow PDUnit\ x \leq\sharp PDUnit\ y$
unfolding *upper-le-def Rep-PDUnit* **by** *simp*

lemma *PDPlus-upper-mono*: $[\![s \leq\sharp t;\ u \leq\sharp v]\!] \Longrightarrow PDPlus\ s\ u \leq\sharp PDPlus\ t\ v$
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *PDPlus-upper-less*: $PDPlus\ t\ u \leq\sharp t$
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *upper-le-PDUnit-PDUnit-iff* [*simp*]:
$(PDUnit\ a \leq\sharp PDUnit\ b) = a \sqsubseteq b$
unfolding *upper-le-def Rep-PDUnit* **by** *fast*

lemma *upper-le-PDPlus-PDUnit-iff*:
$(PDPlus\ t\ u \leq\sharp PDUnit\ a) = (t \leq\sharp PDUnit\ a \vee u \leq\sharp PDUnit\ a)$
unfolding *upper-le-def Rep-PDPlus Rep-PDUnit* **by** *fast*

lemma *upper-le-PDPlus-iff*: $(t \leq\sharp PDPlus\ u\ v) = (t \leq\sharp u \wedge t \leq\sharp v)$
unfolding *upper-le-def Rep-PDPlus* **by** *fast*

lemma *upper-le-induct* [*induct set*: *upper-le*]:
assumes *le*: $t \leq\sharp u$
assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P\ (PDUnit\ a)\ (PDUnit\ b)$
assumes *2*: $\bigwedge t\ u\ a.\ P\ t\ (PDUnit\ a) \Longrightarrow P\ (PDPlus\ t\ u)\ (PDUnit\ a)$
assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ u;\ P\ t\ v]\!] \Longrightarrow P\ t\ (PDPlus\ u\ v)$
shows $P\ t\ u$
using *le* **apply** (*induct u arbitrary*: *t rule*: *pd-basis-induct*)
apply (*erule rev-mp*)
apply (*induct-tac t rule*: *pd-basis-induct*)
apply (*simp add*: *1*)
apply (*simp add*: *upper-le-PDPlus-PDUnit-iff*)
apply (*simp add*: *2*)
apply (*subst PDPlus-commute*)
apply (*simp add*: *2*)
apply (*simp add*: *upper-le-PDPlus-iff 3*)
done

lemma *approx-pd-upper-mono1*:
$i \leq j \Longrightarrow approx\text{-}pd\ i\ t \leq\sharp approx\text{-}pd\ j\ t$
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-approx-mono1*)
apply (*simp add*: *PDPlus-upper-mono*)
done

lemma *approx-pd-upper-le*: $approx\text{-}pd\ i\ t \leq\sharp t$
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-approx-le*)

apply (*simp add*: *PDPlus-upper-mono*)
done

lemma *approx-pd-upper-mono*:
t $\leq\sharp$ *u* $\Longrightarrow$ *approx-pd n t* $\leq\sharp$ *approx-pd n u*
apply (*erule upper-le-induct*)
apply (*simp add*: *compact-approx-mono*)
apply (*simp add*: *upper-le-PDPlus-PDUnit-iff*)
apply (*simp add*: *upper-le-PDPlus-iff*)
done

23.2 Type definition

cpodef (**open**) *'a upper-pd* =
{*S*::*'a*::*profinite pd-basis set*. *upper-le.ideal S*}
apply (*simp add*: *upper-le.adm-ideal*)
apply (*fast intro*: *upper-le.ideal-principal*)
done

lemma *ideal-Rep-upper-pd*: *upper-le.ideal* (*Rep-upper-pd x*)
by (*rule Rep-upper-pd* [*unfolded mem-Collect-eq*])

definition
upper-principal :: *'a pd-basis* $\Rightarrow$ *'a upper-pd* **where**
upper-principal t = *Abs-upper-pd* {*u*. *u* $\leq\sharp$ *t*}

lemma *Rep-upper-principal*:
Rep-upper-pd (*upper-principal t*) = {*u*. *u* $\leq\sharp$ *t*}
unfolding *upper-principal-def*
apply (*rule Abs-upper-pd-inverse* [*unfolded mem-Collect-eq*])
apply (*rule upper-le.ideal-principal*)
done

interpretation *upper-pd*:
ideal-completion [*upper-le approx-pd upper-principal Rep-upper-pd*]
apply *unfold-locales*
apply (*rule approx-pd-upper-le*)
apply (*rule approx-pd-idem*)
apply (*erule approx-pd-upper-mono*)
apply (*rule approx-pd-upper-mono1*, *simp*)
apply (*rule finite-range-approx-pd*)
apply (*rule ex-approx-pd-eq*)
apply (*rule ideal-Rep-upper-pd*)
apply (*rule cont-Rep-upper-pd*)
apply (*rule Rep-upper-principal*)
apply (*simp only*: *less-upper-pd-def less-set-eq*)
done

lemma *upper-principal-less-iff* [*simp*]:

upper-principal t $\sqsubseteq$ *upper-principal u* $\longleftrightarrow$ *t* $\leq\sharp$ *u*
by (*rule upper-pd.principal-less-iff*)

lemma *upper-principal-eq-iff*:
upper-principal t = *upper-principal u* $\longleftrightarrow$ *t* $\leq\sharp$ *u* $\wedge$ *u* $\leq\sharp$ *t*
by (*rule upper-pd.principal-eq-iff*)

lemma *upper-principal-mono*:
t $\leq\sharp$ *u* $\Longrightarrow$ *upper-principal t* $\sqsubseteq$ *upper-principal u*
by (*rule upper-pd.principal-mono*)

lemma *compact-upper-principal*: *compact* (*upper-principal t*)
by (*rule upper-pd.compact-principal*)

lemma *upper-pd-minimal*: *upper-principal* (*PDUnit compact-bot*) $\sqsubseteq$ *ys*
by (*induct ys rule*: *upper-pd.principal-induct*, *simp*, *simp*)

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

lemma *inst-upper-pd-pcpo*: $\bot$ = *upper-principal* (*PDUnit compact-bot*)
by (*rule upper-pd-minimal* [*THEN UU-I*, *symmetric*])

23.3 Approximation

instantiation *upper-pd* :: (*profinite*) *profinite*
begin

definition
approx-upper-pd-def: *approx* = *upper-pd.completion-approx*

instance
apply (*intro-classes*, *unfold approx-upper-pd-def*)
apply (*simp add*: *upper-pd.chain-completion-approx*)
apply (*rule upper-pd.lub-completion-approx*)
apply (*rule upper-pd.completion-approx-idem*)
apply (*rule upper-pd.finite-fixes-completion-approx*)
done

end

instance *upper-pd* :: (*bifinite*) *bifinite* **..**

lemma *approx-upper-principal* [*simp*]:
approx n·(*upper-principal t*) = *upper-principal* (*approx-pd n t*)
unfolding *approx-upper-pd-def*
by (*rule upper-pd.completion-approx-principal*)

lemma *approx-eq-upper-principal*:

$\exists t{\in}$*Rep-upper-pd xs*. *approx n·xs* = *upper-principal* (*approx-pd n t*)
unfolding *approx-upper-pd-def*
by (*rule upper-pd.completion-approx-eq-principal*)

lemma *compact-imp-upper-principal*:
compact xs $\Longrightarrow$ $\exists$ *t*. *xs* = *upper-principal t*
apply (*drule bifinite-compact-eq-approx*)
apply (*erule exE*)
apply (*erule subst*)
apply (*cut-tac n=i* **and** *xs=xs* **in** *approx-eq-upper-principal*)
apply *fast*
done

lemma *upper-principal-induct*:
⟦*adm P*; $\bigwedge$*t*. *P* (*upper-principal t*)⟧ $\Longrightarrow$ *P xs*
by (*rule upper-pd.principal-induct*)

lemma *upper-principal-induct2*:
⟦$\bigwedge$*ys*. *adm* (λ*xs*. *P xs ys*); $\bigwedge$*xs*. *adm* (λ*ys*. *P xs ys*);
$\bigwedge$*t u*. *P* (*upper-principal t*) (*upper-principal u*)⟧ $\Longrightarrow$ *P xs ys*
apply (*rule-tac x=ys* **in** *spec*)
apply (*rule-tac xs=xs* **in** *upper-principal-induct*, *simp*)
apply (*rule allI*, *rename-tac ys*)
apply (*rule-tac xs=ys* **in** *upper-principal-induct*, *simp*)
apply *simp*
done

23.4 Monadic unit and plus

definition
upper-unit :: *'a* → *'a upper-pd* **where**
upper-unit = *compact-basis.basis-fun* (λ*a*. *upper-principal* (*PDUnit a*))

definition
upper-plus :: *'a upper-pd* → *'a upper-pd* → *'a upper-pd* **where**
upper-plus = *upper-pd.basis-fun* (λ*t*. *upper-pd.basis-fun* (λ*u*.
upper-principal (*PDPlus t u*)))

abbreviation
upper-add :: *'a upper-pd* ⇒ *'a upper-pd* ⇒ *'a upper-pd*
(**infixl** +♯ *65*) **where**
xs +♯ *ys* == *upper-plus·xs·ys*

syntax
-upper-pd :: *args* ⇒ *'a upper-pd* ({-}♯)

translations
{*x*,*xs*}♯ == {*x*}♯ +♯ {*xs*}♯
{*x*}♯ == *CONST upper-unit·x*

lemma *upper-unit-Rep-compact-basis* [*simp*]:
  {*Rep-compact-basis a*}♯ = *upper-principal* (*PDUnit a*)
unfolding *upper-unit-def*
by (*simp add*: *compact-basis.basis-fun-principal*
    upper-principal-mono PDUnit-upper-mono)

lemma *upper-plus-principal* [*simp*]:
  upper-principal t +♯ *upper-principal u* = *upper-principal* (*PDPlus t u*)
unfolding *upper-plus-def*
by (*simp add*: *upper-pd.basis-fun-principal*
    upper-pd.basis-fun-mono PDPlus-upper-mono)

lemma *approx-upper-unit* [*simp*]:
  approx n·{*x*}♯ = {*approx n*·*x*}♯
apply (*induct x rule*: *compact-basis-induct*, *simp*)
apply (*simp add*: *approx-Rep-compact-basis*)
done

lemma *approx-upper-plus* [*simp*]:
  approx n·(*xs* +♯ *ys*) = (*approx n*·*xs*) +♯ (*approx n*·*ys*)
by (*induct xs ys rule*: *upper-principal-induct2*, *simp*, *simp*, *simp*)

lemma *upper-plus-assoc*: (*xs* +♯ *ys*) +♯ *zs* = *xs* +♯ (*ys* +♯ *zs*)
apply (*induct xs ys arbitrary*: *zs rule*: *upper-principal-induct2*, *simp*, *simp*)
apply (*rule-tac xs*=*zs* **in** *upper-principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done

lemma *upper-plus-commute*: *xs* +♯ *ys* = *ys* +♯ *xs*
apply (*induct xs ys rule*: *upper-principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-commute*)
done

lemma *upper-plus-absorb*: *xs* +♯ *xs* = *xs*
apply (*induct xs rule*: *upper-principal-induct*, *simp*)
apply (*simp add*: *PDPlus-absorb*)
done

interpretation *aci-upper-plus*: *ab-semigroup-idem-mult* [*op* +♯]
  by *unfold-locales*
    (*rule upper-plus-assoc upper-plus-commute upper-plus-absorb*)+

lemma *upper-plus-left-commute*: *xs* +♯ (*ys* +♯ *zs*) = *ys* +♯ (*xs* +♯ *zs*)
by (*rule aci-upper-plus.mult-left-commute*)

lemma *upper-plus-left-absorb*: *xs* +♯ (*xs* +♯ *ys*) = *xs* +♯ *ys*
by (*rule aci-upper-plus.mult-left-idem*)

lemmas *upper-plus-aci = aci-upper-plus.mult-ac-idem*

lemma *upper-plus-less1*: $xs +\sharp ys \sqsubseteq xs$
apply (*induct xs ys rule*: *upper-principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-upper-less*)
done

lemma *upper-plus-less2*: $xs +\sharp ys \sqsubseteq ys$
by (*subst upper-plus-commute*, *rule upper-plus-less1*)

lemma *upper-plus-greatest*: $[\![xs \sqsubseteq ys; xs \sqsubseteq zs]\!] \Longrightarrow xs \sqsubseteq ys +\sharp zs$
apply (*subst upper-plus-absorb* [*of xs*, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *upper-less-plus-iff*:
$xs \sqsubseteq ys +\sharp zs \longleftrightarrow xs \sqsubseteq ys \wedge xs \sqsubseteq zs$
apply *safe*
apply (*erule trans-less* [*OF* - *upper-plus-less1*])
apply (*erule trans-less* [*OF* - *upper-plus-less2*])
apply (*erule* (*1*) *upper-plus-greatest*)
done

lemma *upper-plus-less-unit-iff*:
$xs +\sharp ys \sqsubseteq \{z\}\sharp \longleftrightarrow xs \sqsubseteq \{z\}\sharp \vee ys \sqsubseteq \{z\}\sharp$
apply (*rule iffI*)
apply (*subgoal-tac*
adm ($\lambda f.\ f{\cdot}xs \sqsubseteq f{\cdot}\{z\}\sharp \vee f{\cdot}ys \sqsubseteq f{\cdot}\{z\}\sharp$))
apply (*drule admD*, *rule chain-approx*)
apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*)
apply (*cut-tac xs=approx i·xs* **in** *compact-imp-upper-principal*, *simp*)
apply (*cut-tac xs=approx i·ys* **in** *compact-imp-upper-principal*, *simp*)
apply (*cut-tac x=approx i·z* **in** *compact-imp-Rep-compact-basis*, *simp*)
apply (*clarify*, *simp add*: *upper-le-PDPlus-PDUnit-iff*)
apply *simp*
apply *simp*
apply (*erule disjE*)
apply (*erule trans-less* [*OF upper-plus-less1*])
apply (*erule trans-less* [*OF upper-plus-less2*])
done

lemma *upper-unit-less-iff* [*simp*]: $\{x\}\sharp \sqsubseteq \{y\}\sharp \longleftrightarrow x \sqsubseteq y$
apply (*rule iffI*)
apply (*rule bifinite-less-ext*)
apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*, *simp*)
apply (*cut-tac x=approx i·x* **in** *compact-imp-Rep-compact-basis*, *simp*)
apply (*cut-tac x=approx i·y* **in** *compact-imp-Rep-compact-basis*, *simp*)
apply (*clarify*, *simp add*: *compact-le-def*)
apply (*erule monofun-cfun-arg*)

done

lemmas *upper-pd-less-simps* =
upper-unit-less-iff
upper-less-plus-iff
upper-plus-less-unit-iff

lemma *upper-unit-eq-iff* [*simp*]: $\{x\}\sharp = \{y\}\sharp \longleftrightarrow x = y$
unfolding *po-eq-conv* **by** *simp*

lemma *upper-unit-strict* [*simp*]: $\{\bot\}\sharp = \bot$
unfolding *inst-upper-pd-pcpo Rep-compact-bot* [*symmetric*] **by** *simp*

lemma *upper-plus-strict1* [*simp*]: $\bot +\sharp\ ys = \bot$
by (*rule UU-I*, *rule upper-plus-less1*)

lemma *upper-plus-strict2* [*simp*]: $xs +\sharp \bot = \bot$
by (*rule UU-I*, *rule upper-plus-less2*)

lemma *upper-unit-strict-iff* [*simp*]: $\{x\}\sharp = \bot \longleftrightarrow x = \bot$
unfolding *upper-unit-strict* [*symmetric*] **by** (*rule upper-unit-eq-iff*)

lemma *upper-plus-strict-iff* [*simp*]:
$xs +\sharp ys = \bot \longleftrightarrow xs = \bot \vee ys = \bot$
apply (*rule iffI*)
apply (*erule rev-mp*)
apply (*rule upper-principal-induct2* [**where** *xs*=*xs* **and** *ys*=*ys*], *simp*, *simp*)
apply (*simp add*: *inst-upper-pd-pcpo upper-principal-eq-iff*
upper-le-PDPlus-PDUnit-iff)
apply *auto*
done

lemma *compact-upper-unit-iff* [*simp*]: *compact* $\{x\}\sharp \longleftrightarrow$ *compact* x
unfolding *bifinite-compact-iff* **by** *simp*

lemma *compact-upper-plus* [*simp*]:
$[\![compact\ xs;\ compact\ ys]\!] \Longrightarrow compact\ (xs +\sharp ys)$
apply (*drule compact-imp-upper-principal*)+
apply (*auto simp add*: *compact-upper-principal*)
done

23.5 Induction rules

lemma *upper-pd-induct1*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\sharp$
assumes *insert*: $\bigwedge x\ ys.\ [\![P\ \{x\}\sharp;\ P\ ys]\!] \Longrightarrow P\ (\{x\}\sharp +\sharp ys)$
shows *P* (*xs*::'*a upper-pd*)
apply (*induct xs rule*: *upper-principal-induct*, *rule P*)

apply (*induct-tac t rule*: *pd-basis-induct1*)
apply (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*])
apply (*rule unit*)
apply (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*]
upper-plus-principal [*symmetric*])
apply (*erule insert* [*OF unit*])
done

lemma *upper-pd-induct*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\sharp$
assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs +\sharp ys)$
shows *P* (*xs*::'*a upper-pd*)
apply (*induct xs rule*: *upper-principal-induct*, *rule P*)
apply (*induct-tac t rule*: *pd-basis-induct*)
apply (*simp only*: *upper-unit-Rep-compact-basis* [*symmetric*] *unit*)
apply (*simp only*: *upper-plus-principal* [*symmetric*] *plus*)
done

23.6 Monadic bind

definition
upper-bind-basis ::
'*a pd-basis* ⇒ ('*a* → '*b upper-pd*) → '*b upper-pd* **where**
upper-bind-basis = *fold-pd*
($\lambda a.\ \Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a)$)
($\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\sharp y{\cdot}f$)

lemma *ACI-upper-bind*:
ab-semigroup-idem-mult ($\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\sharp y{\cdot}f$)
apply *unfold-locales*
apply (*simp add*: *upper-plus-assoc*)
apply (*simp add*: *upper-plus-commute*)
apply (*simp add*: *upper-plus-absorb eta-cfun*)
done

lemma *upper-bind-basis-simps* [*simp*]:
upper-bind-basis (*PDUnit a*) =
($\Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a)$)
upper-bind-basis (*PDPlus t u*) =
($\Lambda\ f.$ *upper-bind-basis* $t{\cdot}f +\sharp$ *upper-bind-basis* $u{\cdot}f$)
unfolding *upper-bind-basis-def*
apply −
apply (*rule fold-pd-PDUnit* [*OF ACI-upper-bind*])
apply (*rule fold-pd-PDPlus* [*OF ACI-upper-bind*])
done

lemma *upper-bind-basis-mono*:
$t \leq\sharp u \Longrightarrow$ *upper-bind-basis* $t \sqsubseteq$ *upper-bind-basis u*

unfolding *expand-cfun-less*
apply (*erule upper-le-induct*, *safe*)
apply (*simp add*: *compact-le-def monofun-cfun*)
apply (*simp add*: *trans-less* [*OF upper-plus-less1*])
apply (*simp add*: *upper-less-plus-iff*)
done

definition
upper-bind :: *'a upper-pd* → (*'a* → *'b upper-pd*) → *'b upper-pd* **where**
upper-bind = *upper-pd.basis-fun upper-bind-basis*

lemma *upper-bind-principal* [*simp*]:
upper-bind·(*upper-principal t*) = *upper-bind-basis t*
unfolding *upper-bind-def*
apply (*rule upper-pd.basis-fun-principal*)
apply (*erule upper-bind-basis-mono*)
done

lemma *upper-bind-unit* [*simp*]:
upper-bind·{*x*}♯·*f* = *f*·*x*
by (*induct x rule*: *compact-basis-induct*, *simp*, *simp*)

lemma *upper-bind-plus* [*simp*]:
upper-bind·(*xs* +♯ *ys*)·*f* = *upper-bind*·*xs*·*f* +♯ *upper-bind*·*ys*·*f*
by (*induct xs ys rule*: *upper-principal-induct2*, *simp*, *simp*, *simp*)

lemma *upper-bind-strict* [*simp*]: *upper-bind*·⊥·*f* = *f*·⊥
unfolding *upper-unit-strict* [*symmetric*] **by** (*rule upper-bind-unit*)

23.7 Map and join

definition
upper-map :: (*'a* → *'b*) → *'a upper-pd* → *'b upper-pd* **where**
upper-map = (Λ *f xs*. *upper-bind*·*xs*·(Λ *x*. {*f*·*x*}♯))

definition
upper-join :: *'a upper-pd upper-pd* → *'a upper-pd* **where**
upper-join = (Λ *xss*. *upper-bind*·*xss*·(Λ *xs*. *xs*))

lemma *upper-map-unit* [*simp*]:
upper-map·*f*·{*x*}♯ = {*f*·*x*}♯
unfolding *upper-map-def* **by** *simp*

lemma *upper-map-plus* [*simp*]:
upper-map·*f*·(*xs* +♯ *ys*) = *upper-map*·*f*·*xs* +♯ *upper-map*·*f*·*ys*
unfolding *upper-map-def* **by** *simp*

lemma *upper-join-unit* [*simp*]:
upper-join·{*xs*}♯ = *xs*

unfolding *upper-join-def* **by** *simp*

lemma *upper-join-plus* [*simp*]:
upper-join·(*xss* +♯ *yss*) = *upper-join*·*xss* +♯ *upper-join*·*yss*
unfolding *upper-join-def* **by** *simp*

lemma *upper-map-ident*: *upper-map*·(Λ *x*. *x*)·*xs* = *xs*
by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-map-map*:
upper-map·*f*·(*upper-map*·*g*·*xs*) = *upper-map*·(Λ *x*. *f*·(*g*·*x*))·*xs*
by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-join-map-unit*:
upper-join·(*upper-map*·*upper-unit*·*xs*) = *xs*
by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-join-map-join*:
upper-join·(*upper-map*·*upper-join*·*xsss*) = *upper-join*·(*upper-join*·*xsss*)
by (*induct xsss rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-join-map-map*:
upper-join·(*upper-map*·(*upper-map*·*f*)·*xss*) =
upper-map·*f*·(*upper-join*·*xss*)
by (*induct xss rule*: *upper-pd-induct*, *simp-all*)

lemma *upper-map-approx*: *upper-map*·(*approx n*)·*xs* = *approx n*·*xs*
by (*induct xs rule*: *upper-pd-induct*, *simp-all*)

end

24 LowerPD: Lower powerdomain

theory *LowerPD*
imports *CompactBasis*
begin

24.1 Basis preorder

definition
lower-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** ≤♭ *50*) **where**
lower-le = (λ*u v*. ∀ *x*∈*Rep-pd-basis u*. ∃ *y*∈*Rep-pd-basis v*. *x* ⊑ *y*)

lemma *lower-le-refl* [*simp*]: *t* ≤♭ *t*
unfolding *lower-le-def* **by** *fast*

lemma *lower-le-trans*: ⟦*t* ≤♭ *u*; *u* ≤♭ *v*⟧ ⟹ *t* ≤♭ *v*
unfolding *lower-le-def*

apply (*rule ballI*)
apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*erule rev-bexI*)
apply (*erule* (*1*) *trans-less*)
done

interpretation *lower-le*: *preorder* [*lower-le*]
by (*rule preorder.intro*, *rule lower-le-refl*, *rule lower-le-trans*)

lemma *lower-le-minimal* [*simp*]: *PDUnit compact-bot* $\leq\flat$ *t*
unfolding *lower-le-def Rep-PDUnit*
by (*simp*, *rule Rep-pd-basis-nonempty* [*folded ex-in-conv*])

lemma *PDUnit-lower-mono*: $x \sqsubseteq y \Longrightarrow$ *PDUnit x* $\leq\flat$ *PDUnit y*
unfolding *lower-le-def Rep-PDUnit* **by** *fast*

lemma *PDPlus-lower-mono*: $[\![s \leq\flat t;\ u \leq\flat v]\!] \Longrightarrow$ *PDPlus s u* $\leq\flat$ *PDPlus t v*
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *PDPlus-lower-less*: *t* $\leq\flat$ *PDPlus t u*
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *lower-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a* $\leq\flat$ *PDUnit b*) = *a* $\sqsubseteq$ *b*
unfolding *lower-le-def Rep-PDUnit* **by** *fast*

lemma *lower-le-PDUnit-PDPlus-iff*:
(*PDUnit a* $\leq\flat$ *PDPlus t u*) = (*PDUnit a* $\leq\flat$ *t* $\vee$ *PDUnit a* $\leq\flat$ *u*)
unfolding *lower-le-def Rep-PDPlus Rep-PDUnit* **by** *fast*

lemma *lower-le-PDPlus-iff*: (*PDPlus t u* $\leq\flat$ *v*) = (*t* $\leq\flat$ *v* $\wedge$ *u* $\leq\flat$ *v*)
unfolding *lower-le-def Rep-PDPlus* **by** *fast*

lemma *lower-le-induct* [*induct set*: *lower-le*]:
assumes *le*: *t* $\leq\flat$ *u*
assumes *1*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P$ (*PDUnit a*) (*PDUnit b*)
assumes *2*: $\bigwedge t\ u\ a.\ P$ (*PDUnit a*) *t* $\Longrightarrow P$ (*PDUnit a*) (*PDPlus t u*)
assumes *3*: $\bigwedge t\ u\ v.\ [\![P\ t\ v;\ P\ u\ v]\!] \Longrightarrow P$ (*PDPlus t u*) *v*
shows *P t u*
using *le*
apply (*induct t arbitrary*: *u rule*: *pd-basis-induct*)
apply (*erule rev-mp*)
apply (*induct-tac u rule*: *pd-basis-induct*)
apply (*simp add*: *1*)
apply (*simp add*: *lower-le-PDUnit-PDPlus-iff*)
apply (*simp add*: *2*)
apply (*subst PDPlus-commute*)
apply (*simp add*: *2*)

apply (*simp add*: *lower-le-PDPlus-iff 3*)
done

lemma *approx-pd-lower-mono1*:
i ≤ *j* ⟹ *approx-pd i t* ≤♭ *approx-pd j t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-approx-mono1*)
apply (*simp add*: *PDPlus-lower-mono*)
done

lemma *approx-pd-lower-le*: *approx-pd i t* ≤♭ *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-approx-le*)
apply (*simp add*: *PDPlus-lower-mono*)
done

lemma *approx-pd-lower-mono*:
t ≤♭ *u* ⟹ *approx-pd n t* ≤♭ *approx-pd n u*
apply (*erule lower-le-induct*)
apply (*simp add*: *compact-approx-mono*)
apply (*simp add*: *lower-le-PDUnit-PDPlus-iff*)
apply (*simp add*: *lower-le-PDPlus-iff*)
done

24.2 Type definition

cpodef (**open**) *'a lower-pd* =
{*S*::*'a*::*profinite pd-basis set*. *lower-le.ideal S*}
apply (*simp add*: *lower-le.adm-ideal*)
apply (*fast intro*: *lower-le.ideal-principal*)
done

lemma *ideal-Rep-lower-pd*: *lower-le.ideal* (*Rep-lower-pd x*)
by (*rule Rep-lower-pd* [*unfolded mem-Collect-eq*])

definition
lower-principal :: *'a pd-basis* ⇒ *'a lower-pd* **where**
lower-principal t = *Abs-lower-pd* {*u*. *u* ≤♭ *t*}

lemma *Rep-lower-principal*:
Rep-lower-pd (*lower-principal t*) = {*u*. *u* ≤♭ *t*}
unfolding *lower-principal-def*
apply (*rule Abs-lower-pd-inverse* [*simplified*])
apply (*rule lower-le.ideal-principal*)
done

interpretation *lower-pd*:
ideal-completion [*lower-le approx-pd lower-principal Rep-lower-pd*]
apply *unfold-locales*

apply (*rule approx-pd-lower-le*)
apply (*rule approx-pd-idem*)
apply (*erule approx-pd-lower-mono*)
apply (*rule approx-pd-lower-mono1*, *simp*)
apply (*rule finite-range-approx-pd*)
apply (*rule ex-approx-pd-eq*)
apply (*rule ideal-Rep-lower-pd*)
apply (*rule cont-Rep-lower-pd*)
apply (*rule Rep-lower-principal*)
apply (*simp only*: *less-lower-pd-def less-set-eq*)
done

lemma *lower-principal-less-iff* [*simp*]:
lower-principal t $\sqsubseteq$ *lower-principal u* $\longleftrightarrow$ *t* $\leq\flat$ *u*
by (*rule lower-pd.principal-less-iff*)

lemma *lower-principal-eq-iff*:
lower-principal t = *lower-principal u* $\longleftrightarrow$ *t* $\leq\flat$ *u* $\wedge$ *u* $\leq\flat$ *t*
by (*rule lower-pd.principal-eq-iff*)

lemma *lower-principal-mono*:
t $\leq\flat$ *u* $\Longrightarrow$ *lower-principal t* $\sqsubseteq$ *lower-principal u*
by (*rule lower-pd.principal-mono*)

lemma *compact-lower-principal*: *compact* (*lower-principal t*)
by (*rule lower-pd.compact-principal*)

lemma *lower-pd-minimal*: *lower-principal* (*PDUnit compact-bot*) $\sqsubseteq$ *ys*
by (*induct ys rule*: *lower-pd.principal-induct*, *simp*, *simp*)

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

lemma *inst-lower-pd-pcpo*: $\bot$ = *lower-principal* (*PDUnit compact-bot*)
by (*rule lower-pd-minimal* [*THEN UU-I*, *symmetric*])

24.3 Approximation

instantiation *lower-pd* :: (*profinite*) *profinite*
begin

definition
approx-lower-pd-def: *approx* = *lower-pd.completion-approx*

instance
apply (*intro-classes*, *unfold approx-lower-pd-def*)
apply (*simp add*: *lower-pd.chain-completion-approx*)
apply (*rule lower-pd.lub-completion-approx*)
apply (*rule lower-pd.completion-approx-idem*)

apply (*rule lower-pd.finite-fixes-completion-approx*)
done

end

instance *lower-pd* :: (*bifinite*) *bifinite* **..**

lemma *approx-lower-principal* [*simp*]:
approx n·(*lower-principal t*) = *lower-principal* (*approx-pd n t*)
unfolding *approx-lower-pd-def*
by (*rule lower-pd.completion-approx-principal*)

lemma *approx-eq-lower-principal*:
∃ *t*∈*Rep-lower-pd xs*. *approx n·xs* = *lower-principal* (*approx-pd n t*)
unfolding *approx-lower-pd-def*
by (*rule lower-pd.completion-approx-eq-principal*)

lemma *compact-imp-lower-principal*:
compact xs ⟹ ∃ *t*. *xs* = *lower-principal t*
apply (*drule bifinite-compact-eq-approx*)
apply (*erule exE*)
apply (*erule subst*)
apply (*cut-tac n*=*i* **and** *xs*=*xs* **in** *approx-eq-lower-principal*)
apply *fast*
done

lemma *lower-principal-induct*:
⟦*adm P*; ⋀*t*. *P* (*lower-principal t*)⟧ ⟹ *P xs*
by (*rule lower-pd.principal-induct*)

lemma *lower-principal-induct2*:
⟦⋀*ys*. *adm* (λ*xs*. *P xs ys*); ⋀*xs*. *adm* (λ*ys*. *P xs ys*);
⋀*t u*. *P* (*lower-principal t*) (*lower-principal u*)⟧ ⟹ *P xs ys*
apply (*rule-tac x*=*ys* **in** *spec*)
apply (*rule-tac xs*=*xs* **in** *lower-principal-induct*, *simp*)
apply (*rule allI*, *rename-tac ys*)
apply (*rule-tac xs*=*ys* **in** *lower-principal-induct*, *simp*)
apply *simp*
done

24.4 Monadic unit and plus

definition
lower-unit :: *'a* → *'a lower-pd* **where**
lower-unit = *compact-basis.basis-fun* (λ*a*. *lower-principal* (*PDUnit a*))

definition
lower-plus :: *'a lower-pd* → *'a lower-pd* → *'a lower-pd* **where**
lower-plus = *lower-pd.basis-fun* (λ*t*. *lower-pd.basis-fun* (λ*u*.

lower-principal (*PDPlus t u*)))

abbreviation
lower-add :: *'a lower-pd* ⇒ *'a lower-pd* ⇒ *'a lower-pd*
(**infixl** +♭ *65*) **where**
xs +♭ *ys* == *lower-plus*·*xs*·*ys*

syntax
-lower-pd :: *args* ⇒ *'a lower-pd* ({-}♭)

translations
{*x*,*xs*}♭ == {*x*}♭ +♭ {*xs*}♭
{*x*}♭ == *CONST lower-unit*·*x*

lemma *lower-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♭ = *lower-principal* (*PDUnit a*)
unfolding *lower-unit-def*
by (*simp add*: *compact-basis.basis-fun-principal*
lower-principal-mono PDUnit-lower-mono)

lemma *lower-plus-principal* [*simp*]:
lower-principal t +♭ *lower-principal u* = *lower-principal* (*PDPlus t u*)
unfolding *lower-plus-def*
by (*simp add*: *lower-pd.basis-fun-principal*
lower-pd.basis-fun-mono PDPlus-lower-mono)

lemma *approx-lower-unit* [*simp*]:
approx n·{*x*}♭ = {*approx n*·*x*}♭
apply (*induct x rule*: *compact-basis-induct*, *simp*)
apply (*simp add*: *approx-Rep-compact-basis*)
done

lemma *approx-lower-plus* [*simp*]:
approx n·(*xs* +♭ *ys*) = (*approx n*·*xs*) +♭ (*approx n*·*ys*)
by (*induct xs ys rule*: *lower-principal-induct2*, *simp*, *simp*, *simp*)

lemma *lower-plus-assoc*: (*xs* +♭ *ys*) +♭ *zs* = *xs* +♭ (*ys* +♭ *zs*)
apply (*induct xs ys arbitrary*: *zs rule*: *lower-principal-induct2*, *simp*, *simp*)
apply (*rule-tac xs*=*zs* **in** *lower-principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done

lemma *lower-plus-commute*: *xs* +♭ *ys* = *ys* +♭ *xs*
apply (*induct xs ys rule*: *lower-principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-commute*)
done

lemma *lower-plus-absorb*: *xs* +♭ *xs* = *xs*
apply (*induct xs rule*: *lower-principal-induct*, *simp*)

apply (*simp add*: *PDPlus-absorb*)
done

interpretation *aci-lower-plus*: *ab-semigroup-idem-mult* [*op* +♭]
by *unfold-locales*
(*rule lower-plus-assoc lower-plus-commute lower-plus-absorb*)+

lemma *lower-plus-left-commute*: *xs* +♭ (*ys* +♭ *zs*) = *ys* +♭ (*xs* +♭ *zs*)
by (*rule aci-lower-plus.mult-left-commute*)

lemma *lower-plus-left-absorb*: *xs* +♭ (*xs* +♭ *ys*) = *xs* +♭ *ys*
by (*rule aci-lower-plus.mult-left-idem*)

lemmas *lower-plus-aci* = *aci-lower-plus.mult-ac-idem*

lemma *lower-plus-less1*: *xs* ⊑ *xs* +♭ *ys*
apply (*induct xs ys rule*: *lower-principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-lower-less*)
done

lemma *lower-plus-less2*: *ys* ⊑ *xs* +♭ *ys*
by (*subst lower-plus-commute*, *rule lower-plus-less1*)

lemma *lower-plus-least*: ⟦*xs* ⊑ *zs*; *ys* ⊑ *zs*⟧ ⟹ *xs* +♭ *ys* ⊑ *zs*
apply (*subst lower-plus-absorb* [*of zs*, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *lower-plus-less-iff*:
xs +♭ *ys* ⊑ *zs* ⟷ *xs* ⊑ *zs* ∧ *ys* ⊑ *zs*
apply *safe*
apply (*erule trans-less* [*OF lower-plus-less1*])
apply (*erule trans-less* [*OF lower-plus-less2*])
apply (*erule* (*1*) *lower-plus-least*)
done

lemma *lower-unit-less-plus-iff*:
{*x*}♭ ⊑ *ys* +♭ *zs* ⟷ {*x*}♭ ⊑ *ys* ∨ {*x*}♭ ⊑ *zs*
apply (*rule iffI*)
apply (*subgoal-tac*
adm (λ*f*. *f*·{*x*}♭ ⊑ *f*·*ys* ∨ *f*·{*x*}♭ ⊑ *f*·*zs*))
apply (*drule admD*, *rule chain-approx*)
apply (*drule-tac f*=*approx i* **in** *monofun-cfun-arg*)
apply (*cut-tac x*=*approx i*·*x* **in** *compact-imp-Rep-compact-basis*, *simp*)
apply (*cut-tac xs*=*approx i*·*ys* **in** *compact-imp-lower-principal*, *simp*)
apply (*cut-tac xs*=*approx i*·*zs* **in** *compact-imp-lower-principal*, *simp*)
apply (*clarify*, *simp add*: *lower-le-PDUnit-PDPlus-iff*)
apply *simp*
apply *simp*

apply (*erule disjE*)
apply (*erule trans-less* [*OF - lower-plus-less1*])
apply (*erule trans-less* [*OF - lower-plus-less2*])
done

lemma *lower-unit-less-iff* [*simp*]: $\{x\}\flat \sqsubseteq \{y\}\flat \longleftrightarrow x \sqsubseteq y$
apply (*rule iffI*)
apply (*rule bifinite-less-ext*)
apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*, *simp*)
apply (*cut-tac x=approx i·x* **in** *compact-imp-Rep-compact-basis*, *simp*)
apply (*cut-tac x=approx i·y* **in** *compact-imp-Rep-compact-basis*, *simp*)
apply (*clarify*, *simp add*: *compact-le-def*)
apply (*erule monofun-cfun-arg*)
done

lemmas *lower-pd-less-simps* =
lower-unit-less-iff
lower-plus-less-iff
lower-unit-less-plus-iff

lemma *lower-unit-eq-iff* [*simp*]: $\{x\}\flat = \{y\}\flat \longleftrightarrow x = y$
unfolding *po-eq-conv* **by** *simp*

lemma *lower-unit-strict* [*simp*]: $\{\bot\}\flat = \bot$
unfolding *inst-lower-pd-pcpo Rep-compact-bot* [*symmetric*] **by** *simp*

lemma *lower-unit-strict-iff* [*simp*]: $\{x\}\flat = \bot \longleftrightarrow x = \bot$
unfolding *lower-unit-strict* [*symmetric*] **by** (*rule lower-unit-eq-iff*)

lemma *lower-plus-strict-iff* [*simp*]:
$xs +\flat ys = \bot \longleftrightarrow xs = \bot \wedge ys = \bot$
apply *safe*
apply (*rule UU-I*, *erule subst*, *rule lower-plus-less1*)
apply (*rule UU-I*, *erule subst*, *rule lower-plus-less2*)
apply (*rule lower-plus-absorb*)
done

lemma *lower-plus-strict1* [*simp*]: $\bot +\flat ys = ys$
apply (*rule antisym-less* [*OF - lower-plus-less2*])
apply (*simp add*: *lower-plus-least*)
done

lemma *lower-plus-strict2* [*simp*]: $xs +\flat \bot = xs$
apply (*rule antisym-less* [*OF - lower-plus-less1*])
apply (*simp add*: *lower-plus-least*)
done

lemma *compact-lower-unit-iff* [*simp*]: *compact* $\{x\}\flat \longleftrightarrow$ *compact x*
unfolding *bifinite-compact-iff* **by** *simp*

lemma *compact-lower-plus* [*simp*]:
  ⟦*compact xs*; *compact ys*⟧ ⟹ *compact* (*xs* +♭ *ys*)
apply (*drule compact-imp-lower-principal*)+
apply (*auto simp add*: *compact-lower-principal*)
done

24.5 Induction rules

lemma *lower-pd-induct1*:
  assumes *P*: *adm P*
  assumes *unit*: ⋀*x*. *P* {*x*}♭
  assumes *insert*:
    ⋀*x ys*. ⟦*P* {*x*}♭; *P ys*⟧ ⟹ *P* ({*x*}♭ +♭ *ys*)
  shows *P* (*xs*::'*a lower-pd*)
apply (*induct xs rule*: *lower-principal-induct*, *rule P*)
apply (*induct-tac t rule*: *pd-basis-induct1*)
apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*])
apply (*rule unit*)
apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*]
            lower-plus-principal [*symmetric*])
apply (*erule insert* [*OF unit*])
done

lemma *lower-pd-induct*:
  assumes *P*: *adm P*
  assumes *unit*: ⋀*x*. *P* {*x*}♭
  assumes *plus*: ⋀*xs ys*. ⟦*P xs*; *P ys*⟧ ⟹ *P* (*xs* +♭ *ys*)
  shows *P* (*xs*::'*a lower-pd*)
apply (*induct xs rule*: *lower-principal-induct*, *rule P*)
apply (*induct-tac t rule*: *pd-basis-induct*)
apply (*simp only*: *lower-unit-Rep-compact-basis* [*symmetric*] *unit*)
apply (*simp only*: *lower-plus-principal* [*symmetric*] *plus*)
done

24.6 Monadic bind

definition
  lower-bind-basis ::
  '*a pd-basis* ⇒ ('*a* → '*b lower-pd*) → '*b lower-pd* **where**
  lower-bind-basis = *fold-pd*
    (λ*a*. Λ *f*. *f*·(*Rep-compact-basis a*))
    (λ*x y*. Λ *f*. *x*·*f* +♭ *y*·*f*)

lemma *ACI-lower-bind*:
  ab-semigroup-idem-mult (λ*x y*. Λ *f*. *x*·*f* +♭ *y*·*f*)
apply *unfold-locales*
apply (*simp add*: *lower-plus-assoc*)
apply (*simp add*: *lower-plus-commute*)
apply (*simp add*: *lower-plus-absorb eta-cfun*)

done

lemma *lower-bind-basis-simps* [*simp*]:
lower-bind-basis (*PDUnit a*) =
(Λ *f*. *f*·(*Rep-compact-basis a*))
lower-bind-basis (*PDPlus t u*) =
(Λ *f*. *lower-bind-basis t*·*f* +♭ *lower-bind-basis u*·*f*)
unfolding *lower-bind-basis-def*
apply −
apply (*rule fold-pd-PDUnit* [*OF ACI-lower-bind*])
apply (*rule fold-pd-PDPlus* [*OF ACI-lower-bind*])
done

lemma *lower-bind-basis-mono*:
t ≤♭ *u* ⟹ *lower-bind-basis t* ⊑ *lower-bind-basis u*
unfolding *expand-cfun-less*
apply (*erule lower-le-induct*, *safe*)
apply (*simp add*: *compact-le-def monofun-cfun*)
apply (*simp add*: *rev-trans-less* [*OF lower-plus-less1*])
apply (*simp add*: *lower-plus-less-iff*)
done

definition
lower-bind :: *'a lower-pd* → (*'a* → *'b lower-pd*) → *'b lower-pd* **where**
lower-bind = *lower-pd.basis-fun lower-bind-basis*

lemma *lower-bind-principal* [*simp*]:
lower-bind·(*lower-principal t*) = *lower-bind-basis t*
unfolding *lower-bind-def*
apply (*rule lower-pd.basis-fun-principal*)
apply (*erule lower-bind-basis-mono*)
done

lemma *lower-bind-unit* [*simp*]:
lower-bind·{*x*}♭·*f* = *f*·*x*
by (*induct x rule*: *compact-basis-induct*, *simp*, *simp*)

lemma *lower-bind-plus* [*simp*]:
lower-bind·(*xs* +♭ *ys*)·*f* = *lower-bind*·*xs*·*f* +♭ *lower-bind*·*ys*·*f*
by (*induct xs ys rule*: *lower-principal-induct2*, *simp*, *simp*, *simp*)

lemma *lower-bind-strict* [*simp*]: *lower-bind*·⊥·*f* = *f*·⊥
unfolding *lower-unit-strict* [*symmetric*] **by** (*rule lower-bind-unit*)

24.7 Map and join

definition
lower-map :: (*'a* → *'b*) → *'a lower-pd* → *'b lower-pd* **where**
lower-map = (Λ *f xs*. *lower-bind*·*xs*·(Λ *x*. {*f*·*x*}♭))

definition
lower-join :: *'a lower-pd lower-pd* → *'a lower-pd* **where**
lower-join = (Λ *xss*. *lower-bind*·*xss*·(Λ *xs*. *xs*))

lemma *lower-map-unit* [*simp*]:
lower-map·*f*·{*x*}♭ = {*f*·*x*}♭
unfolding *lower-map-def* **by** *simp*

lemma *lower-map-plus* [*simp*]:
lower-map·*f*·(*xs* +♭ *ys*) = *lower-map*·*f*·*xs* +♭ *lower-map*·*f*·*ys*
unfolding *lower-map-def* **by** *simp*

lemma *lower-join-unit* [*simp*]:
lower-join·{*xs*}♭ = *xs*
unfolding *lower-join-def* **by** *simp*

lemma *lower-join-plus* [*simp*]:
lower-join·(*xss* +♭ *yss*) = *lower-join*·*xss* +♭ *lower-join*·*yss*
unfolding *lower-join-def* **by** *simp*

lemma *lower-map-ident*: *lower-map*·(Λ *x*. *x*)·*xs* = *xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-map-map*:
lower-map·*f*·(*lower-map*·*g*·*xs*) = *lower-map*·(Λ *x*. *f*·(*g*·*x*))·*xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-join-map-unit*:
lower-join·(*lower-map*·*lower-unit*·*xs*) = *xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-join-map-join*:
lower-join·(*lower-map*·*lower-join*·*xsss*) = *lower-join*·(*lower-join*·*xsss*)
by (*induct xsss rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-join-map-map*:
lower-join·(*lower-map*·(*lower-map*·*f*)·*xss*) =
lower-map·*f*·(*lower-join*·*xss*)
by (*induct xss rule*: *lower-pd-induct*, *simp-all*)

lemma *lower-map-approx*: *lower-map*·(*approx n*)·*xs* = *approx n*·*xs*
by (*induct xs rule*: *lower-pd-induct*, *simp-all*)

end

25 ConvexPD: Convex powerdomain

theory *ConvexPD*
imports *UpperPD LowerPD*
begin

25.1 Basis preorder

definition
convex-le :: *'a pd-basis* ⇒ *'a pd-basis* ⇒ *bool* (**infix** ≤♮ *50*) **where**
convex-le = (λ*u v*. *u* ≤♯ *v* ∧ *u* ≤♭ *v*)

lemma *convex-le-refl* [*simp*]: *t* ≤♮ *t*
unfolding *convex-le-def* **by** (*fast intro*: *upper-le-refl lower-le-refl*)

lemma *convex-le-trans*: ⟦*t* ≤♮ *u*; *u* ≤♮ *v*⟧ ⟹ *t* ≤♮ *v*
unfolding *convex-le-def* **by** (*fast intro*: *upper-le-trans lower-le-trans*)

interpretation *convex-le*: *preorder* [*convex-le*]
by (*rule preorder.intro*, *rule convex-le-refl*, *rule convex-le-trans*)

lemma *upper-le-minimal* [*simp*]: *PDUnit compact-bot* ≤♮ *t*
unfolding *convex-le-def Rep-PDUnit* **by** *simp*

lemma *PDUnit-convex-mono*: *x* ⊑ *y* ⟹ *PDUnit x* ≤♮ *PDUnit y*
unfolding *convex-le-def* **by** (*fast intro*: *PDUnit-upper-mono PDUnit-lower-mono*)

lemma *PDPlus-convex-mono*: ⟦*s* ≤♮ *t*; *u* ≤♮ *v*⟧ ⟹ *PDPlus s u* ≤♮ *PDPlus t v*
unfolding *convex-le-def* **by** (*fast intro*: *PDPlus-upper-mono PDPlus-lower-mono*)

lemma *convex-le-PDUnit-PDUnit-iff* [*simp*]:
(*PDUnit a* ≤♮ *PDUnit b*) = *a* ⊑ *b*
unfolding *convex-le-def upper-le-def lower-le-def Rep-PDUnit* **by** *fast*

lemma *convex-le-PDUnit-lemma1*:
(*PDUnit a* ≤♮ *t*) = (∀ *b*∈*Rep-pd-basis t*. *a* ⊑ *b*)
unfolding *convex-le-def upper-le-def lower-le-def Rep-PDUnit*
using *Rep-pd-basis-nonempty* [*of t*, *folded ex-in-conv*] **by** *fast*

lemma *convex-le-PDUnit-PDPlus-iff* [*simp*]:
(*PDUnit a* ≤♮ *PDPlus t u*) = (*PDUnit a* ≤♮ *t* ∧ *PDUnit a* ≤♮ *u*)
unfolding *convex-le-PDUnit-lemma1 Rep-PDPlus* **by** *fast*

lemma *convex-le-PDUnit-lemma2*:
(*t* ≤♮ *PDUnit b*) = (∀ *a*∈*Rep-pd-basis t*. *a* ⊑ *b*)
unfolding *convex-le-def upper-le-def lower-le-def Rep-PDUnit*
using *Rep-pd-basis-nonempty* [*of t*, *folded ex-in-conv*] **by** *fast*

lemma *convex-le-PDPlus-PDUnit-iff* [*simp*]:
(*PDPlus t u* ≤♮ *PDUnit a*) = (*t* ≤♮ *PDUnit a* ∧ *u* ≤♮ *PDUnit a*)

unfolding *convex-le-PDUnit-lemma2 Rep-PDPlus* **by** *fast*

lemma *convex-le-PDPlus-lemma*:
assumes *z*: *PDPlus t u* $\leq\natural$ *z*
shows $\exists$ *v w*. *z* = *PDPlus v w* $\wedge$ *t* $\leq\natural$ *v* $\wedge$ *u* $\leq\natural$ *w*
proof (*intro exI conjI*)
let *?A* = {*b*$\in$*Rep-pd-basis z*. $\exists$ *a*$\in$*Rep-pd-basis t*. *a* $\sqsubseteq$ *b*}
let *?B* = {*b*$\in$*Rep-pd-basis z*. $\exists$ *a*$\in$*Rep-pd-basis u*. *a* $\sqsubseteq$ *b*}
let *?v* = *Abs-pd-basis ?A*
let *?w* = *Abs-pd-basis ?B*
have *Rep-v*: *Rep-pd-basis ?v* = *?A*
apply (*rule Abs-pd-basis-inverse*)
apply (*rule Rep-pd-basis-nonempty* [*of t*, *folded ex-in-conv*, *THEN exE*])
apply (*cut-tac z*, *simp only*: *convex-le-def lower-le-def*, *clarify*)
apply (*drule-tac x*=*x* **in** *bspec*, *simp add*: *Rep-PDPlus*, *erule bexE*)
apply (*simp add*: *pd-basis-def*)
apply *fast*
done
have *Rep-w*: *Rep-pd-basis ?w* = *?B*
apply (*rule Abs-pd-basis-inverse*)
apply (*rule Rep-pd-basis-nonempty* [*of u*, *folded ex-in-conv*, *THEN exE*])
apply (*cut-tac z*, *simp only*: *convex-le-def lower-le-def*, *clarify*)
apply (*drule-tac x*=*x* **in** *bspec*, *simp add*: *Rep-PDPlus*, *erule bexE*)
apply (*simp add*: *pd-basis-def*)
apply *fast*
done
show *z* = *PDPlus ?v ?w*
apply (*insert z*)
apply (*simp add*: *convex-le-def*, *erule conjE*)
apply (*simp add*: *Rep-pd-basis-inject* [*symmetric*] *Rep-PDPlus*)
apply (*simp add*: *Rep-v Rep-w*)
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp only*: *upper-le-def*)
apply (*drule* (*1*) *bspec*, *erule bexE*)
apply (*simp add*: *Rep-PDPlus*)
apply *fast*
apply *fast*
done
show *t* $\leq\natural$ *?v u* $\leq\natural$ *?w*
apply (*insert z*)
apply (*simp-all add*: *convex-le-def upper-le-def lower-le-def Rep-PDPlus Rep-v Rep-w*)
apply *fast+*
done
qed

lemma *convex-le-induct* [*induct set*: *convex-le*]:
assumes *le*: *t* $\leq\natural$ *u*

assumes *2*: $\bigwedge t\ u\ v.\ [\![P\ t\ u;\ P\ u\ v]\!] \Longrightarrow P\ t\ v$
assumes *3*: $\bigwedge a\ b.\ a \sqsubseteq b \Longrightarrow P\ (PDUnit\ a)\ (PDUnit\ b)$
assumes *4*: $\bigwedge t\ u\ v\ w.\ [\![P\ t\ v;\ P\ u\ w]\!] \Longrightarrow P\ (PDPlus\ t\ u)\ (PDPlus\ v\ w)$
shows $P\ t\ u$
using *le* **apply** (*induct t arbitrary*: *u rule*: *pd-basis-induct*)
apply (*erule rev-mp*)
apply (*induct-tac u rule*: *pd-basis-induct1*)
apply (*simp add*: *3*)
apply (*simp*, *clarify*, *rename-tac a b t*)
apply (*subgoal-tac P* (*PDPlus* (*PDUnit a*) (*PDUnit a*)) (*PDPlus* (*PDUnit b*) *t*))
apply (*simp add*: *PDPlus-absorb*)
apply (*erule* (*1*) *4* [*OF 3*])
apply (*drule convex-le-PDPlus-lemma*, *clarify*)
apply (*simp add*: *4*)
done

lemma *approx-pd-convex-mono1*:
$i \leq j \Longrightarrow$ *approx-pd i t* $\leq\natural$ *approx-pd j t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-approx-mono1*)
apply (*simp add*: *PDPlus-convex-mono*)
done

lemma *approx-pd-convex-le*: *approx-pd i t* $\leq\natural$ *t*
apply (*induct t rule*: *pd-basis-induct*)
apply (*simp add*: *compact-approx-le*)
apply (*simp add*: *PDPlus-convex-mono*)
done

lemma *approx-pd-convex-mono*:
t $\leq\natural$ *u* $\Longrightarrow$ *approx-pd n t* $\leq\natural$ *approx-pd n u*
apply (*erule convex-le-induct*)
apply (*erule* (*1*) *convex-le-trans*)
apply (*simp add*: *compact-approx-mono*)
apply (*simp add*: *PDPlus-convex-mono*)
done

25.2 Type definition

cpodef (**open**) *'a convex-pd* =
{*S*::*'a*::*profinite pd-basis set*. *convex-le.ideal S*}
apply (*simp add*: *convex-le.adm-ideal*)
apply (*fast intro*: *convex-le.ideal-principal*)
done

lemma *ideal-Rep-convex-pd*: *convex-le.ideal* (*Rep-convex-pd xs*)
by (*rule Rep-convex-pd* [*unfolded mem-Collect-eq*])

lemma *Rep-convex-pd-mono*: *xs* $\sqsubseteq$ *ys* $\Longrightarrow$ *Rep-convex-pd xs* $\subseteq$ *Rep-convex-pd ys*

unfolding *less-convex-pd-def less-set-eq* **.**

definition
convex-principal :: *'a pd-basis* ⇒ *'a convex-pd* **where**
convex-principal t = *Abs-convex-pd* {*u*. *u* ≤♮ *t*}

lemma *Rep-convex-principal*:
Rep-convex-pd (*convex-principal t*) = {*u*. *u* ≤♮ *t*}
unfolding *convex-principal-def*
apply (*rule Abs-convex-pd-inverse* [*simplified*])
apply (*rule convex-le.ideal-principal*)
done

interpretation *convex-pd*:
ideal-completion [*convex-le approx-pd convex-principal Rep-convex-pd*]
apply *unfold-locales*
apply (*rule approx-pd-convex-le*)
apply (*rule approx-pd-idem*)
apply (*erule approx-pd-convex-mono*)
apply (*rule approx-pd-convex-mono1*, *simp*)
apply (*rule finite-range-approx-pd*)
apply (*rule ex-approx-pd-eq*)
apply (*rule ideal-Rep-convex-pd*)
apply (*rule cont-Rep-convex-pd*)
apply (*rule Rep-convex-principal*)
apply (*simp only*: *less-convex-pd-def less-set-eq*)
done

lemma *convex-principal-less-iff* [*simp*]:
convex-principal t ⊑ *convex-principal u* ⟷ *t* ≤♮ *u*
by (*rule convex-pd.principal-less-iff*)

lemma *convex-principal-eq-iff* [*simp*]:
convex-principal t = *convex-principal u* ⟷ *t* ≤♮ *u* ∧ *u* ≤♮ *t*
by (*rule convex-pd.principal-eq-iff*)

lemma *convex-principal-mono*:
t ≤♮ *u* ⟹ *convex-principal t* ⊑ *convex-principal u*
by (*rule convex-pd.principal-mono*)

lemma *compact-convex-principal*: *compact* (*convex-principal t*)
by (*rule convex-pd.compact-principal*)

lemma *convex-pd-minimal*: *convex-principal* (*PDUnit compact-bot*) ⊑ *ys*
by (*induct ys rule*: *convex-pd.principal-induct*, *simp*, *simp*)

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

lemma *inst-convex-pd-pcpo*: $\bot = \textit{convex-principal}\ (\textit{PDUnit}\ \textit{compact-bot})$
by (*rule convex-pd-minimal* [*THEN UU-I*, *symmetric*])

25.3 Approximation

instantiation *convex-pd* :: (*profinite*) *profinite*
begin

definition
approx-convex-pd-def: *approx* = *convex-pd.completion-approx*

instance
apply (*intro-classes*, *unfold approx-convex-pd-def*)
apply (*simp add*: *convex-pd.chain-completion-approx*)
apply (*rule convex-pd.lub-completion-approx*)
apply (*rule convex-pd.completion-approx-idem*)
apply (*rule convex-pd.finite-fixes-completion-approx*)
done

end

instance *convex-pd* :: (*bifinite*) *bifinite* **..**

lemma *approx-convex-principal* [*simp*]:
$\textit{approx}\ n{\cdot}(\textit{convex-principal}\ t) = \textit{convex-principal}\ (\textit{approx-pd}\ n\ t)$
unfolding *approx-convex-pd-def*
by (*rule convex-pd.completion-approx-principal*)

lemma *approx-eq-convex-principal*:
$\exists\, t{\in}\textit{Rep-convex-pd}\ xs.\ \textit{approx}\ n{\cdot}xs = \textit{convex-principal}\ (\textit{approx-pd}\ n\ t)$
unfolding *approx-convex-pd-def*
by (*rule convex-pd.completion-approx-eq-principal*)

lemma *compact-imp-convex-principal*:
$\textit{compact}\ xs \Longrightarrow \exists\, t.\ xs = \textit{convex-principal}\ t$
apply (*drule bifinite-compact-eq-approx*)
apply (*erule exE*)
apply (*erule subst*)
apply (*cut-tac n=i* **and** *xs=xs* **in** *approx-eq-convex-principal*)
apply *fast*
done

lemma *convex-principal-induct*:
$[\![\textit{adm}\ P;\ \bigwedge t.\ P\ (\textit{convex-principal}\ t)]\!] \Longrightarrow P\ xs$
by (*rule convex-pd.principal-induct*)

lemma *convex-principal-induct2*:
$[\![\bigwedge ys.\ \textit{adm}\ (\lambda xs.\ P\ xs\ ys);\ \bigwedge xs.\ \textit{adm}\ (\lambda ys.\ P\ xs\ ys);$
$\bigwedge t\ u.\ P\ (\textit{convex-principal}\ t)\ (\textit{convex-principal}\ u)]\!] \Longrightarrow P\ xs\ ys$

apply (*rule-tac x=ys* **in** *spec*)
apply (*rule-tac xs=xs* **in** *convex-principal-induct*, *simp*)
apply (*rule allI*, *rename-tac ys*)
apply (*rule-tac xs=ys* **in** *convex-principal-induct*, *simp*)
apply *simp*
done

25.4 Monadic unit and plus

definition
convex-unit :: *'a* → *'a convex-pd* **where**
convex-unit = *compact-basis.basis-fun* (λ*a*. *convex-principal* (*PDUnit a*))

definition
convex-plus :: *'a convex-pd* → *'a convex-pd* → *'a convex-pd* **where**
convex-plus = *convex-pd.basis-fun* (λ*t*. *convex-pd.basis-fun* (λ*u*.
convex-principal (*PDPlus t u*)))

abbreviation
convex-add :: *'a convex-pd* ⇒ *'a convex-pd* ⇒ *'a convex-pd*
(**infixl** +♮ *65*) **where**
xs +♮ *ys* == *convex-plus*·*xs*·*ys*

syntax
-convex-pd :: *args* ⇒ *'a convex-pd* ({-}♮)

translations
{*x*,*xs*}♮ == {*x*}♮ +♮ {*xs*}♮
{*x*}♮ == *CONST convex-unit*·*x*

lemma *convex-unit-Rep-compact-basis* [*simp*]:
{*Rep-compact-basis a*}♮ = *convex-principal* (*PDUnit a*)
unfolding *convex-unit-def*
by (*simp add*: *compact-basis.basis-fun-principal*
convex-principal-mono PDUnit-convex-mono)

lemma *convex-plus-principal* [*simp*]:
convex-principal t +♮ *convex-principal u* = *convex-principal* (*PDPlus t u*)
unfolding *convex-plus-def*
by (*simp add*: *convex-pd.basis-fun-principal*
convex-pd.basis-fun-mono PDPlus-convex-mono)

lemma *approx-convex-unit* [*simp*]:
approx n·{*x*}♮ = {*approx n*·*x*}♮
apply (*induct x rule*: *compact-basis-induct*, *simp*)
apply (*simp add*: *approx-Rep-compact-basis*)
done

lemma *approx-convex-plus* [*simp*]:

approx n·(xs +♮ ys) = approx n·xs +♮ approx n·ys
by (*induct xs ys rule*: *convex-principal-induct2*, *simp*, *simp*, *simp*)

lemma *convex-plus-assoc*:
(xs +♮ ys) +♮ zs = xs +♮ (ys +♮ zs)
apply (*induct xs ys arbitrary*: *zs rule*: *convex-principal-induct2*, *simp*, *simp*)
apply (*rule-tac xs=zs* **in** *convex-principal-induct*, *simp*)
apply (*simp add*: *PDPlus-assoc*)
done

lemma *convex-plus-commute*: *xs +♮ ys = ys +♮ xs*
apply (*induct xs ys rule*: *convex-principal-induct2*, *simp*, *simp*)
apply (*simp add*: *PDPlus-commute*)
done

lemma *convex-plus-absorb*: *xs +♮ xs = xs*
apply (*induct xs rule*: *convex-principal-induct*, *simp*)
apply (*simp add*: *PDPlus-absorb*)
done

interpretation *aci-convex-plus*: *ab-semigroup-idem-mult* [*op +♮*]
by *unfold-locales*
(*rule convex-plus-assoc convex-plus-commute convex-plus-absorb*)+

lemma *convex-plus-left-commute*: *xs +♮ (ys +♮ zs) = ys +♮ (xs +♮ zs)*
by (*rule aci-convex-plus.mult-left-commute*)

lemma *convex-plus-left-absorb*: *xs +♮ (xs +♮ ys) = xs +♮ ys*
by (*rule aci-convex-plus.mult-left-idem*)

lemmas *convex-plus-aci = aci-convex-plus.mult-ac-idem*

lemma *convex-unit-less-plus-iff* [*simp*]:
{x}♮ ⊑ ys +♮ zs ⟷ {x}♮ ⊑ ys ∧ {x}♮ ⊑ zs
apply (*rule iffI*)
apply (*subgoal-tac*
adm (*λf. f·{x}♮ ⊑ f·ys ∧ f·{x}♮ ⊑ f·zs*))
apply (*drule admD*, *rule chain-approx*)
apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*)
apply (*cut-tac x=approx i·x* **in** *compact-imp-Rep-compact-basis*, *simp*)
apply (*cut-tac xs=approx i·ys* **in** *compact-imp-convex-principal*, *simp*)
apply (*cut-tac xs=approx i·zs* **in** *compact-imp-convex-principal*, *simp*)
apply (*clarify*, *simp*)
apply *simp*
apply *simp*
apply (*erule conjE*)
apply (*subst convex-plus-absorb* [*of* {*x*}♮, *symmetric*])
apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *convex-plus-less-unit-iff* [*simp*]:
  $xs +\natural ys \sqsubseteq \{z\}\natural \longleftrightarrow xs \sqsubseteq \{z\}\natural \wedge ys \sqsubseteq \{z\}\natural$
 apply (*rule iffI*)
  apply (*subgoal-tac*
    adm ($\lambda f.\ f{\cdot}xs \sqsubseteq f{\cdot}\{z\}\natural \wedge f{\cdot}ys \sqsubseteq f{\cdot}\{z\}\natural$))
   apply (*drule admD*, *rule chain-approx*)
    apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*)
    apply (*cut-tac xs=approx i·xs* **in** *compact-imp-convex-principal*, *simp*)
    apply (*cut-tac xs=approx i·ys* **in** *compact-imp-convex-principal*, *simp*)
    apply (*cut-tac x=approx i·z* **in** *compact-imp-Rep-compact-basis*, *simp*)
    apply (*clarify*, *simp*)
   apply *simp*
  apply *simp*
 apply (*erule conjE*)
 apply (*subst convex-plus-absorb* [*of* $\{z\}\natural$, *symmetric*])
 apply (*erule* (*1*) *monofun-cfun* [*OF monofun-cfun-arg*])
done

lemma *convex-unit-less-iff* [*simp*]: $\{x\}\natural \sqsubseteq \{y\}\natural \longleftrightarrow x \sqsubseteq y$
 apply (*rule iffI*)
  apply (*rule bifinite-less-ext*)
  apply (*drule-tac f=approx i* **in** *monofun-cfun-arg*, *simp*)
  apply (*cut-tac x=approx i·x* **in** *compact-imp-Rep-compact-basis*, *simp*)
  apply (*cut-tac x=approx i·y* **in** *compact-imp-Rep-compact-basis*, *simp*)
  apply (*clarify*, *simp add*: *compact-le-def*)
 apply (*erule monofun-cfun-arg*)
done

lemma *convex-unit-eq-iff* [*simp*]: $\{x\}\natural = \{y\}\natural \longleftrightarrow x = y$
unfolding *po-eq-conv* **by** *simp*

lemma *convex-unit-strict* [*simp*]: $\{\bot\}\natural = \bot$
unfolding *inst-convex-pd-pcpo Rep-compact-bot* [*symmetric*] **by** *simp*

lemma *convex-unit-strict-iff* [*simp*]: $\{x\}\natural = \bot \longleftrightarrow x = \bot$
unfolding *convex-unit-strict* [*symmetric*] **by** (*rule convex-unit-eq-iff*)

lemma *compact-convex-unit-iff* [*simp*]:
  compact $\{x\}\natural \longleftrightarrow$ *compact x*
unfolding *bifinite-compact-iff* **by** *simp*

lemma *compact-convex-plus* [*simp*]:
  $[\![$*compact xs*; *compact ys*$]\!] \Longrightarrow$ *compact* $(xs +\natural ys)$
apply (*drule compact-imp-convex-principal*)+
apply (*auto simp add*: *compact-convex-principal*)
done

25.5 Induction rules

lemma *convex-pd-induct1*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\natural$
assumes *insert*: $\bigwedge x\ ys.\ [\![P\ \{x\}\natural;\ P\ ys]\!] \Longrightarrow P\ (\{x\}\natural +\natural\ ys)$
shows *P* (*xs*::'*a convex-pd*)
apply (*induct xs rule*: *convex-principal-induct*, *rule P*)
apply (*induct-tac t rule*: *pd-basis-induct1*)
apply (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*])
apply (*rule unit*)
apply (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*]
convex-plus-principal [*symmetric*])
apply (*erule insert* [*OF unit*])
done

lemma *convex-pd-induct*:
assumes *P*: *adm P*
assumes *unit*: $\bigwedge x.\ P\ \{x\}\natural$
assumes *plus*: $\bigwedge xs\ ys.\ [\![P\ xs;\ P\ ys]\!] \Longrightarrow P\ (xs +\natural\ ys)$
shows *P* (*xs*::'*a convex-pd*)
apply (*induct xs rule*: *convex-principal-induct*, *rule P*)
apply (*induct-tac t rule*: *pd-basis-induct*)
apply (*simp only*: *convex-unit-Rep-compact-basis* [*symmetric*] *unit*)
apply (*simp only*: *convex-plus-principal* [*symmetric*] *plus*)
done

25.6 Monadic bind

definition
convex-bind-basis ::
'*a pd-basis* ⇒ ('*a* → '*b convex-pd*) → '*b convex-pd* **where**
convex-bind-basis = *fold-pd*
($\lambda a.\ \Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a)$)
($\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\natural\ y{\cdot}f$)

lemma *ACI-convex-bind*:
ab-semigroup-idem-mult ($\lambda x\ y.\ \Lambda\ f.\ x{\cdot}f +\natural\ y{\cdot}f$)
apply *unfold-locales*
apply (*simp add*: *convex-plus-assoc*)
apply (*simp add*: *convex-plus-commute*)
apply (*simp add*: *convex-plus-absorb eta-cfun*)
done

lemma *convex-bind-basis-simps* [*simp*]:
convex-bind-basis (*PDUnit a*) =
($\Lambda\ f.\ f{\cdot}(Rep\text{-}compact\text{-}basis\ a)$)
convex-bind-basis (*PDPlus t u*) =
($\Lambda\ f.$ *convex-bind-basis* $t{\cdot}f +\natural$ *convex-bind-basis* $u{\cdot}f$)
unfolding *convex-bind-basis-def*

apply −
apply (*rule fold-pd-PDUnit* [*OF ACI-convex-bind*])
apply (*rule fold-pd-PDPlus* [*OF ACI-convex-bind*])
done

lemma *monofun-LAM*:
⟦*cont f*; *cont g*; ⋀*x*. *f x* ⊑ *g x*⟧ ⟹ (Λ *x*. *f x*) ⊑ (Λ *x*. *g x*)
by (*simp add*: *expand-cfun-less*)

lemma *convex-bind-basis-mono*:
t ≤♮ *u* ⟹ *convex-bind-basis t* ⊑ *convex-bind-basis u*
apply (*erule convex-le-induct*)
apply (*erule* (*1*) *trans-less*)
apply (*simp add*: *monofun-LAM compact-le-def monofun-cfun*)
apply (*simp add*: *monofun-LAM compact-le-def monofun-cfun*)
done

definition
convex-bind :: *'a convex-pd* → (*'a* → *'b convex-pd*) → *'b convex-pd* **where**
convex-bind = *convex-pd.basis-fun convex-bind-basis*

lemma *convex-bind-principal* [*simp*]:
convex-bind·(*convex-principal t*) = *convex-bind-basis t*
unfolding *convex-bind-def*
apply (*rule convex-pd.basis-fun-principal*)
apply (*erule convex-bind-basis-mono*)
done

lemma *convex-bind-unit* [*simp*]:
convex-bind·{*x*}♮·*f* = *f*·*x*
by (*induct x rule*: *compact-basis-induct*, *simp*, *simp*)

lemma *convex-bind-plus* [*simp*]:
convex-bind·(*xs* +♮ *ys*)·*f* = *convex-bind*·*xs*·*f* +♮ *convex-bind*·*ys*·*f*
by (*induct xs ys rule*: *convex-principal-induct2*, *simp*, *simp*, *simp*)

lemma *convex-bind-strict* [*simp*]: *convex-bind*·⊥·*f* = *f*·⊥
unfolding *convex-unit-strict* [*symmetric*] **by** (*rule convex-bind-unit*)

25.7 Map and join

definition
convex-map :: (*'a* → *'b*) → *'a convex-pd* → *'b convex-pd* **where**
convex-map = (Λ *f xs*. *convex-bind*·*xs*·(Λ *x*. {*f*·*x*}♮))

definition
convex-join :: *'a convex-pd convex-pd* → *'a convex-pd* **where**
convex-join = (Λ *xss*. *convex-bind*·*xss*·(Λ *xs*. *xs*))

lemma *convex-map-unit* [*simp*]:
convex-map·*f*·(*convex-unit*·*x*) = *convex-unit*·(*f*·*x*)
unfolding *convex-map-def* **by** *simp*

lemma *convex-map-plus* [*simp*]:
convex-map·*f*·(*xs* +♮ *ys*) = *convex-map*·*f*·*xs* +♮ *convex-map*·*f*·*ys*
unfolding *convex-map-def* **by** *simp*

lemma *convex-join-unit* [*simp*]:
convex-join·{*xs*}♮ = *xs*
unfolding *convex-join-def* **by** *simp*

lemma *convex-join-plus* [*simp*]:
convex-join·(*xss* +♮ *yss*) = *convex-join*·*xss* +♮ *convex-join*·*yss*
unfolding *convex-join-def* **by** *simp*

lemma *convex-map-ident*: *convex-map*·(Λ *x*. *x*)·*xs* = *xs*
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-map-map*:
convex-map·*f*·(*convex-map*·*g*·*xs*) = *convex-map*·(Λ *x*. *f*·(*g*·*x*))·*xs*
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-join-map-unit*:
convex-join·(*convex-map*·*convex-unit*·*xs*) = *xs*
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-join-map-join*:
convex-join·(*convex-map*·*convex-join*·*xsss*) = *convex-join*·(*convex-join*·*xsss*)
by (*induct xsss rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-join-map-map*:
convex-join·(*convex-map*·(*convex-map*·*f*)·*xss*) =
convex-map·*f*·(*convex-join*·*xss*)
by (*induct xss rule*: *convex-pd-induct*, *simp-all*)

lemma *convex-map-approx*: *convex-map*·(*approx n*)·*xs* = *approx n*·*xs*
by (*induct xs rule*: *convex-pd-induct*, *simp-all*)

25.8 Conversions to other powerdomains

Convex to upper

lemma *convex-le-imp-upper-le*: *t* ≤♮ *u* ⟹ *t* ≤♯ *u*
unfolding *convex-le-def* **by** *simp*

definition
convex-to-upper :: *'a convex-pd* → *'a upper-pd* **where**
convex-to-upper = *convex-pd.basis-fun upper-principal*

lemma *convex-to-upper-principal* [*simp*]:
convex-to-upper·(*convex-principal t*) = *upper-principal t*
unfolding *convex-to-upper-def*
apply (*rule convex-pd.basis-fun-principal*)
apply (*rule upper-principal-mono*)
apply (*erule convex-le-imp-upper-le*)
done

lemma *convex-to-upper-unit* [*simp*]:
convex-to-upper·{*x*}♮ = {*x*}♯
by (*induct x rule*: *compact-basis-induct*, *simp*, *simp*)

lemma *convex-to-upper-plus* [*simp*]:
convex-to-upper·(*xs* +♮ *ys*) = *convex-to-upper*·*xs* +♯ *convex-to-upper*·*ys*
by (*induct xs ys rule*: *convex-principal-induct2*, *simp*, *simp*, *simp*)

lemma *approx-convex-to-upper*:
approx i·(*convex-to-upper*·*xs*) = *convex-to-upper*·(*approx i*·*xs*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

Convex to lower

lemma *convex-le-imp-lower-le*: *t* ≤♮ *u* ⟹ *t* ≤♭ *u*
unfolding *convex-le-def* **by** *simp*

definition
convex-to-lower :: *'a convex-pd* → *'a lower-pd* **where**
convex-to-lower = *convex-pd.basis-fun lower-principal*

lemma *convex-to-lower-principal* [*simp*]:
convex-to-lower·(*convex-principal t*) = *lower-principal t*
unfolding *convex-to-lower-def*
apply (*rule convex-pd.basis-fun-principal*)
apply (*rule lower-principal-mono*)
apply (*erule convex-le-imp-lower-le*)
done

lemma *convex-to-lower-unit* [*simp*]:
convex-to-lower·{*x*}♮ = {*x*}♭
by (*induct x rule*: *compact-basis-induct*, *simp*, *simp*)

lemma *convex-to-lower-plus* [*simp*]:
convex-to-lower·(*xs* +♮ *ys*) = *convex-to-lower*·*xs* +♭ *convex-to-lower*·*ys*
by (*induct xs ys rule*: *convex-principal-induct2*, *simp*, *simp*, *simp*)

lemma *approx-convex-to-lower*:
approx i·(*convex-to-lower*·*xs*) = *convex-to-lower*·(*approx i*·*xs*)
by (*induct xs rule*: *convex-pd-induct*, *simp*, *simp*, *simp*)

Ordering property

lemma *convex-pd-less-iff*:
(*xs* ⊑ *ys*) =
(*convex-to-upper*·*xs* ⊑ *convex-to-upper*·*ys* ∧
convex-to-lower·*xs* ⊑ *convex-to-lower*·*ys*)
apply (*safe elim*!: *monofun-cfun-arg*)
apply (*rule bifinite-less-ext*)
apply (*drule-tac f*=*approx i* **in** *monofun-cfun-arg*)
apply (*drule-tac f*=*approx i* **in** *monofun-cfun-arg*)
apply (*cut-tac xs*=*approx i*·*xs* **in** *compact-imp-convex-principal*, *simp*)
apply (*cut-tac xs*=*approx i*·*ys* **in** *compact-imp-convex-principal*, *simp*)
apply *clarify*
apply (*simp add*: *approx-convex-to-upper approx-convex-to-lower convex-le-def*)
done

lemmas *convex-plus-less-plus-iff* =
convex-pd-less-iff [**where** *xs*=*xs* +♮ *ys* **and** *ys*=*zs* +♮ *ws*, *standard*]

lemmas *convex-pd-less-simps* =
convex-unit-less-plus-iff
convex-plus-less-unit-iff
convex-plus-less-plus-iff
convex-unit-less-iff
convex-to-upper-unit
convex-to-upper-plus
convex-to-lower-unit
convex-to-lower-plus
upper-pd-less-simps
lower-pd-less-simps

end

theory *HOLCF*
imports *Sprod Ssum Up Lift Discrete One Tr Domain ConvexPD Main*
uses
holcf-logic.ML
Tools/cont-consts.ML
Tools/domain/domain-library.ML
Tools/domain/domain-syntax.ML
Tools/domain/domain-axioms.ML
Tools/domain/domain-theorems.ML
Tools/domain/domain-extender.ML
Tools/adm-tac.ML

begin

defaultsort *pcpo*

declaration ⟪ *fn* - =>
  Simplifier.map-ss (*fn simpset* => *simpset addSolver*
    (*mk-solver'* *adm-tac* (*fn ss* =>
      adm-tac (*cut-facts-tac* (*Simplifier.prems-of-ss ss*) *THEN'* *cont-tacRs ss*))));
⟫

end