Some Domain Theory and Denotational Semantics in Coq

Nick Benton[1], Andrew Kennedy[1], and Carsten Varming[2]*

[1] Microsoft Research, Cambridge, UK
{nick,akenn}@microsoft.com
[2] Carnegie-Mellon University, Pittsburgh, USA
varming@cmu.edu

Abstract. We present a Coq formalization of constructive ω-cpos (extending earlier work by Paulin-Mohring) up to and including the inverse-limit construction of solutions to mixed-variance recursive domain equations, and the existence of invariant relations on those solutions. We then define operational and denotational semantics for both a simply-typed CBV language with recursion and an untyped CBV language, and establish soundness and adequacy results in each case.

1 Introduction

The use of proof assistants in formalizing language metatheory and implementing certified tools has grown enormously over the last five years or so. Most current work on mechanizing language definitions and type soundness results, certified compilation, proof carrying code, and so on has been based on operational semantics. But in our work on both certified compilation and on the semantics of languages with state, we have often found ourselves wanting a Coq formalization of the kind of denotational semantics that we have grown accustomed to working with on paper.

Mechanizing domain theory and denotational semantics has an illustrious history. Provers such as HOL, Isabelle/HOL and Coq can all trace an ancestral line back to Milner's LCF [16], which was a proof checker for Scott's PPλ logic of cpos, continuous functions and admissible predicates. And although later systems were built on less domain-specific foundations, there have subsequently been dozens of formalizations of different notions of domains and bits of semantics, with examples in all the major provers. Few, however, have really gone far enough to be useful. This paper describes our Coq formalization of ω-cpos and the denotational semantics of both typed and untyped versions of a simple functional language, going considerably further than previous work. A companion paper [8] describes a non-trivial compiler correctness theorem that has been formalized and proved using one of these denotational models.

* Research supported in part by National Science Foundation Grants CCF-0541021, CCF-0429505.

Our formalization is based on a Coq library for constructive pointed ω-cpos and continuous functions written by Paulin-Mohring [20] as a basis for a semantics of Kahn networks, and of probabilistic programs [6]. Section 2 describes our slight generalization of Paulin-Mohring's library to treat predomains and a general lift monad. In Section 3, we then define a simply-typed call-by-value functional language, give it a denotational semantics using our predomains and prove the standard soundness and adequacy theorems, establishing the correspondence between the operational and denotational semantics. These results seem not to have been previously mechanized for a higher-order language.

Section 4 is about solving recursive domain equations. We formalize Scott's inverse limit construction along the lines of work by Freyd [11, 12] and Pitts [22, 23]. This approach characterizes the solutions as minimal invariants, yielding reasoning principles that allow one to construct and work with recursively-defined predicates and relations over the recursively-defined domains. In Section 5, we define the semantics of an untyped call-by-value language using a particular recursive domain, and use the associated reasoning principles to again establish soundness and adequacy theorems.

2 Basic Domain Theory

This first part of the development is essentially unchanged from the earlier work of Paulin-Mohring [20]. The main difference is that Paulin-Mohring formalized *pointed* cpos and continuous maps, with a special-case construction of flat cpos (those that arise from adding a bottom element under all elements of an otherwise discretely ordered set), whereas we use potentially bottomless cpos ('predomains') and formalize a general constructive lift monad.

2.1 Complete Partial Orders

We start by defining the type of preorders, comprising a carrier type *tord* (to which :> means we can implicitly coerce), a binary relation *Ole* (written infix as $\sqsubseteq$), and proofs that *Ole* is reflexive and transitive:

```
Record ord:= mk_ord
  {tord :> Type;
   Ole : tord → tord → Prop;
   Ole_refl : ∀ x : tord, Ole x x;
   Ole_trans : ∀ x y z : tord, Ole x y → Ole y z → Ole x z}.
Infix "⊑" := Ole.
```

The equivalence relation == is then defined to be the symmetrisation of $\sqsubseteq$:

```
Definition Oeq (O : ord) (x y : O) := x ⊑ y ∧ y ⊑ x.
Infix "==" := Oeq (at level 70).
```

Both == and $\sqsubseteq$ are declared as parametric `Setoid` relations, with $\sqsubseteq$ being a partial order modulo ==. Most of the constructions that follow are proved and declared to be morphisms with respect to these relations, which then allows convenient (in)equational rewriting in proofs.

The type of monotone functions between partial orders is a parameterized record type, comprising a function between the underlying types of the two order parameters and a proof that that function preserves order:

```
Definition monotonic (O1 O2 : ord) (f : O1 → O2) := ∀ x y, x ⊑ y → f x ⊑ f y.
Record fmono (O1 O2 : ord) := mk_fmono
                {fmonot :> O1 → O2;
                 fmonotonic: monotonic fmonot}.
```

For any $O_1\ O_2 : \mathit{ord}$, the monotonic function space $O_1 \to_m O_2 : \mathit{ord}$ is defined by equipping $\mathit{fmono}\ O_1\ O_2$ with the order inherited from the codomain, $f \sqsubseteq g$ iff $f\,x \sqsubseteq g\,x$ for all x.

We define $\mathit{natO} : \mathit{ord}$ by equipping the set of natural numbers, nat, with the usual 'vertical' order, $\leq$. If $c : \mathit{natO} \to_m O$ for some $O : \mathit{ord}$, we call c a *chain* in O. Now a complete partial order is defined as a dependent record comprising an underlying order, *tord*, a function $\bigsqcup$ for computing the least upper bound of any chain in *tord*, and proofs that this *is* both an upper bound ($\mathit{le\_lub}$), and less than or equal to any other upper bound ($\mathit{lub\_le}$):

```
Record cpo:= mk_cpo
  {tcpo :> ord;
   ⨆: (natO→m tcpo) → tcpo;
   le_lub : ∀ (c : natO→m tcpo) (n : nat), c n ⊑ ⨆c;
   lub_le : ∀ (c : natO→m tcpo) (x : tcpo), (∀ n, c n ⊑ x) → ⨆c ⊑ x}.
```

This definition of a complete partial order is constructive in the sense that we require least upper bounds of chains not only to exist, but to be computable in Coq's logic of total functions.

A monotone function f between two *cpo*s, D_1 and D_2, is *continuous* if it preserves (up to ==) least upper bounds. One direction of this is already a consequence of monotonicity, so we just have to specify the other:

```
Definition continuous (D1 D2 : cpo) (f : D1 →m D2) :=
                  ∀ c : natO→m D1, f (⨆c) ⊑ ⨆(f ∘c).
Record fconti (D1 D2 : cpo) := mk_fconti
  {fcontit : D1 →m D2;
   fcontinuous : continuous fcontit}.
```

For any $D_1\ D_2 : \mathit{cpo}$, the continuous function space $D_1 \to_c D_2 : \mathit{ord}$ is defined by equipping the type $\mathit{fconti}\ D_1\ D_2$ with the pointwise order inherited from D_2. We then define $D_1 \Rightarrow_c D_2 : \mathit{cpo}$ by equipping $D_1 \to_c D_2$ with least upper bounds computed pointwise: if $c : \mathit{natO} \to_m (D_1 \to_c D_2)$ is a chain, then $\bigsqcup c : (D_1 \to_c D_2)$ is $\lambda d_1.\bigsqcup(\lambda n.c\,n\,d_1)$.

If $D : \mathit{cpo}$, write $\mathit{ID}\ D : D \to_c D$ for the continuous identity function on D. If $f : D \to_c E$ and $g : E \to_c F$ write $g \circ f : D \to_c F$ for their composition. Composition of continuous maps is associative, with ID as a unit.

Discrete cpos If X : `Type` then equipping X with the order $x_1 \sqsubseteq x_2$ iff $x_1 = x_2$ (i.e. Leibniz equality) yields a cpo that we write $\mathit{Discrete}\ X$.

Finite products Write **1** for the one-point cpo, *Discrete unit*, which is terminal, in that for any $f\ g : D \to_c \mathbf{1}$, $f == g$. If $D_1\ D_2 : \mathit{cpo}$ then equipping

the usual product of the underlying types of their underlying orders with the pointwise ordering yields a product order. Equipping that order with a pointwise least upper bound operation $\bigsqcup c = (\bigsqcup(fst \circ c), \bigsqcup(snd \circ c))$ for $c \to_m D_1 \times D_2$ yields a product cpo $D_1 \times D_2$ with continuous $\pi_i : D_1 \times D_2 \to_c D_i$. We write $\langle f, g \rangle$ for the unique (up to $==$) continuous function such that $f == \pi_1 \circ \langle f, g \rangle$ and $g == \pi_2 \circ \langle f, g \rangle$.

Closed structure We can define operations $curry : (D \times E \to_c F) \to (D \to_c E \Rightarrow_c F)$ and $ev : (E \Rightarrow_c D) \times E \to_c D$ such that for any $f : D \times E \to_c F$, $curry(f)$ is the unique continuous map such that $f == ev \circ \langle curry\, f \circ \pi_1, \pi_2 \rangle$. We define $uncurry : (D \Rightarrow_c E \Rightarrow_c F) \to_c D \times E \Rightarrow_c F$ by $uncurry = curry(ev \circ \langle ev \circ \langle \pi_1, \pi_1 \circ \pi_2 \rangle, \pi_2 \circ \pi_2 \rangle)$ and we check that $uncurry(curry(f)) == f$ and $curry(uncurry(h)) == h$ for all f and h.

So our internal category $\mathbb{CPO}$ of *cpo*s and continuous maps is Cartesian closed. We elide the details of other constructions, including finite coproducts, strict function spaces and general indexed products, that are in the formalization. Although our cpos are not required to have least elements, those that do are of special interest. We use Coq's typeclass mechanism to capture them:

```
Class Pointed(D : cpo) := { ⊥ : D; Pleast : ∀ d : D, ⊥ ⊑ d }.
Instance DOne_pointed : Pointed 1.
Instance prod_pointed A B { pa : PointedA} {pb : PointedB} : Pointed(A × B).
Instance fun_pointed A B {pb : PointedB} : Pointed(A ⇒cB).
```

Now if D is *Pointed*, and $f : D \to_c D$ then we can define *fixp f*, the least fixed point of f in the usual way, as the least upper bound of the chain of iterates of f starting at $\bot$. We define $FIXP : (D \Rightarrow_c D) \to_c D$ to be the 'internalised' version of *fixp*.

If D : *cpo* and $P : D \to Prop$, then P is *admissible* if for all chains c : $natO \to_m D$ such that $(\forall n.\ P(c_n))$, one has $P(\bigsqcup c)$. In such a case, the subset type $\{d : D \mid P(d)\}$ with the order and lubs inherited from D is a cpo. We can also prove the standard fixed point induction principle:

```
Definition fixp_ind D { pd : PointedD} : ∀ (F: D →mD)(P : D→ Prop),
      admissible P → P ⊥ → (∀ x, P x → P (F x)) → P (fixp F).
```

The main technical complexity in this part of the formalization is simply the layering of definitions, with (for example) *cpo*s being built on *ord*s, and $D \Rightarrow_c E$ being built on $D \to_c E$, which is built on $D \to_m E$, which is built on $D \to E$. Definitions have to be built up in multiple staged versions and there are many implicit coercions and hints for Coq's `auto` tactic, which are tricky to get right. There is also much boilerplate associated with morphism declarations supporting setoid rewriting, and there is some tension between the elementwise and 'point-free' styles of working.

2.2 The lift monad

The basic order theory of the previous section goes through essentially as it does when working classically on paper. In particular, the definitions of lubs in products and function spaces are already constructive. But lifting will allow

us to express general partial recursive functions, which, in Coq's logic of total functions, is clearly going to involve some work. Our solution is a slight generalization of Paulin-Mohring's treatment of the particular case of flat cpos, which in turn builds on work of Capretta [9] on general recursion in type theory. We exploit Coq's support for coinductive datatypes [10], defining lifting in terms of a type *Stream* of potentially infinite streams:

```
Variable D : cpo.
CoInductive Stream := Eps : Stream → Stream | Val : D → Stream.
```

An element of *Stream* is (classically) either the infinite $Eps\,(Eps\,(Eps\,(\ldots)))$, or some finite sequence of *Eps* steps, terminated by $Val\ d$ for some $d : D$, $Eps\,(Eps\,(\ldots\ Eps\,(Val\ d)\ \ldots))$. One can think of *Stream* as defining a resumptions monad, which we will subsequently quotient to define lifting. For x : *Stream* and n : *nat*, *pred_nth* $x\ n$ is the stream that results from removing the first n *Eps* steps from x. The order on *Stream* is coinductively defined by

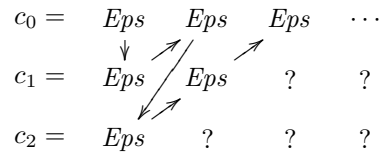
```
CoInductive DLle : Stream→ Stream→ Prop :=
 | DLleEps : ∀ x y, DLle x y → DLle (Eps x) (Eps y)
 | DLleEpsVal : ∀ x d, DLle x (Val d) → DLle (Eps x) (Val d)
 | DLleVal : ∀ d d' n y, pred_nth y n = Val d' → d ⊑ d' → DLle (Val d) y.
```

which satisfies the following coinduction principle:

```
Lemma DLle_rec : ∀ R : Stream→ Stream→ Prop,
   (∀ x y, R (Eps x) (Eps y) → R x y) →
   (∀ x d, R (Eps x) (Val d) → R x (Val d)) →
   (∀ d y, R (Val d) y → ∃ n, ∃ d', pred_nth y n = Val d' ∧ d ⊑ d')
   → ∀ x y, R x y → DLle x y.
```

The coinduction principle is used to show that *DLle* is reflexive and transitive, allowing us to construct a preorder $DL\_ord : ord$ (and we now write the usual $\sqsubseteq$ for the order). The infinite stream of *Eps*'s, Ω, is the least element of the order.

Constructing a *cpo* from *DL_ord* is slightly subtle. We need to define a function that maps chains $c : (natO \to_m DL\_ord)$ to their lubs in *DL_ord*. An important observation is that if some c_n is non-Ω, i.e. there exists a d_n such that $c_n == Val\,d_n$, then for any $m \geq n$, there is a d_m such that $c_m == Val\,d_m$ and that moreover, the sequence $d_n, d_{n+1}, \ldots$, forms a chain in D. Classically, the idea is that we look for such a c_n; if we find one, then we can construct a chain in D and return *Val* applied to the least upper bound of that chain. If there's no such chain then the least upper bound is Ω. But we cannot simply test whether a particular c_n is Ω or not: we can only examine finite prefixes. So we make a 'parallel' corecursive search through all the c_ns, which may be pictured something like this:[3]



[3] In reality, the output stream 'ticks' less frequently than the picture would suggest.

The output we are trying to produce is an element of *DL_ord*. Each time our interleaving search finds an *Eps* , we produce an *Eps* on the output. So if every element of the chain is Ω, we will end up producing Ω on the output. But should we find a *Val d* after outputting some finite number of *Eps* s, then we know all later elements of the chain are also non-Ω, so we go ahead and build the chain in D that they form and compute its least upper bound using the lub operation of D. The details of this construction, and the proof that it does indeed yield the least upper bound of the chain c, involve interesting bits of constructive reasoning: going from knowing that there *is* a chain in D to actually having that chain in one's hand so as to take its lub uses (a provable form of) constructive indefinite description, for example. But at the end of the day, we end up with a constructive definition of $D_\perp : cpo$, which is clearly *Pointed*.

Lifting gives a strong monad [17] on $\mathbb{CPO}$. The unit $\eta : D \to_c D_\perp$ applies the *Val* constructor. If $f : D \to_c E_\perp$ define *kleisli* $f : D_\perp \to_c E_\perp$ to be the map

$$cofix\ kl\ (d : D_\perp) : E_\perp\ :=\ match\ d\ with\ Eps\ dl\ \Rightarrow Eps\ (kl\ dl)\ |\ Val\ d' \Rightarrow f\ d'$$

Thinking operationally, the way in which *kleisli* sequences computations is very intuitive. To run *kleisli* f d, we start by running d. Every time d takes an *Eps* step, we do too, so if d diverges so does *kleisli* f d. Should d yield a value d', however, the remaining steps are those of $f\,d'$. We prove that *kleisli* f actually is a continuous function and, amongst other things, satisfies all the equations making $(-_\perp, \eta, kleisli(-))$ a Kleisli triple on $\mathbb{CPO}$. It is also convenient to have 'parameterized' versions of the Kleisli operators *Kleislir* $D\ E\ :\ (D \times E \to_c F_\perp) \to (D \times E_\perp \to_c F_\perp)$ defined by composing *kleisli* with the evident strength $\tau : D \times E_\perp \to_c (D \times E)_\perp$.

3 A Simply-Typed Functional Language

Our first application of the domain theory formalization is mechanize the denotational semantics of PCF_v, a simply-typed, call-by-value functional language with recursion. Types in PCF_v consist of integer, boolean, functions and products; we represent typing environments by a list of types.

```
Inductive Ty := Int | Bool | Arrow (τ1 τ2 : Ty) | Prod (τ1 τ2 : Ty).
Infix " -> " := Arrow. Infix " * " := Prod (at level 55).
Definition Env := list Ty.
```

We separate syntactic values v from general expressions e, and restrict the syntax to ANF (administrative normal form), with explicit sequencing of evaluation by *LET* and inclusion of values into expressions by *VAL*. As usual, there immediately arises the question of how to represent binders. Our first attempt used de Bruijn indices of type `nat` in the syntax representation, and a separate type for typing judgments:

```
Inductive Value := VAR : nat → Value | FIX : Ty → Ty → Exp → Value |...
Inductive VTy (Γ : Env) (τ : Ty) : Value → Type :=
| TVAR : ∀ m , nth_error Γm = Some τ→ VTy Γ (VAR m) τ
```

| *TFIX* : ∀ *e* τ_1 τ_2, ($\tau = \tau_1$ -> τ_2) → *ETy* (τ_1 :: τ_1 -> τ_2 :: Γ) *e* τ_2 → *VTy* Γ (*FIX* τ_1 τ_2 *e*) τ ...

The major drawback of the above is that typing judgments contain proof objects: simple equalities between types, as in *TFIX*, and the existence of a variable in the environment, as in *TVAR*. It's necessary to prove (at some length) that any two typings of the same term are equal, whilst definitions and theorems are hedged with well-formedness side-conditions.

We recently switched to a strongly-typed term representation in which variable and term types are indexed by *Ty* and *Env*, ensuring that terms are well-typed by construction. Definitions and theorems become more natural and much more concise, and the problems with equality proofs go away.[4] Here is the complete definition of well-typed terms:

```
Inductive Var : Env → Ty → Type :=
| ZVAR : ∀ Γ τ, Var (τ :: Γ) τ    | SVAR : ∀ Γ τ τ', Var Γ τ → Var (τ' :: Γ) τ.

Inductive Value : Env → Ty → Type :=
| TINT : ∀ Γ, nat → Value Γ Int    | TBOOL : ∀ Γ, bool → Value Γ Bool
| TVAR : ∀ Γ τ, Var Γ τ → Value Γ τ
| TFIX : ∀ Γ τ1 τ2, Exp (τ1 :: τ1 -> τ2 :: Γ) τ2 → Value Γ (τ1 -> τ2)
| TPAIR : ∀ Γ τ1 τ2, Value Γ τ1 → Value Γ τ2 → Value Γ (τ1 * τ2)
with Exp : Env → Ty → Type :=
| TFST : ∀ Γ τ1 τ2, Value Γ (τ1 * τ2) → Exp Γ τ1
| TSND : ∀ Γ τ1 τ2, Value Γ (τ1 * τ2) → Exp Γ τ2
| TOP : ∀ Γ, (nat → nat → nat) → Value Γ Int → Value Γ Int → Exp Γ Int
| TGT : ∀ Γ, Value Γ Int → Value Γ Int → Exp Γ Bool
| TVAL : ∀ Γ τ, Value Γ τ → Exp Γ τ
| TLET : ∀ Γ τ1 τ2, Exp Γ τ1 → Exp (τ1 :: Γ) τ2 → Exp Γ τ2
| TAPP : ∀ Γ τ1 τ2, Value Γ (τ1 -> τ2) → Value Γ τ1 → Exp Γ τ2
| TIF : ∀ Γ τ, Value Γ Bool → Exp Γ τ → Exp Γ τ → Exp Γ τ.

Definition CExp τ := Exp nil τ. Definition CValue τ := Value nil τ.
```

Variables of type *Var* Γ τ are represented by a "typed" de Bruijn index that is in essence a proof that τ lives at that index in Γ. The typing rule associated with each term constructor can be read directly off its definition: for example, *TLET* takes an expression typed as τ_1 under Γ, and another expression typed as τ_2 under Γ extended with a new variable of type τ_1; its whole type is then τ_2 under Γ. The abbreviations *CExp* and *CValue* define closed terms.

Now the operational semantics can be presented very directly:

```
Inductive Ev : ∀ τ, CExp τ → CValue τ → Prop :=
| e_Val : ∀ τ (v : CValue τ), TVAL v ⇓ v
| e_Op : ∀ op n1 n2, TOP op (TINT n1) (TINT n2) ⇓ TINT (op n1 n2)
| e_Gt : ∀ n1 n2, TGT (TINT n1) (TINT n2) ⇓ TBOOL (ble_nat n2 n1)
| e_Fst : ∀ τ1 τ2 (v1 : CValue τ1) (v2 : CValue τ2), TFST (TPAIR v1 v2) ⇓ v1
| e_Snd : ∀ τ1 τ2 (v1 : CValue τ1) (v2 : CValue τ2), TSND (TPAIR v1 v2) ⇓ v2
| e_App : ∀ τ1 τ2 e (v1 : CValue τ1) (v2 : CValue τ2),
    substExp (doubleSubst v1 (TFIX e)) e ⇓ v2 → TAPP (TFIX e) v1 ⇓ v2
```

[4] The new `Program` and `dependent destruction` tactics in Coq 8.2 are invaluable for working with this kind of strongly dependent representation.

```
| e_Let : ∀ τ1 τ2 e1 e2 (v1 : CValue τ1) (v2 : CValue τ2),
    e1 ⇓ v1 → substExp (singleSubst v1) e2 ⇓ v2 → TLET e1 e2 ⇓ v2
| e_IfTrue : ∀ τ (e1 e2 : CExp τ) v, e1 ⇓ v → TIF (TBOOL true) e1 e2 ⇓ v
| e_IfFalse : ∀ τ (e1 e2 : CExp τ) v, e2 ⇓ v → TIF (TBOOL false) e1 e2 ⇓ v
where "e '⇓' v" := (Ev e v).
```

Substitutions are typed maps from variables to values:

```
Definition Subst Γ Γ' := ∀ τ, Var Γ τ → Value Γ' τ.
Definition hdSubst Γ Γ' τ : Subst (τ::Γ) Γ' → Value Γ' τ := ....
Definition tlSubst Γ Γ' τ : Subst (τ::Γ) Γ' → Subst Γ Γ' := ....
```

To apply a substitution on de Bruijn terms (functions *substVal* and *substExp*), one would conventionally define a *shift* operator, but the full dependent type of this operator (namely $Val(\Gamma +\!\!+ \Gamma') \tau \to Val(\Gamma +\!\!+ [\tau'] +\!\!+ \Gamma') \tau$) is hard to work with. Instead, we first define general *renamings* (maps from variables to variables), and then bootstrap substitution on terms, defining *shift* in terms of renaming [5, 15, 1]. Definitions and lemmas regarding composition must be similarly bootstrapped: first composition of renamings is defined, then composition of substitution with renaming, and finally composition of substitutions.

3.1 Denotational semantics

The semantics of types is inductive, using product of cpo's to interpret products and continuous functions into a lifted cpo to represent call-by-value functions.

```
Fixpoint SemTy τ := match τ with
 | Int ⇒ Discrete nat   | Bool ⇒ Discrete bool
 | τ1 -> τ2 ⇒ SemTy τ1 ⇒c (SemTy τ2)⊥
 | τ1 * τ2 ⇒ SemTy τ1 × SemTy τ2   end.
Fixpoint SemEnv Γ := match Γ with nil ⇒ 1 | τ :: Γ' ⇒ SemEnv Γ' × SemTy τ
end.
```

We interpret $Value\ \Gamma\ \tau$ in $SemEnv\ \Gamma \to_c SemTy\ \tau$. Expressions are similar, except that the range is a lifted cpo. Note how we have used a 'point-free' style, with no explicit mention of value environments.

```
Fixpoint SemVar Γ τ (var : Var Γ τ) : SemEnv Γ →c SemTy τ :=
match var with   ZVAR _ _ ⇒ π2   | SVAR _ _ _ v ⇒ SemVar v ∘ π1   end.

Fixpoint SemExp Γ τ (e : Exp Γ τ) : SemEnv Γ →c(SemTy τ)⊥:=
match e with
| TOP _ op v1 v2 ⇒ η ∘ uncurry (SimpleOp2 op) ∘ ⟨ SemVal v1 , SemVal v2 ⟩
| TGT _ v1 v2 ⇒ η ∘ uncurry (SimpleOp2 ble_nat) ∘ ⟨ SemVal v2 , SemVal v1 ⟩
| TAPP _ _ _ v1 v2 ⇒ ev ∘ ⟨ SemVal v1 , SemVal v2 ⟩
| TVAL _ _ v ⇒ η ∘ SemVal v
| TLET _ _ _ e1 e2 ⇒ Kleislir(SemExp e2) ∘ ⟨ ID, SemExp e1 ⟩
| TIF _ _ v e1 e2 ⇒ (choose _ @3_ (SemExp e1)) (SemExp e2) (SemVal v)
| TFST _ _ _ v ⇒ η ∘ π1∘ SemVal v
| TSND _ _ _ v ⇒ η ∘ π2∘ SemVal v
end with SemVal Γ τ (v : Value Γ τ) : SemEnv Γ →cSemTy τ :=
match v with
| TINT _ n ⇒ K _ (n : Discrete nat)
```

```
| TBOOL _ b ⇒ K _ (b : Discrete bool)
| TVAR _ _ i ⇒ SemVar i
| TFIX _ _ _ e ⇒ FIXP ∘ curry (curry (SemExp e))
| TPAIR _ _ _ v1 v2 ⇒ ⟨ SemVal v1 , SemVal v2 ⟩    end.
```

In the above *SimpleOp2* lifts binary Coq functions to continuous maps on discrete cpos, K is the usual combinator and *choose* is a continuous conditional.

3.2 Soundness and adequacy

We first prove *soundness*, showing that if an expression e evaluates to a value v, then the denotation of e is indeed the denotation of v. This requires that substitution commutes with the semantic meaning function. We define the 'semantics' of a substitution $s : Subst\ \Gamma'\ \Gamma$ to be a map in $SemEnv\ \Gamma \to_c SemEnv\ \Gamma'$.

```
Fixpoint SemSubst Γ Γ' : Subst Γ' Γ → SemEnv Γ →c SemEnv Γ' :=
  match Γ' with
  | nil ⇒ fun s ⇒ K _ (tt : 1)
  | _ :: _ ⇒ fun s ⇒ ⟨ SemSubst (tlSubst s) , SemVal (hdSubst s) ⟩ end.
```

This is then used to prove the substitution lemma, which in turn is used in the *e_App* and *e_Let* cases of the soundness proof.

```
Lemma SemCommutesWithSubst:
    (∀ Γ τ (v : Value Γ τ) Γ' (s : Subst Γ Γ'),
    SemVal v ∘ SemSubst s == SemVal (substVal s v))
∧ (∀ Γ τ (e : Exp Γ τ) Γ' (s : Subst Γ Γ'),
    SemExp e ∘ SemSubst s == SemExp (substExp s e)).

Theorem Soundness: ∀ τ (e : CExp τ) v, e ⇓ v → SemExp e == η ∘ SemVal v.
```

We now prove *adequacy*, showing that if the denotation of a closed expression e is some (lifted) element, then e converges to a value. The proof uses a logical relation between syntax and semantics. We start by defining a *liftRel* operation that takes a relation between a cpo and values and lifts it to a relation between a lifted cpo and expressions, then use this to define *relExp* in terms of *relVal*.

```
Definition liftRel τ (R : SemTy τ → CValue τ → Prop) :=
  fun d e ⇒ ∀ d', d == Val d' → ∃ v, e ⇓ v ∧ R d' v.

Fixpoint relVal τ : SemTy τ → CValue τ → Prop := match τ with
| Int ⇒ fun d v ⇒ v = TINT d
| Bool ⇒ fun d v ⇒ v = TBOOL d
| τ1 -> τ2 ⇒ fun d v ⇒ ∃ e, v = TFIX e ∧ ∀ d1 v1, relVal τ1 d1 v1 → liftRel (relVal
τ2) (d d1) (substExp (doubleSubst v1 v ) e)
| τ1 * τ2 ⇒ fun d v ⇒ ∃ v1, ∃ v2, v = TPAIR v1 v2 ∧ relVal τ1 (π1 d) v1 ∧ relVal τ2
(π2 d) v2 end.

Fixpoint relEnv Γ : SemEnv Γ → Subst Γ nil → Prop := match Γ with
  | nil ⇒ fun _ _ ⇒ True
  | τ :: Γ ⇒ fun d s ⇒ relVal τ (π2 d) (hdSubst s) ∧ relEnv Γ (π1 d) (tlSubst s) end.

Definition relExp τ := liftRel (relVal τ).
```

The logical relation reflects == and is admissible:

```
Lemma relVal_lower: ∀ τ d d' v, d ⊑ d' → relVal τ d' v → relVal τ d v.
Lemma relVal_admissible: ∀ τ v, admissible (fun d ⇒ relVal τ d v).
```

These lemmas are then used in the proof of the Fundamental Theorem for the logical relation, which is proved by induction on the structure of terms.

```
Theorem FT: (∀ Γ τ v ρ s, relEnv Γ ρ s → relVal τ (SemVal v ρ) (substVal s v))
          ∧ (∀ Γ τ e ρ s, relEnv Γ ρ s → relExp τ (SemExp e ρ) (substExp s e)).
```

Now we instantiate the fundamental theorem with closed expressions to obtain

```
Corollary Adequacy: ∀ τ (e : CExp τ) d, SemExp e tt == Val d → ∃ v, e ⇓ v.
```

4 Recursive Domain Equations

We now outline our formalization of the solution of mixed-variance recursive domain equations, such as arise in modelling untyped higher-order languages, languages with higher-typed store or languages with general recursive types.

The basic technology for solving domain equations is Scott's inverse limit construction, our formalization of which follows an approach due to Freyd [11, 12] and Pitts [23]. A key idea is to separate the positive and negative occurences, specifying recursive domains as fixed points of locally continuous bi-functors $F : \mathbb{CPO}^{op} \times \mathbb{CPO} \to \mathbb{CPO}$, i.e. domains D such that such that $F(D, D) \simeq D$.

The type of mixed variance locally-continuous bifunctors on $\mathbb{CPO}$ is defined as the type of records comprising an action on pairs of objects (ob), a continuous action on pairs of morphisms (mor), contravariant in the first argument and covariant in the second, together with proofs that mor respects both composition ($morph\_comp$) and identities ($morph\_id$):

```
Record BiFunctor := mk_functor
  { ob : cpo → cpo → cpo;
    mor: ∀ (A B C D : cpo), (B ⇒c A) × (C ⇒c D) ⇒c (ob A C ⇒c ob B D) ;
    morph_comp: ∀ A B C D E F f g h k,
                  morB E D F (f, g) ∘ morA B C D (h, k)
                  == mor_ _ _ _ (h ∘f, g ∘k) ;
    morph_id : ∀ A B, mor_ _ _ _ (IDA , IDB) == ID_
```

We single out the *strict* bifunctors, taking pointed cpos to pointed cpos:

```
Definition FStrict : BiFunctor → Type :=
  fun BF ⇒ ∀ D E, PointedD → PointedE → (Pointed(ob BF D E)).
```

We build interesting bifunctors from a few primitive ones in a combinatory style. If $D : cpo$ then $BiConst\,D : BiFunctor$ on objects is constantly D and on morphisms is constantly $ID\,D$. This is strict if D is pointed. $BiArrow : BiFunctor$ on objects takes (D, E) to $D \Rightarrow_c E$ with the action on morphisms given by conjugation. This is strict. If $F : BiFunctor$ then $BiLift\,F : BiFunctor$ on objects takes (D, E) to $(ob\,F\,(D, E))_\perp$ and on morphisms composes $mor\,F$ with the morphism part of the lift functor. This is always strict. If $F_1\ F_2 : BiFunctor$ then $BiPair\,F_1\,F_2 : BiFunctor$ on objects takes (D, E) to $(ob\,F_1\,(D, E)) \times (ob\,F_2\,(D, E))$ with the evident action on morphisms. This is strict if both F_1 and F_2 are. The definition of $BiSum\,F_1\,F_2 : BiFunctor$ is similar, though this is not generally strict as our coproduct is a separated sum.

A pair $(f : D \to_c E, g : E \to_c D)$ is an embedding-projection (e-p) pair if $g \circ f == id_D$ and $f \circ g \sqsubseteq id_E$. If F is a bifunctor and (f, g) an e-p pair, then $(mor\,F\,(g, f),\ mor\,F\,(f, g))$ is an e-p pair.

Now let $F : BiFunctor$ and $FS : FStrict\,F$. We define $\langle D_i \rangle$ to be the sequence of *cpo*s defined by $D_0 = \mathbf{1}$ and $D_{n+1} = ob\,F\,(D_n, D_n)$. We then define a sequence of e-p pairs:

$$e_0 = \bot : D_0 \to_c D_1 \qquad p_0 = \bot : D_1 \to_c D_0$$
$$e_{n+1} = mor\,F\,(p_n, e_n) : D_{n+1} \to_c D_{n+2} \quad p_{n+1} = mor\,F\,(e_n, p_n) : D_{n+2} \to_c D_{n+1}.$$

Let $\pi_i : \Pi_j D_j \to_c D_i$ be the projections from the product of all the D_js. The predicate $P : \Pi_j D_j \to Prop$ defined by $P\,d := \forall i,\ \pi_i\ d == p_n(\pi_{i+1}\ d)$ is admissible, so we can define the sub-cpo D_∞ to be $\{d \mid P\,d\}$ with order and lubs inherited from the indexed product. D_∞ will be the *cpo* we seek, so we now need to construct the required isomorphism.

Define $t_n : D_n \to D_\infty$ to be the map that for $i < n$ projects D_n to D_i via $p_i \circ \cdots \circ p_{n-1}$ and for $i > n$ embeds D_n in D_i via $e_n \circ \cdots \circ e_{i-1}$. Then $mor\,F\,(t_i, \pi_i) : ob\,F\,(D_\infty, D_\infty) \to_c ob\,F\,(D_i, D_i) = D_{i+1}$, so $t_{i+1} \circ mor\,F\,(t_i, \pi_i) : ob\,F\,(D_\infty, D_\infty) \to_c D_\infty$, and $mor\,F\,(\pi_i, t_i) \circ \pi_{1+1} : D_\infty \to_c ob\,F\,(D_\infty, D_\infty)$. We then define

$$UNROLL := \bigsqcup_i (mor\,F\,(\pi_i, t_i) \circ \pi_{i+1}) : D_\infty \to_c ob\,F\,(D_\infty, D_\infty)$$
$$ROLL := \bigsqcup_i (t_{i+1} \circ mor\,F\,(t_i, \pi_i)) : ob\,F\,(D_\infty, D_\infty) \to_c D_\infty$$

Some calculation shows that $ROLL \circ UNROLL == ID\,D_\infty$ and $UNROLL \circ ROLL == ID(ob\,F\,(D_\infty, D_\infty))$, so we have constructed the desired isomorphism.

In order to do anything useful with recursively defined domains, we really need some general reasoning principles that allow us to avoid unpicking all the complex details of the construction above every time we want to prove something. One 'partially' abstract interface to the construction reveals that D_∞ comes equipped with a chain of retractions $\rho_i : D_\infty \to_c D_\infty$ such that $\bigsqcup_i \rho_i == IDD_\infty$; concretely, ρ_i can be taken to be $t_i \circ \pi_i$. A more abstract and useful principle is given by Pitts's [23] characterization of the solution as a *minimal invariant*, which is how we will establish the existence of a recursively defined logical relation in Section 5.1. Let $\delta : (D_\infty \Rightarrow_c D_\infty) \to_c (D_\infty \Rightarrow_c D_\infty)$ be given by

$$\delta\ e\ = ROLL \circ mor\,F\,(e, e) \circ UNROLL$$

The minimal invariance property is then the assertion that IDD_∞ is equal to $fix(\delta)$, which we prove via a pointwise comparison of the chain of retractions whose lub we know to be the identity function with the chain whose lub gives the least fixed point of δ.

5 A Uni-Typed Lambda Calculus

We now apply the technology of the previous section to formalize the denotational semantics of an uni-typed (untyped) CBV lambda calculus with constants.

This time the values are variables, numeric constants, and λ abstractions; expressions are again in ANF with *LET* and *VAL* constructs, together with function application, numeric operations, and a zero-test conditional. For binding, we use de Bruijn indices and separate well-formedness judgments *ETyping* and *VTyping*. The evaluation relation is as follows:

```
Inductive Evaluation : Exp → Value → Type :=
| e_Value : ∀ v, VAL v ⇓ v
| e_App : ∀ e v₂ v, substExp [v₂] e ⇓ v → (APP (LAMBDA e) v₂) ⇓ v
| e_Let : ∀ e₁ v₁ e₂ v₂, e₁ ⇓ v₁ → substExp [v₁] e₂ ⇓ v₂ → (LET e₁ e₂) ⇓ v₂
| e_Ifz1 : ∀ e₁ e₂ v₁, e₁ ⇓ v₁ → IFZ (NUM O) e₁ e₂ ⇓ v₁
| e_Ifz2 : ∀ e₁ e₂ v₂ n, e₂ ⇓ v₂ → IFZ (NUM (S n)) e₁ e₂ ⇓ v₂
| e_Op : ∀ op v₁ n1 v₂ n2, OP op (NUM n1) (NUM n2) ⇓ NUM (op n1 n2)
where "e '⇓' v" := (Evaluation e v).
Inductive Converges e : Prop := e_Conv : ∀ v (_ : Evaluation e v), Converges e.
Notation "e '⇓'" := (Converges e).
```

Note that *Evaluation* is here in `Type` rather than `Prop`, which is a knock-on effect of separating the definition of terms from that of well-formedness.[5] We plan instead to make untyped terms well-scoped by construction, using the same techniques as we did for the typed language in Section 3.

5.1 Semantic Model

We interpret the unityped language in a solution for the recursive domain equation $D \simeq (nat + (D \to_c D))_\perp$, following the intuition that a computation either diverges or produces a value which is a number or a function. This is not the 'tightest' domain equation one could use for CBV: one could make function space strict, or equivalently make the argument of the function space be a domain of values rather than computations. But this equation still gives an adequate model. The construction in Coq is an instantiation of results from the previous section. First we build the strict bifunctor $F(D, E) = (nat + (D \to_c E))_\perp$:

```
Definition FS := BiLift_strict (BiSum (BiConst (Discrete nat)) BiArrow).
```

And then we construct the solution, defining domains D_∞ for computations and V_∞ for values:

```
Definition D∞ := D∞ FS.
Definition V∞ := Dsum (Discrete nat) (D∞ →c D∞).
Definition Roll : (V∞)⊥ →c D∞ := ROLL FS.
Definition Unroll : D∞ →c (V∞)⊥ := UNROLL FS.
Definition UR_iso : Unroll ∘ Roll == ID _ := DIso_ur FS.
Definition RU_iso : Roll ∘ Unroll == ID _ := DIso_ru FS.
```

For environments we define the n-ary product of V_∞ and projection function.

```
Fixpoint SemEnv n : cpo := match n with O ⇒ 1| S n ⇒ SemEnv n × V∞ end.
```

[5] We induce over evaluations to construct well-formedness derivations when showing well-formedness preservation, and well-formedness derivations are themselves in `Type` so that we can use them to inductively define the denotational semantics.

```
Fixpoint projenv (m n : nat) : (m < n) → SemEnv n →c V∞ :=
match m, n with
| m, O ⇒ fun inconsistent ⇒ match (lt_n_O m inconsistent) with end
| O, S n ⇒ fun _ ⇒ π2
| S m, S n ⇒ fun h ⇒ projenv (lt_S_n _ _ h) ∘ π1 end.
```

We define a lifting operator $Dlift : (V_\infty \to_c D_\infty) \to_c D_\infty \to_c D_\infty$ and an operator $Dapp : V_\infty \times V_\infty \to_c (V_\infty)_\perp$ that applies the first component of a pair to the second, returning $\perp$ in the when the first component is not a function. (Coproducts are introduced with *INL* and *INR*, and eliminated with $[\cdot, \cdot]$.)

```
Definition Dlift : (V∞ →c D∞) →c D∞ →c D∞ :=
    curry (Roll ∘ ev ∘⟨kleisli ∘ (Unroll ∘ −) ∘ π1, Unroll ∘ π2⟩).
Definition Dapp : V∞ × V∞ →c (V∞)⊥ :=
    ev ∘ ⟨[⊥ : Discrete nat →c D∞ →c (V∞)⊥, (Unroll ∘ −)] ∘ π1, Roll ∘ η ∘ π2⟩.
```

We can then define the semantics of the unityped language:

```
Fixpoint SemVal v n (vt : VTyping n v) : SemEnv n →c V∞ :=
match vt with
| TNUM n ⇒ INL _ _ ∘ (@K _ (Discrete nat) n)
| TVAR m nthm ⇒ projenv nthm
| TLAMBDA t b ⇒ INR _ _ ∘ Dlift ∘ curry (Roll ∘ SemExp b)
end with SemExp e n (et : ETyping n e) : SemEnv n →c (V∞)⊥ :=
match et with
| TAPP _ _ v1 v2 ⇒ Dapp ∘ ⟨SemVal v1, SemVal v2⟩
| TVAL _ v ⇒ η ∘ SemVal v
| TLET _ _ e1 e2 ⇒ ev ∘ ⟨curry(Kleislir(SemExp e2)), SemExp e1⟩
| TOP op _ _ v1 v2 ⇒
        uncurry(Operator2 (η ∘ INL _ _ ∘ uncurry(SimpleOp2 op))) ∘
          ⟨[η, ⊥ : (D∞ →c D∞) →c (Discrete nat)⊥] ∘ SemVal v1,
           [η, ⊥ : (D∞ →c D∞) →c (Discrete nat)⊥] ∘ SemVal v2⟩
| TIFZ _ _ _ v e1 e2 ⇒ ev ∘
         ⟨[[K _ (SemExp e1), K _ (SemExp e2)] ∘ zeroCase,
           ⊥ : (D∞ →c D∞) →c SemEnv n →c (V∞)⊥] ∘ (SemVal v), ID _⟩ end.
```

5.2 Soundness and Adequacy

As with the typed language, the proof of soundness makes use of a substitution lemma, and in addition uses the isomorphism of the domain D_∞ in the case for *APP*. The proof then proceeds by induction, using equational reasoning to show that evaluation preserves semantics:

```
Lemma Soundness: ∀ e v (et : ETyping 0 e) (vt : VTyping 0 v),
            e ⇓ v → SemExp et == η ∘ SemVal vt.
```

We again use a logical relation between syntax and semantics to prove adequacy, but now cannot define the relation by induction on types. Instead we have a recursive specification of a logical relation over our recursively defined domain, but it is not at all clear that such a relation exists: because of the mixed variance of the function space, the operator on relations whose fixpoint we seek is not monotone. Following Pitts [23], however, we again use the technique of

separating positive and negative occurrences, defining a monotone operator in the complete lattice of *pairs* of relations, with the superset order in the first component and the subset order in the second. A fixed point of that operator is then constructed by Knaster-Tarski.

We first define a notion of *admissibility* on ==-respecting relations between elements of our domain of values V_∞ and closed syntactic values in *Value* and show that this is closed under intersection, so admissible relations form a complete lattice.

We then define a relational action corresponding to the bifunctor used in defining our recursive domain. This action, $RelV$, maps a pair of relations R, S on $(V_\infty)_\perp \times Value$ to a new relation that relates $(inl\ m)$ to $(NUM\ m)$ for all $m : nat$, and relates $(inr\ f)$ to $(LAMBDA\ e)$ just when $f : D_\infty \to_c D_\infty$ is strict and satisfies the 'logical' property

$$\begin{array}{rl} \forall (d, v) \in R, \forall d', Unroll(f(Roll(Val\ d))) == Val\ d' \\ \to \quad \exists v', substExp\ v\ e \Downarrow v' \wedge (d', v') \in S \end{array}$$

We then show that $RelV$ maps admissible relations to admissible relations and is contravariant in its first argument and covariant in its second. Hence the function $\lambda R : RelAdm^{op}.\ \lambda S : RelAdm.\ (RelV\ S\ R, RelV\ R\ S)$ is monotone on the complete lattice $RelAdm^{op} \times RelAdm$. Thus it has a least fixed point (Δ^-, Δ^+). By applying the minimal invariant property from the previous section, we prove that in fact $\Delta^- == \Delta^+$, so we have found a fixed point, Δ of $RelV$, which is the logical relation we need to prove adequacy.

We extend Δ to Δ_e, a relation on $(V_\infty)_\perp \times Exp$, by $(d, e) \in \Delta_e$ if and only if for all d' if $d == Val\ d'$ then there exists a value v and a derivation $e \Downarrow v$ such that $(d', v) \in \Delta$.

The fundamental theorem for this relation is that for any environment *env*, derivations of $VTyping\ n\ v$ and $ETyping\ n\ e$, and any list *vs* of n closed values such that $nth\_error\ vs\ i = Some\ v$ implies $(projenv\ i\ env, v) \in \Delta$ for all i and v, $(SemVal\ v\ env,\ substVal\ vs\ v) \in \Delta$ and $(SemExp\ e\ env,\ substExp\ vs\ e) \in \Delta_e$.

Adequacy is then a corollary of the fundamental theorem:

`Theorem` *Adequacy*: $\forall\ e\ (te : ETyping\ 0\ e)\ d, SemExp\ te\ tt == Val\ d \to e \Downarrow$.

6 Discussion

As we noted in the introduction, there have been many mechanized treatments of different aspects of domain theory and denotational semantics. One rough division of this previous work is between axiomatic approaches and those in which definitions and proofs of basic results about cpos, continuous functions and so on are made explicitly with the prover's logic. LCF falls into the first category, as does Reus's work on synthetic domain theory in LEGO [25]. In the second category, HOLCF, originally due to Regensburger [24] and later reworked by Müller et al [18], uses Isabelle's axiomatic type class mechanism to define and prove properties of domain-theoretic structures within higher order logic.

HOLCPO [2, 4] was an extension of HOL with similar goals, and basic definitions have also been formalized in PVS [7]. Coq's library includes a formalization by Kahn of some general theory of dcpos [14].

HOLCF is probably the most developed of these systems, and has been used to prove interesting results [19, 26], but working in a richer dependent type theory gives us some advantages. We can express the semantics of a typed language as a dependently typed map from syntax to semantics, rather than only being able to do shallow embeddings – this is clearly necessary if one wishes to prove theorems like adequacy or do compiler correctness. Secondly, it seems one really needs dependent types to work conveniently with monads and logical relations, or to formalize the inverse limit construction.[6]

The constructive nature of our formalization and the coinductive treatment of lifting has both benefits and drawbacks. On the minus side, some of the proofs and constructions are much more complex than they would be classically and one does sometimes have to pay attention to which of two classically-equivalent forms of definition one works with. Worse, some constructions do not seem to be possible, such as the smash product of pointed domains; not being able to define $\otimes$ was one motivation for moving from Paulin-Mohring's pointed cpos to our unpointed ones. One benefit that we have not yet seriously investigated, however, is that it is possible to extract actual executable code from the denotational semantics. Indeed, the lift monad can be seen as a kind of syntax-free operational semantics, not entirely unlike game semantics; this perspective, and possible connections with step-indexing, seem to merit further study.

The Coq development is of a reasonable size. The domain theory library, including the theory of recursive domain equations, is around 7000 lines. The formalization of the typed language and its soundness and adequacy proofs are around 1700 lines and the untyped language takes around 2500. Although all the theorems go through (with no axioms), we have to admit that the development is currently rather 'rough'. Nevertheless, we have already used it as the basis of a non-trivial formalization of some new research [8] and our intention is to develop the formalization into something that is more widely useful. Apart from general polishing, we plan to abstract some of the structure of our category of domains to make it convenient to work simultaneously with different categories, including categories of algebras. We would also like to provide better support for 'diagrammatic' rewriting in monoidal (multi)categories. It is convenient to use Setoid rewriting for pointfree equational reasoning, direct translating the normal categorical commuting diagrams. But dealing with all the structural morphisms is still awkward, and it should be possible to support something more like the diagrammatic proofs one can do with 'string diagrams' [13].

[6] Agerholm [3] formalized the construction of a model of the untyped lambda calculus using HOL-ST, a version of HOL that supports ZF-like set theory; this is elegant but HOL-ST is not widely used and no denotational semantics seems to have been done with the model. Petersen [21] formalized a reflexive cpo based on $P\omega$ in HOL, though this also appears not to have been developed far enough to be useful.

References

1. R. Adams. Formalized metatheory with terms represented by an indexed family of types. In *Types for Proofs and Programs*, volume 3839 of *LNCS*, 2006.
2. S. Agerholm. Domain theory in HOL. In *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*, 1994.
3. S. Agerholm. Formalizing a model of the lambda calculus in HOL-ST. Technical Report 354, University of Cambridge Computer Laboratory, 1994.
4. S. Agerholm. LCF examples in HOL. *The Computer Journal*, 38(2), 1995.
5. T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, volume 1683 of *LNCS*, 1999.
6. P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. In *Mathematics of Program Construction*, volume 4014 of *LNCS*, 2006.
7. F. Bartels, A. Dold, H. Pfeifer, F. W. Von Henke, and H. Rueß. Formalizing fixed-point theory in PVS. Technical report, Universität Ulm, 1996.
8. N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ACM International Conference on Functional Programming*, 2009.
9. V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1, 2005.
10. T. Coquand. Infinite objects in type theory. In *Types for Proofs and Programs*, volume 806 of *LNCS*, 1993.
11. P. Freyd. Recursive types reduced to inductive types. In *IEEE Symposium on Logic in Computer Science*, 1990.
12. P. Freyd. Remarks on algebraically compact categories. In *Applications of Categories in Computer Science*, volume 177 of *LMS Lecture Notes*, 1992.
13. A. Joyal and R. Street. The geometry of tensor calculus. *Adv. in Math. 88*, 1991.
14. G. Kahn. Elements of domain theory. In the Coq users' contributions library, 1993.
15. C. McBride. Type-preserving renaming and substitution. Unpublished draft.
16. R. Milner. Logic for computable functions: Description of a machine implementation. Technical Report STAN-CS-72-288, Stanford University, 1972.
17. E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.
18. O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *J. Functional Programming*, 9, 1999.
19. T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10, 1998.
20. C. Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In *From Semantics to Computer Science. Essays in Honour of G Kahn*. 2009.
21. K. D. Petersen. Graph model of LAMBDA in higher order logic. In *Higher-Order Logic Users Group Workshop*, volume 780 of *LNCS*, 1993.
22. A. M. Pitts. Computational adequacy via 'mixed' inductive definitions. In *Mathematical Foundations of Programming Semantics*, volume 802 of *LNCS*, 1994.
23. A. M. Pitts. Relational properties of domains. *Inf. Comput.*, 127, 1996.
24. F. Regensburger. HOLCF: Higher order logic of computable functions. In *Theorem Proving in Higher Order Logics*, volume 971 of *LNCS*, 1995.
25. B. Reus. Formalizing a variant of synthetic domain theory. *J. Automated Reasoning*, 23, 1999.
26. C. Varming and L. Birkedal. Higher-order separation logic in Isabelle/HOLCF. In *Mathematical Foundations of Programming Semantics*, 2008.