Under consideration for publication in Math. Struct. in Comp. Science

Formalizing Domains, Ultrametric Spaces and Semantics of Programming Languages

NICK BENTON[1], LARS BIRKEDAL[2], ANDREW KENNEDY[1]

and CARSTEN VARMING[3†]

[1] *Microsoft Research, Cambridge, UK*
[2] *IT University of Copenhagen, Denmark*
[3] *Carnegie Mellon University, Pittsburgh, USA*

Received 12 July 2010

We describe a Coq formalization of constructive ω-cpos, ultrametric spaces and ultrametric-enriched categories, up to and including the inverse-limit construction of solutions to mixed-variance recursive equations in both categories enriched over ω-cppos and categories enriched over ultrametric spaces.
We show how these mathematical structures may be used in formalizing semantics for three representative programming languages. Specifically, we give operational and denotational semantics for both a simply-typed CBV language with recursion and an untyped CBV language, establishing soundness and adequacy results in each case, and then use a Kripke logical relation over a recursively-defined metric space of worlds to give an interpretation of types over a step-counting operational semantics for a language with recursive types and general references.

1. Introduction

The use of proof assistants in formalizing language metatheory and implementing certified tools has grown enormously over the last decade, and is now a major trend in programming language research. Most current work on mechanizing language definitions and safety proofs, certified compilation, proof carrying code, and so on uses fairly elementary operational methods, such as syntactic type soundness and rather intensional simulation relations for compiler correctness. Indeed, one of the motivations for the recent development of some of the most exciting new operational methods, notably step-indexing and its refinements, has been a desire for techniques that are more amenable to machine formalization than is domain theory, which has often been viewed as too complex to work with in a proof assistant.

Whilst the authors are enthusiastic users of operational methods, we have also all – when working on paper – made considerable use of denotational semantics, for example in

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

modelling higher-order languages with references. When it is applicable, denotational semantics tends to offer stronger theorems, more reusable results and important intuitions (not least in interpreting 'functions' as functions); we are loath to abandon it as soon as we start to mechanize our work. Furthermore, operational techniques have themselves become decidedly non-trivial and started to involve general constructions, such as the indirection theory of Hobor et al. [2010], which are most usefully understood and generalized by being explicit about the semantic categories in which they really take place. Indeed, when modelling programming languages with higher-order references the whole space of semantic types is naturally defined by a solution to a recursive domain equation in a category of complete bounded ultrametric spaces. This holds irrespective of whether one defines the semantics of types over an operational or a denotational semantics of the underlying programming language [Birkedal et al., 2010a]. The present paper describes a formalization of many of the denotational techniques we have found useful in practice, showing that they are indeed amenable to mechanization.

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 [1972b] LCF, 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 gone far enough to be applicable to formalizing even what one might cover in an undergraduate course on denotational semantics.

We will describe a Coq formalization of ω-cpos, bisected complete ultrametric spaces, the denotational semantics of a typed and untyped versions of a simple functional language, and a semantics of types for a language with recursive types and a higher order store based on operational semantics. All this goes considerably further than previous work on mechanizing semantics; another paper [Benton and Hur, 2009] describes a non-trivial compiler correctness theorem that was formalized and proved using one of the models presented here. We have also used the framework to formalize full abstraction results for a trace-based semantics of cooperative threads by Abadi and Plotkin [2009].

Section 2 introduces the structuring mechanisms used in the development. The first is a formalization of some basic category theory, allowing us to abstract much of the theory relating to constructions, such as products, that are shared by several different kinds of mathematical structure. The second is the 'packed classes' pattern [Garillot et al., 2009] for defining and working with hierarchies of mathematical structures. Section 3 describes the formalization of ω-cpos, which is based on an earlier library for constructive pointed ω-cpos and continuous functions written by Paulin-Mohring [2009] as a basis for a semantics of Kahn networks, and of probabilistic programs [Audebaud and Paulin-Mohring, 2006]. In Section 4, we 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 classic results seem not to have been previously mechanized for a higher-order language.

Section 5 is about solving recursive domain equations. We formalize Scott's inverse limit

construction along the lines of work by Freyd [1990, 1992] and Pitts [1994, 1996]. 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. Our solutions can be found in any $\mathcal{O}$-category, i.e., any cppo-enriched category, as presented by Smyth and Plotkin [1982]. In Section 6, we define the semantics of an untyped call-by-value language using a particular recursive domain found in the Kleisli category for our lift monad, and use the associated reasoning principles to again establish soundness and adequacy theorems.

Section 7 concerns the formalization of complete bounded ultra-metric spaces (CBUlt), and how to find solutions to recursive domain equations in general $\mathcal{M}$-categories, i.e., categories enriched over CBUlt. Section 8 continues with a denotational interpretation of the types of a lambda calculus with recursive types and higher order store. The semantics is found in the category of pre-ordered complete bounded ultra-metric spaces (PreCBUlt), where we need to solve a recursive domain equation to model the space of semantic types. We conclude this section with a sound interpretation of the language.

An earlier version of some of this work was published as a conference paper [Benton et al., 2009]. The fragments of Coq appearing in the paper have been typeset from the compilable sources, but are far from self-contained. The complete Coq development, which builds under Coq 8.2pl1 plus Ssreflect, is available from the authors' homepages.

2. Packaging Structures and Categories

Many different mathematical structures are used in programming language semantics, some of which are built on others (e.g. cpos being a further specialization of partial orders) and many of which support similar constructions (e.g. cartesian products). An earlier version of our formalization of domain theory used Coq's dependent records to make independent definitions of each structure, and of each of the corresponding constructions and lemmas, together some fairly ad-hoc use of coercions. This straightforward approach worked, but did suffer from a proliferation of notations (e.g. for the many different kinds of structure-preserving map), and the presence of many rather similar definitions and lemmas (which it was also hard to name consistently). Whilst extending the formalization to cover ultrametric spaces we have also refactored it, using two techniques in particular to manage complexity. The first is a design pattern, due to Garillot et al. [2009], for mixin-style packaging of mathematical structures. The second is the formalization of a certain amount of category theory, enabling us to work with slightly more abstract and reusable definitions, lemmas and notations.

2.1. *Setoids*

All the structures with which we will be working have a notion of equality, which will generally not be Coq's intensional '='. We thus define a top-level structure of setoids: types equipped with an equivalence relation [Barthe et al., 2003]:

```
Module Setoid.
  Definition axiom (T:Type) (e:T → T → Prop) := equiv _ e.
```

```
  Record mixin_of T := Mixin
  { set_eq : T → T → Prop; set_equiv : axiom set_eq }.
  Notation class_of := mixin_of (only parsing).
  Structure type := Pack {sort :> Type; _:class_of sort; _:Type}.
  Definition class cT := let: Pack _ c _ := cT return class_of cT in c.
  Definition unpack K (k : ∀ T (c : class_of T), K T c) cT :=
    let: Pack T c _ := cT return K _ (class cT) in k _ c.
  Definition pack T c := @Pack T c T.
End Setoid.
Notation setoidType := Setoid.type.
Notation SetoidMixin := Setoid.Mixin.
Notation SetoidType := Setoid.pack.
Definition tset_eq := fun T ⇒ Setoid.set_eq (Setoid.class T).
Infix "≡ " := tset_eq (at level 70).
Lemma tset_refl (T:setoidType) (x:T) : x ≡ x.
Lemma tset_trans (T:setoidType) (x y z:T) : x ≡ y → y ≡ z → x ≡ z.
Lemma tset_sym (T:setoidType) (x y:T) : x ≡ y → y ≡ x.
Add Parametric Relation (T:setoidType) : T (@tset_eq T)
 reflexivity proved by (@tset_refl T) symmetry proved by (@tset_sym T)
 transitivity proved by (@tset_trans T) as tset_eqrel.
```

We make pervasive use of the packed classes pattern of Garillot et al. [2009]; this looks somewhat prolix at first sight, but is a very flexible and uniform way of constructing and inferring hierarchies of structures. Coq modules are used as namespaces, allowing standardized internal naming for the different aspects of each structure. Then `Setoid.axiom e` says that `e` is an equivalence relation, the `T`-parameterized `Setoid.mixin` bundles a notion of equality on `T` with a proof that it satisfies the axiom, whilst `Setoid.type` packs a particular carrier type together with the corresponding mixin. We will later see how this two-stage separation of the `Packed` type and the `class` is used to build hierarchical structures, but for now note that the coercion `sort :> Type` means that if `T : setoidType` then `x : T` is shorthand for `x : sort T`. We also register equalities on `setoidTypes` as relations with respect to which we wish to be able to rewrite, using Sozeau's [2009] recent generalized re-implementation of Coq's setoid rewriting framework.

2.2. *Categories*

Categories are another top-level structure. The mixin is parameterized by a type `T` for the objects and a dependent type `T → T → setoidType` for morphisms, and specifies a composition operation (`tcomp`) and identities (`tid`) together with proofs that these satisfy the usual axioms for categories, and that composition respects equality on morphisms:

```
Module Category.
  Section Axioms.
    Variable Ob:Type.
    Variable Morph : Ob → Ob → setoidType.
    Variable tcomp : ∀ T0 T1 T2, Morph T1 T2 → Morph T0 T1 → Morph T0 T2.
    Variable tid : ∀ T0, Morph T0 T0.
```

```
    Definition tid_left := ∀ T0 T1 (f:Morph T0 T1), tcomp (tid T1) f ≡ f.
    Definition tid_right := ∀ T0 T1 (f:Morph T0 T1), tcomp f (tid T0) ≡ f.
    Definition tcomp_assoc := ∀ T0 T1 T2 T3 (f:Morph T2 T3) (g:Morph T1 T2)
             (h:Morph T0 T1), (tcomp f (tcomp g h)) ≡ (tcomp (tcomp f g) h).
    Definition tcomp_respect := ∀ T0 T1 T2 (f f' : Morph T1 T2)
          (g g' : Morph T0 T1), f ≡ f' → g ≡ g' → tcomp f g ≡ tcomp f' g'.
  End Axioms.
  Definition axiom O M c i :=
    @tid_left O M c i ∧ tid_right c i ∧ tcomp_assoc c ∧ tcomp_respect c.
  Record mixin_of T (Morph : T → T → setoidType) := Mixin
  { tcomp : ∀ T0 T1 T2, Morph T1 T2 → Morph T0 T1 → Morph T0 T2;
    tid : ∀ T0, Morph T0 T0;
    tcategory : axiom tcomp tid }.
  Notation class_of := mixin_of.
  Structure cat := Pack {object :> Type;
    morph :> object → object → setoidType ; _:class_of morph; _:Type}.
  Definition class cT := let: Pack _ _ c _ := cT return class_of cT in c.
  Definition unpack (K:∀ O (M:O → O → setoidType) (c:class_of M), Type)
    (k : ∀ O (M: O → O → setoidType) (c:class_of M), K _ _ c) (cT:cat) :=
    let: Pack _ M c _ := cT return @K _ _ (class cT) in k _ _ c.
  Definition pack T M c := @Pack T M c T.
  Definition comp (cT:cat) : ∀ (A B C:cT),
              morph B C → morph A B → morph A C := tcomp (class cT).
  Definition id (cT:cat) : ∀ (A:cT), morph A A := tid (class cT).
End Category.
Notation catType := Category.cat.
Notation morph := Category.morph.
Notation object := Category.object.
Infix "⟶ " := Category.morph (at level 55, right associativity)
Infix "∘ " := Category.comp (at level 35)
Notation Id := Category.id.
Add Parametric Morphism (C:catType) X Y Z : (@Category.comp C X Y Z)
  with signature (@tset_eq (C Y Z)) ==>(@tset_eq (C X Y)) ==>(@tset_eq (C X Z))
  as comp_eq_compat.
Lemma comp_assoc (C:catType) (W X Y Z : C) (f:W ⟶ X) (g : X ⟶ Y)
    (h : Y ⟶ Z) : h ∘ (g ∘ f) ≡ h ∘ g ∘ f.
Lemma comp_idR (C:catType) (X Y : C) (f : X ⟶ Y) : f ∘ Id ≡ f.
```

Again, there is complexity associated with the packed classes pattern, though this is essentially the same boilerplate as for `Setoid`. That composition respects the setoid equality on morphisms is registered with the rewriting machinery, so the in-context equational rewrites used in traditional 'diagram chasing' proofs translate directly into Coq. The external interface is provided by the notations, such as ∘ , and several lemmas, such as `comp_assoc`, defined just after the module.

We next formalize categories with more specialized structure, including finite products and coproducts, exponentials, limits and colimits. The following extract from the definition of categories with products demonstrates how the packed classes pattern works:

```
Module CatProduct.
  Definition prod_diagram (C : catType) (A B P : C) (π1 : C P A) (π2 : C P B)
                          (X : C) (f : C X A) (g : C X B) (h : C X P) :=
                        π1∘ h ≡ f ∧ π2∘ h ≡ g.
  Definition axiom (C:catType) (prod : C → C → C) (π1 : ∀ A B, C (prod A B) A)
     (π2 : ∀ A B, C (prod A B) B) (h: ∀ A B Z, C Z A → C Z B → C Z (prod A B)) :=
    ∀ A B X f g,
        @prod_diagram C A B (prod A B) (π1 _ _) (π2 _ _) X f g (h A B X f g) ∧
          ∀ m, prod_diagram (π1 _ _) (π2 _ _) f g m → m ≡ (h A B X f g).
  Record mixin_of (C:catType) := Mixin
  { prod : C → C → C;
    π1: ∀ A B, C (prod A B) A;
    π2: ∀ A B, C (prod A B) B;
    prod_ex : ∀ A B Z, C Z A → C Z B → C Z (prod A B); _ : axiom π1π2prod_ex}.
  Record class_of T (M:T → T → setoidType) : Type :=
    Class { base :> Category.class_of M ; ext :> mixin_of (Category.Pack base T)}.
  Structure cat := Pack {object :> Type;
        morph :> object → object → setoidType ; _ : class_of morph; _ : Type}.
  Definition class cT := let: Pack _ _ c _ := cT return class_of cT in c.
  Definition pack := let k T M c m := Pack (@Class T M c m) T in Category.unpack k.
  Coercion catType (cT:cat) := Category.pack (class cT).
End CatProduct.

Notation prodCat := CatProduct.cat.
Canonical Structure CatProduct.catType.
Definition prod (C:prodCat) (A B:C) : C :=
                (CatProduct.prod (CatProduct.class C) A B).
Definition π1(C:prodCat) (A B:C) : morph (prod A B) A :=
                (CatProduct.π1 (CatProduct.class C) A B).
Definition π2(C:prodCat) (A B:C) : morph (prod A B) B :=
                (CatProduct.π2 (CatProduct.class C) A B).
Definition prod_fun (C:prodCat) (Z A B:C) (f:C Z A) (g:C Z B) :
  morph Z (prod A B) := (CatProduct.prod_ex (CatProduct.class C) f g).
Infix "*" := prod
Notation "'⟨' f , g '⟩'" := (prod_fun f g) (at level 30)
Notation "f '×' g" := (prod_fun (f ∘ π1) (g ∘ π2)) (at level 30)
Lemma prod_fun_fst (C:prodCat) (X Y Z:C) (f:Z ⟶ Y) (g:Z ⟶ X) :
                                        π1∘ ⟨f , g⟩ ≡ f.
Lemma prod_fun_snd (C:prodCat) (X Y Z:C) (f:Z ⟶ Y) (g:Z ⟶ X) :
                                        π2∘ ⟨f , g⟩ ≡ g.
Lemma prod_unique (C:prodCat) (X Y Z : C) (h h':Z ⟶ X * Y) :
  π1∘ h ≡ π1∘ h' → π2∘ h ≡ π2∘ h' → h ≡ h'.
Add Parametric Morphism (C:prodCat) X Y Z : (@prod_fun C X Y Z)
  with signature (@tset_eq (C X Y)) ==>(@tset_eq (C X Z)) ==>
                (@tset_eq (C X (Y * Z))) as prod_fun_eq_compat.
Lemma prod_map_prod_fun (C:prodCat) (X Y Z W A : C) (f:X ⟶ Y) (g: Z ⟶ W)
              (h:A ⟶ _) (k:A ⟶ _) : f ×g ∘ ⟨h,k⟩ ≡ ⟨f ∘ h, g ∘ k⟩.
```

```
move ⇒ C X Y Z W A f g h k. apply prod_unique; rewrite comp_assoc.
- do 2 rewrite prod_fun_fst. rewrite <- comp_assoc. by rewrite prod_fun_fst.
- do 2 rewrite prod_fun_snd. rewrite <- comp_assoc. by rewrite prod_fun_snd.
Qed.
```

The `class` component of each structure is a dictionary, in the sense of Haskell type classes. For categories with products, this comprises two sub-dictionaries: one (`base`) carrying the features of a category and one (`ext`) giving a product structure thereon. `CatProduct.pack` makes a category-with-products out of a category and a corresponding product mixin, using the CPS-style destructor `Category.unpack` to access the components of the category. The coercion `CatProduct.catType` allows categories-with-products to be coerced to categories; registering this as a `Canonical Structure` allows Coq to infer the intended structure when we use category operations (such as the notation for arrows, composition and equality) with `prodCat`s. If `C:prodCat` then `CatProduct.object C` is convertible with `Category.object (CatProduct.catType C)`, for example, and it is the canonical structure declaration that allows Coq to solve the (higher-order) unification problem `CatProduct.object C = Category.object ??`. The `prod_map_prod_fun` lemma demonstrates some of the payoff from setting up this packaging and rewriting machinery: the statement and proof are just as they would be on paper.

3. Basic Domain Theory

We now turn to the formalization of the concrete category of ω-cpos, building on that by Paulin-Mohring [2009]. Apart from the packaging, the main difference is that Paulin-Mohring treated *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.

The type of preorders, `ordType`, packs a carrier `T` with a mixin containing a binary relation `Ole` (written infix as $\sqsubseteq$) and a proof that `Ole` is reflexive and transitive. The symmetrisation of $\sqsubseteq$ is an equivalence relation, so any `ordType` is a `setoidType`. The packaging of preorder is set up, using `Coercion`s and `Canonical Structure`, to infer this setoid structure automatically when needed:

```
Module PreOrd.
  Definition axiom T (Ole : T → T → Prop) :=
    ∀ x , Ole x x ∧ ∀ y z, (Ole x y → Ole y z → Ole x z).
  Record mixin_of T := Mixin {Ole : T → T → Prop; _ : axiom Ole}.
  Notation class_of := mixin_of (only parsing).
  Lemma setAxiom T (c:mixin_of T):Setoid.axiom (fun x y ⇒ Ole c x y ∧ Ole c y x).
  Coercion base2 T (c:class_of T) : Setoid.class_of T := Setoid.Mixin (setAxiom c).
  Structure type := Pack {sort :> Type; _ : class_of sort; _ : Type}.
  Definition class cT := let: Pack _ c _ := cT return class_of cT in c.
  Coercion setoidType (cT:type) := Setoid.Pack (class cT) cT.
End PreOrd.
Notation ordType := PreOrd.type.
```

```
Notation OrdMixin := PreOrd.Mixin.
Notation OrdType := PreOrd.pack.
Canonical Structure PreOrd.setoidType.
```

That `Ole` is a partial order modulo $\equiv$ is registered with Coq's generalized rewriting machinery; by also establishing the sense in which the constructions that follow are morphisms with respect to these relations, we can easily perform (in)equational rewriting in proofs. We henceforth omit 'standard' parts of the packed class infrastructure (such as `type` and `class` above) in listing fragments.

We define a structure of monotone functions between partial orders:

```
Definition monotonic (O1 O2 : ordType) (f : O1→ O2) := ∀ x y, x ⊑ y → f x ⊑ f y.
Module FMon. Section fmon.
 Variable O1 O2 : ordType.
 Record mixin_of (f:O1 → O2) := Mixin { ext :> monotonic f}.
 Structure type : Type := Pack {sort :> O1 → O2; _ : class_of sort; _ : O1 → O2}.
End fmon. End FMon.
Notation fmono := FMon.type.
```

And this structure itself inherits an `ordType` structure from the codomain:

```
Lemma fmono_axiom (O1 O2:ordType) :
    PreOrd.axiom (fun f g:fmono O1 O2 ⇒ ∀ x, f x ⊑ g x).
Canonical Structure fmono_ordMixin (T T':ordType) := OrdMixin (@fmono_axiom T T').
Canonical Structure fmono_ordType T T' :=
                          Eval hnf in OrdType (fmono_ordMixin T T').
```

We define `ordCatType:catType` the category of pre-orders (`ordType`s) and monotone functions (`fmono O O'`) as the canonical category for `ordType`s.

Define `natO : ordType` by equipping the natural numbers with the usual 'vertical' order, $\leq$. If `O : ordType` and `c : natO` $\longrightarrow$ `O`, call `c` a *chain* in `O`.

```
Lemma natO_axiom : PreOrd.axiom (fun n m : nat ⇒ leq n m).
Canonical Structure natO_ordMixin := OrdMixin natO_axiom.
Canonical Structure natO_ordType := Eval hnf in OrdType (natO_ordMixin).
Notation natO := natO_ordType.
```

A complete partial order comprises a preorder `T`, a function $\bigsqcup$ computing least upper bounds of chains in `T`, and a proof that this *is* always both an upper bound and less than or equal to any other upper bound:

```
Module CPO.
Definition axiom (T:ordType) (⨆ : (natO ⟶ T) → T) :=
       ∀ (c:natO ⟶ T) x n, (c n ⊑ ⨆c) ∧ ((∀ n, c n ⊑ x) → ⨆c ⊑ x).
Record mixin_of (T:ordType) : Type := Mixin {⨆: (natO ⟶ T) → T; _ : axiom ⨆}.
Record class_of (T:Type) : Type :=
  Class {base :> PreOrd.class_of T; ext :> mixin_of (PreOrd.Pack base T) }.
Coercion ordType cT := PreOrd.Pack (class cT) cT.
Definition setoidType cT := Setoid.Pack (class cT) cT.
End CPO.
```

```
Notation cpoType := CPO.type.
Notation CpoMixin := CPO.Mixin.
Notation CpoType := CPO.pack.
Canonical Structure CPO.ordType.
Canonical Structure CPO.setoidType.
```

It is important to note that this definition of a complete partial order is constructive: we require least upper bounds of chains not merely to exist, but to be computable in Coq's logic of total functions.

A monotone function f between two `cpoType`s is *continuous* if it preserves (up to $\equiv$) least upper bounds. One direction is already a consequence of monotonicity, so we just have to specify the other. We package continuous functions, inheriting from monotone ones.

```
Definition continuous (D1 D2 : cpoType) (f : ordCatType D1 D2) :=
              ∀ c : natO ⟶ D1, f (⨆ c) ⊑ ⨆(f ∘ c).
Module FCont. Section fcont.
 Variable O1 O2 : cpoType.
 Record mixin_of (f:fmono O1 O2) := Mixin {cont :> continuous f }.
 Record class_of (f : O1 → O2) :=
          Class {base :> FMon.mixin_of f; ext :> mixin_of (FMon.Pack base f) }.
 Coercion fmono f : fmono O1 O2 := FMon.Pack (class f) f.
End fcont. End FCont.
Notation fcont := FCont.type.
Canonical Structure FCont.fmono.
```

From the continuous identity map, `cid`, and a proof that composition preserves continuity, we construct the category `cpoCatType` of ω-cpos and continuous maps:

```
Lemma cpoCatAxiom : Category.axiom ccomp cid.
Canonical Structure cpoCatMixin := CatMixin cpoCatAxiom.
Canonical Structure cpoCatType := Eval hnf in CatType cpoCatMixin.
```

We next define various standard constructions on ω-cpos, including:

Discrete cpos. Equipping any `X : Type` with the order $x_1 \sqsubseteq x_2$ iff $x_1 = x_2$ (i.e. Leibniz equality) yields a cpo that we write `discrete_cpoType X`. We declare `discrete_cpoType nat` and `discrete_cpoType bool` as canonical structures.

Finite products. `discrete_cpoType unit` is a terminal object, `One`, endowing `cpoCatType` with the structure of a `terminalCat`. The Cartesian product of any two predomains with the pointwise ordering and least upper bound operator (`⨆c = (⨆(π1∘ c),⨆(π2∘ c))`) is a categorical product, making `cpoCatType` a `prodCat`.

Cartesian closure. For any predomains `X` and `Y`, we equip the continuous function space `fconti X Y` with the pointwise order inherited from `Y`, yielding a preorder `fcont_ordType X Y`. We then define `X ⟹ Y : cpoType` by equipping `fcont_ordType X Y` with least upper bounds computed pointwise: if `c : natO ⟶ fcont_ordType X Y` is a chain, then `⨆c` is $\lambda d_1.\bigsqcup(\lambda n.c\,n\,d_1)$. We then define

`ev : (X ⟹ Y) * X ⟶ Y` and `exp_fun : (X * Y ⟶ Z) → (X ⟶ (Y ⟹ Z))`

and show that they satisfy the diagrams needed to give `cpoCatType` the extra structure of an `expCat`, i.e. a cartesian closed category.

We elide the details of other constructions, including finite coproducts and general indexed products, in the formalization. Whilst our cpos are not required to have least elements, those that do are of special interest. We start with pointed preorders:

```
Module Pointed.
  Definition axiom (T:ordType) (l:T) := ∀ x, l ⊑ x.
  Record mixin_of (T:ordType) := Mixin {least_elem : T; _ : axiom least_elem}.
  Record class_of T := Class
   { base :> PreOrd.class_of T; ext :> mixin_of (PreOrd.Pack base T)}.
  Coercion ordType cT := PreOrd.Pack (class cT) cT.
  Definition setoidType cT := Setoid.Pack (class cT) cT.
  Definition least cT := least_elem (class cT).
End Pointed.
Notation ⊥:= Pointed.least.
Canonical Structure Pointed.ordType.
Canonical Structure Pointed.setoidType.
Lemma leastP (O:pointedType) (x:O) : ⊥⊑ x.
```

A `cppoType` is then defined as a pointed `cpoType`, and we define the cartesian closed category `cppoCatType` of pointed ω-cpos and continuous functions.

Given `D : cppoType` and `f : D ⟶ D`, we 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 $\bot$. `FIXP : (D ⟹ D) ⟶ D` is the 'internalised' version of `fixp`. We then prove a fixed point induction principle for *admissible* predicates:

```
Definition admissible (D:cpoType) (P:D → Prop) :=
           ∀ f : natO ⟶ D, (∀ n, P (f n)) → P (⨆ f).
Lemma fixp_ind (D:cppoType) : ∀ (F: D ⟶ D)(P:D → Prop),
           admissible P → P ⊥→ (∀ x, P x → P (F x)) → P (fixp F).
```

For `D:cpoType` and `P:D→ Prop` admissible, we also define a sub-cpo structure on `{d:D | P d}`, inheriting order and lubs from `D`.

The main technical complexity in this first part of our formalization is simply the packaging and layering of so many definitions. The packed classes pattern and the explicit use of category-theoretic abstractions have both shortened and improved the structure of the development relative to our previous version, and the external interface we will use later is much more uniform. There is still plenty of room for further improvement, however, particularly in regard to the tension between the elementwise (type-theoretic) and pointfree (categorical) styles of working.

3.1. *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 generalizes Paulin-Mohring's treatment of the particular case of flat cpos, which in turn builds on work of Capretta [2005] on general recursion in type theory. We exploit Coq's support for coinductive datatypes [Coquand, 1993], defining the lift of `D` for `D:cpoType` in terms of a type `Stream` of potentially infinite streams:

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

An element of `Stream` is (classically) either the infinite `Eps(Eps(Eps( ... )))`, or a finite sequence `Eps (Eps (...Val d)...))` of `Eps` ticks, terminated by `Val d` for some `d:D`. 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 D → Stream D → 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 D → Stream D → 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 : pointedType` (and we now write the usual $\sqsubseteq$ for the order). The infinite stream of `Eps`'s, Ω, is the least element of `DL_ord`. The inferred `setoidType` structure, $\equiv$, quotients `DL_ord` by equating finite streams of any length that produce $\equiv$ -equal elements of `D`.

Constructing a `cpoType` from `DL_ord` is slightly subtle. We need to define a function that maps chains `c:natO ⟶ 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 $c_n \equiv$ `Val` d_n, then for any $m \geq n$, there is a d_m such that $c_m \equiv$ `Val` d_m and that moreover, the sequence $d_n, d_{n+1}, \ldots$, forms a chain in D. Classically, we'd like to just say that if there is such a c_n, the lub is `Val` applied to the lub of the associated chain in D, otherwise the lub is Ω. But that simple idea does not work in a constructive setting. There is no computable function that tests whether a particular c_n is Ω or not, because one can only examine finite prefixes. Using a standard trick from recursion

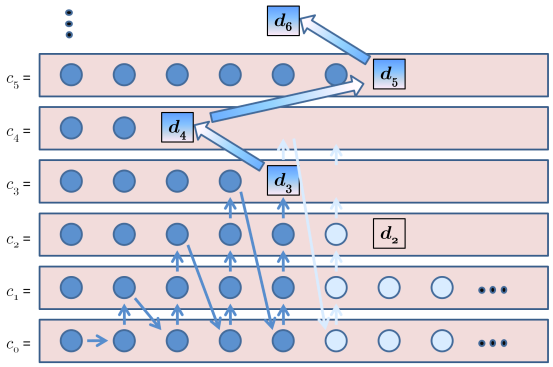

Fig. 1. Least upper bounds in $D_\perp$

theory, however, we can make a parallel corecursive search through all the c_ns simultaneously, as illustrated in Fig. 1. The output stream we need to produce is an element of `DL_ord`. Each time the interleaving search finds an `Eps` (the solid circles), 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 (in the figure, d_3) after outputting some finite number (in the figure, thirteen) 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 ($d_3 \sqsubseteq d_4 \sqsubseteq d_5 \sqsubseteq \cdots$), compute the least upper bound, $\sqcup d_i$, of *that* using the lub operation of D, and output `Val`$(\sqcup d_i)$. The definition of this construction in Coq, and the proof that it does indeed yield the least upper bound of the chain c, involve non-trivial constructive reasoning. In particular, we use a provable form of constructive indefinite description to go from knowing that there *is* a chain in D to actually having that chain in our hand, so as to be able to take its lub. The end result is a constructive lub operation on `DL_ord`, from which we can construct `D`$_\perp$ `: cppoType`, as required.

Lifting is a strong monad [Moggi, 1991] on `cpoCatType`. The unit η `: D` $\longrightarrow$ `D`$_\perp$ just applies the `Val` constructor. If `f:D` $\longrightarrow$ `E`$_\perp$ define `kleisli f : D`$_\perp$ $\longrightarrow$ `E`$_\perp$ to be the map

```
cofix kl (l:Stream D) := match l with Eps l ⇒ Eps (kl l) | Val d ⇒ f d end.
```

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'`. `kleisli f` is continuous and satisfies the equations making $(-_\perp, \eta, \texttt{kleisli})$ a Kleisli triple on `cpoCatType`. It is also convenient to have parameterized versions, `KLEISLIR D E : D * E` $\longrightarrow$ `F`$_\perp \to$ `D * E`$_\perp$ $\longrightarrow$ `F`$_\perp$, defined by composing `kleisli` with the evident strength τ `: D * E`$_\perp$ $\longrightarrow$ `(D * E)`$_\perp$.

4. A Simply-Typed Functional Language

Our first application of the formalization of ω-cpos is the denotational semantics of PCF$_v$, a simply-typed, call-by-value functional language with recursion. PCF$_v$ has integer, boolean, function and pair types. Typing environments are lists of types.

```
Inductive Ty := Int | Bool | Arrow (ty1 t2 : Ty) | Prod (ty1 t2 : Ty).
Infix " --> " := Arrow (at level 55, right associativity).
Infix " * " := Prod (at level 55).
Definition Env := seq Ty.
```

The term representation is 'strongly-typed': Coq types for variables and terms are indexed by `Ty` and `Env`, so terms are well-typed by construction. We discuss working with strongly-typed term representations in more detail elsewhere [Benton et al., 2010], so here just present the basic definitions. We separate syntactic values v from general expressions e, and restrict the syntax to 'administrative normal form', with explicit sequencing of evaluation by `LET` and inclusion of values into expressions by `VAL`:

```
Inductive Var : Env → Ty → Type :=
| ZVAR : ∀ E t, Var (t :: E) t
```

```
| SVAR : ∀ E t t', Var E t → Var (t' :: E) t.
Inductive Value E : Ty → Type :=
| TINT : nat → Value E Int
| TBOOL : bool → Value E Bool
| TVAR :> ∀ t, Var E t → Value E t
| TFIX : ∀ t1 t2, Exp (t1 :: t1 --> t2 :: E) t2 → Value E (t1 --> t2)
| TPAIR : ∀ t1 t2, Value E t1 → Value E t2 → Value E (t1 * t2)
with Exp E : Ty → Type :=
| TFST : ∀ t1 t2, Value E (t1 * t2) → Exp E t1
| TSND : ∀ t1 t2, Value E (t1 * t2) → Exp E t2
| TOP : (nat → nat → nat) → Value E Int → Value E Int → Exp E Int
| TGT : Value E Int → Value E Int → Exp E Bool
| TVAL : ∀ t, Value E t → Exp E t
| TLET : ∀ t1 t2, Exp E t1 → Exp (t1 :: E) t2 → Exp E t2
| TAPP : ∀ t1 t2, Value E (t1 --> t2) → Value E t1 → Exp E t2
| TIF : ∀ t, Value E Bool → Exp E t → Exp E t → Exp E t.
Definition CExp t := Exp nil t.
Definition CValue t := Value nil t.
```

Variables of type `Var env t` are 'typed de Bruijn indices', essentially proofs that `t` is at a particular position in `env`. The typing rule for each term constructor can be read directly off its definition. For example, `TLET` takes an expression typed as `t1` under `env`, and another expression typed as `t2` under `env` extended with a new variable of type `t1`; its whole type is then `t2` under `env`. A typed substitution `s : Sub env env'` provides a value `v : Value env' t` for each type `t` in the environment `env`. We define notation for writing substitutions as lists. Substitutions are applied to expressions by

```
subExp : ∀ (env env':Env) (ty:Ty), Sub env env' → Exp env ty → Exp env' ty
```

whilst `subVal` does the same thing for values. The operational semantics is very direct, and evaluation preserves types just by construction:

```
Inductive Ev: ∀ t, CExp t → CValue t → Prop :=
| e_Val: ∀ t (v : CValue t), 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 (n2 ≤ n1)
| e_Fst : ∀ t1 t2 (v1 : CValue t1) (v2 : CValue t2), TFST (TPAIR v1 v2) ⇓ v1
| e_Snd : ∀ t1 t2 (v1 : CValue t1) (v2 : CValue t2), TSND (TPAIR v1 v2) ⇓ v2
| e_App : ∀ t1 t2 e (v1 : CValue t1) (v2 : CValue t2),
          subExp [ v1, TFIX e ] e ⇓ v2 → TAPP (TFIX e) v1 ⇓ v2
| e_Let : ∀ t1 t2 e1 e2 (v1 : CValue t1) (v2 : CValue t2),
          e1 ⇓ v1 → subExp [ v1 ] e2 ⇓ v2 → TLET e1 e2 ⇓ v2
| e_IfTrue : ∀ t (e1 e2 : CExp t) v, e1 ⇓ v → TIF (TBOOL true) e1 e2 ⇓ v
| e_IfFalse : ∀ t (e1 e2 : CExp t) v, e2 ⇓ v → TIF (TBOOL false) e1 e2 ⇓ v
where "e '⇓' v" := (Ev e v).
```

4.1. *Denotational semantics*

The inductive semantics of types uses the product of ω-cpos to model products and continuous functions into a lifted ω-cpo to model call-by-value functions:

```
Fixpoint SemTy t : cpoType :=
  match t with
  | Int ⇒ nat_cpoType
  | Bool ⇒ (One:cpoType) + One
  | t1 --> t2 ⇒ SemTy t1 ⟹ (SemTy t2)⊥
  | t1 * t2 ⇒ SemTy t1 * SemTy t2
  end.
Fixpoint SemEnv E : cpoType :=
  match E with
  | nil ⇒ One
  | t :: E ⇒ SemEnv E * SemTy t
  end.
```

We interpret `Value E t` in `SemEnv E` $\longrightarrow$ `SemTy t`. Expressions are similar, but the range is a lifted cpo.

```
Fixpoint SemVar E t (var : Var E t) : SemEnv E ⟶ SemTy t :=
  match var with
  | ZVAR _ _ ⇒ π2
  | SVAR _ _ _ v ⇒ SemVar v ∘ π1
  end.
Fixpoint SemExp E t (e : Exp E t) : SemEnv E ⟶ (SemTy t)⊥ :=
match e with
| TOP op v1 v2  ⇒ η ∘ SimpleBOp op ∘ ⟨ SemVal v1 , SemVal v2 ⟩
| TGT v1 v2     ⇒ η ∘ SimpleBOp (fun x y ⇒ if leq x y then inl _ tt else inr _ tt)
                    ∘ ⟨ 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 (SemExp e1) (SemExp e2) (SemVal v)
| TFST _ _ v    ⇒ η ∘ π1 ∘ SemVal v
| TSND _ _ v    ⇒ η ∘ π2 ∘ SemVal v
end with SemVal E t (v : Value E t) : SemEnv E ⟶ SemTy t :=
match v with
| TINT n        ⇒ const _ n
| TBOOL b       ⇒ const _ (if b then inl _ tt else inr _ tt)
| TVAR _ i      ⇒ SemVar i
| TFIX t1 t2 e  ⇒ (FIXP : cpoCatType _ _) ∘ exp_fun (exp_fun (SemExp e))
| TPAIR _ _ v1 v2 ⇒ ⟨SemVal v1 , SemVal v2 ⟩
end.
```

`SimpleBOp` lifts Coq functions to continuous maps on discrete cpos, `const` is the K combinator and `choose` is a continuous conditional. Observe that this is exactly how one would usually present the categorical (point-free) semantics of this language. Inference fills in

the category in which we are working, its cartesian closed structure, the pointedness of the interpretation of function types that justifies the use of `FIXP`, and so on.

4.2. *Relating denotational and operational semantics*

The *soundness* theorem says that if an expression e evaluates to a value v, then the denotation of e is the lift of that of v. We start by giving the semantics of substitutions as continuous functions, defining

```
Fixpoint SemSub E E' : Sub E' E → SemEnv E ⟶ SemEnv E' := ...
```

by induction on `E'`. This allows us to state the crucial substitution lemma, which in turn is used in the `e_App` and `e_Let` cases of the soundness proof.

```
Lemma SemCommutesWithSub E:
   (∀ t (v:Value E t) E' (s:Sub E E'), SemVal v ∘ SemSub s ≡ SemVal (subVal s v))
∧ (∀ t (e:Exp E t) E' (s:Sub E E'), SemExp e ∘ SemSub s ≡ SemExp (subExp s e)).

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

We next prove *adequacy*: 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 t (R : SemTy t → CValue t → Prop) :=
  fun d e ⇒ ∀ d', d ≡ Val d' → ∃ v, e ⇓ v ∧ R d' v.
Fixpoint relVal t : SemTy t → CValue t → Prop :=
 match t with
  | Int       ⇒ fun d v ⇒ v = TINT d
  | Bool      ⇒ fun d v ⇒ v = TBOOL d
  | t1 --> t2 ⇒ fun d v ⇒ ∃ e, v = TFIX e ∧
    ∀ d1 v1, relVal t1 d1 v1 → liftRel (relVal t2) (d d1) (subExp [ v1, v ] e)
  | t1 * t2   ⇒ fun d v ⇒ ∃ v1, ∃ v2,
              v = TPAIR v1 v2 ∧ relVal t1 (fst d) v1 ∧ relVal t2 (snd d) v2
 end.
Definition relExp ty := liftRel (relVal ty).
```

The logical relation is down-closed on the left and is admissible:

```
Lemma relVal_lower: ∀ t d d' v, d ⊑ d' → relVal t d' v → relVal t d v.
Lemma rel_admissible: ∀ t v, admissible (fun d ⇒ relVal t 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 FundamentalTheorem E:
  (∀ t v senv s, relEnv E senv s → relVal t (SemVal v senv) (subVal s v)) ∧
  (∀ t e senv s, relEnv E senv s → liftRel (relVal t) (SemExp e senv) (subExp s e)).
```

Now we instantiate the fundamental theorem with closed expressions to obtain

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

The development of this section follows closely that given by Winskel [1993, chap. 11].

5. 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 [1990, 1992] and Pitts [1996], generalized to categories $\mathcal{C}$ enriched over `cppoCatType`. A key idea is to separate the positive and negative occurrences, specifying recursive domains as fixed points of locally continuous bi-functors $F : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$, i.e. objects D such that such that $F(D, D) \simeq D$.

We start by defining the type of `cppoCatType`-enriched categories with a terminal object and continuous composition:

```
Module BaseCat.
  Record mixin_of (O:Type) (M:O → O → cppoType) := Mixin
  { catm :> Category.mixin_of M;
    terminal :> CatTerminal.mixin_of (Category.Pack catm O);
    comp : ∀ X Y Z : O, M Y Z * M X Y ⟶ M X Z;
    comp_comp : ∀ X Y Z m m', comp X Y Z (m,m') ≡ Category.tcomp catm m m'
  }.
  Coercion base2 T (M:T → T → cppoType) (c:class_of M) := CatTerminal.Class c.
  Coercion terminalCat (cT:cat) : terminalCat := CatTerminal.Pack (class cT) cT.
  Definition catType (cT:cat) : catType := Category.Pack (class cT) cT.
  Definition cppoType (cT:cat) (X Y:cT) : cppoType :=
              CPPO.Pack (CPPO.class (morph X Y)) (cT X Y).
End BaseCat.
Notation cppoMorph := BaseCat.cppoType.
Notation CppoECatType := BaseCat.pack.
```

Composition in `cppoCatType`-enriched categories is inherited via the `catm` field in the mixin, but we also specify that composition agrees with a continuous map `comp` in `cppoCatType`. There are projections, such as `cppoMorph`, from morphisms to all the points in the ordered type hierarchy, most of which we have elided. These are all `Canonical Structure`s, but not declared as coercions.

A mixed variance locally-continuous bifunctor on a `cppoCatType`-enriched category $\mathcal{M}$ comprises an action on pairs of objects (`ob`), a continuous action on pairs of morphisms (`morph`), contravariant in the first argument and covariant in the second, together with proofs that `morph` respects both composition (`morph_comp`) and identities (`morph_id`):

```
Record BiFunctor : Type := mk_functor
{ ob : M → M → M ;
  morph : ∀ (T0 T1 T2 T3 : M), (cppoMorph T1 T0) * (cppoMorph T2 T3) ⟶
                               (cppoMorph (ob T0 T2) (ob T1 T3)) ;
  morph_comp: ∀ T0 T1 T2 T3 T4 T5 (f:M T4 T1) (g:M T3 T5) (h:M T1 T0)
```

```
    (k:M T2 T3), getmorph (morph T1 T4 T3 T5 (f,g)) ∘ (morph T0 T1 T2 T3 (h, k)) ≡
                                        morph _ _ _ _ (h ∘ f, g ∘ k) ;
  morph_id : ∀ T0 T1, morph T0 T0 T1 T1 (Id:M _ _, Id:M _ _) ≡ (Id : M _ _)}.
```

A pair $(f : D \longrightarrow E, g : E \longrightarrow D)$ is an embedding-projection (e-p) pair if $g \circ f \equiv id_D$ and $f \circ g \sqsubseteq id_E$. Now let `F:BiFunctor`. For any e-p pair (f, g), (`morph F` (g, f), `morph F` (f, g)) is an e-p pair. We define a sequence of objects by iterating `F`, starting with the terminal object in $\mathcal{M}$ and linked by a corresponding sequence of e-p pairs:

```
Fixpoint Diter (n:nat) :=
    match n return M with | O ⇒ One | S n ⇒ ob F (Diter n) (Diter n) end.
Fixpoint Injection (n:nat) : (Diter n) ⟶ (Diter (S n)) :=
match n with
| O ⇒ ⊥
| S n ⇒ morph F _ _ _ _ (Projection n, Injection n)
end with Projection (n:nat) : (Diter (S n)) ⟶ (Diter n) :=
match n with
| O ⇒ terminal_morph _
| S n ⇒ morph F _ _ _ _ (Injection n,Projection n)
end.
Variable comp_left_strict : ∀ (X Y Z:M) (f:M X Y),
   (⊥:Y ⟶ Z) ∘ f ≡ (⊥:X ⟶ Z).
Lemma eppair_IP: ∀ n, eppair (Injection n) (Projection n).
```

The further assumption that composition is left-strict will be discharged when we instantiate the general framework later. We call such sequences 'towers':

```
Record Tower : Type := mk_tower
{ tobjects : nat → M;
  tmorphisms : ∀ i, (tobjects (S i)) ⟶ (tobjects i);
  tmorphismsI : ∀ i, (tobjects i) ⟶ (tobjects (S i));
  teppair : ∀ i, eppair (tmorphismsI i) (tmorphisms i)}.

Definition DTower := mk_tower eppair_IP.
```

We define limits of towers and require the enriched category to have a limiting cone for every tower:

```
Record Cone (To:Tower) : Type := mk_basecone
{ tcone :> M;
  mcone : ∀ i, tcone ⟶ (tobjects To i);
  mconeCom : ∀ i, tmorphisms To i ∘ mcone (S i) ≡ mcone i }.

Record Limit (To:Tower) : Type := mk_baselimit
{lcone :> Cone To;
 limitExists : ∀ (A:Cone To), (tcone A) ⟶ (tcone lcone);
 limitCom : ∀ (A:Cone To), ∀ n, mcone A n ≡ mcone lcone n ∘ limitExists A;
 limitUnique : ∀ (A:Cone To) (h: (tcone A) ⟶ (tcone lcone))
    (C:∀  n, mcone A n ≡ mcone lcone n ∘ h), h ≡ limitExists A }.

Variable L:∀  T:Tower, Limit T.
```

Under these assumptions, we can construct the desired solution:

```
Definition DInf : M := tcone (L DTower).
Definition Fold : (ob F DInf DInf) ⟶ DInf := ...
Definition Unfold : DInf ⟶ (ob F DInf DInf) := ...
Lemma FU_id : Fold ∘ Unfold ≡ Id.
Lemma UF_id : Unfold ∘ Fold ≡ Id.
```

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 `DInf` comes equipped with a chain of retractions ρ_i : `DInf` $\longrightarrow$ `DInf` such that $\bigsqcup_i \rho_i \equiv Id$. A more abstract and useful principle is given by Pitts's [1996] characterization of the solution as a *minimal invariant*, which is how we will establish the existence of a recursively defined logical relation in Section 6.1. We define `delta : (cppoMorph DInf DInf)` $\longrightarrow$ `(cppoMorph DInf DInf)` such that

```
Lemma delta_simpl e : delta e = Fold ∘ morph F _ _ _ _ (e,e) ∘ Unfold.
```

We then show the minimal invariance property by 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 `delta`:

```
Lemma id_min : (FIXP delta : M _ _) ≡ Id.
```

6. 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. We index the type `Value` of values and `Exp` of expressions by an environment of type `nat`. The evaluation relation is as follows:

```
Inductive Evaluation : Exp 0 → Value 0 → Prop :=
| e_Val : ∀ v, VAL v ⇓ v
| e_App : ∀ e1 v2 v, subExp [v2] e1 ⇓ v → APP (LAMBDA e1) v2 ⇓ v
| e_Let : ∀ e1 v1 e2 v2, e1 ⇓ v1 → subExp [v1] e2 ⇓ v2 → LET e1 e2 ⇓ v2
| e_Ifz1 : ∀ e1 e2 v1, e1 ⇓ v1 → IFZ (INT 0) e1 e2 ⇓ v1
| e_Ifz2 : ∀ e1 e2 v2 n, e2 ⇓ v2 → IFZ (INT (S n)) e1 e2 ⇓ v2
| e_Op : ∀ op n1 n2, OP op (INT n1) (INT n2) ⇓ INT (op n1 n2)
where "e ⇓ v" := (Evaluation e v).
```

6.1. *Semantic Model*

We interpret the unityped language in a solution for the recursive domain equation $D \simeq \mathbb{N} + (D \Longrightarrow D_\perp)$, following the intuition that a value is either a number or a

function, and functions may diverge when applied to values. The solution is found in the Kleisli category for the lift monad, i.e., our encoding of the category of cpos and partial functions. Notice that this Kleisli category is classically isomorphic to the category $\mathrm{Cppo}_\perp$ of domains and strict functions often used on paper [Pitts, 1996, Reynolds, 1974]. The construction in Coq is an instantiation of results from the previous section.

We start by constructing the Kleisli category as a `cppoCatType`-enriched category with left strict composition:

```
Lemma kcpoCatAxiom : @Category.axiom cpoType
  (fun X Y ⇒ exp_cppoType X (liftCppoType Y)) (fun X Y Z f g ⇒ kleisli f ∘ g) (@η).
Canonical Structure kcpoCatMixin := CatMixin kcpoCatAxiom.
Canonical Structure kcpoCatType := Eval hnf in CatType kcpoCatMixin.

Canonical Structure kcpoBaseCatType := Eval hnf in CppoECatType kcpoBaseCatMixin.
Lemma leftss : (∀ (X Y Z : kcpoBaseCatType) (f : kcpoBaseCatType X Y),
    (⊥:kcpoCatType _ _) ∘ f ≡ (⊥: X ⟶ Z)).
```

And we show that it has all limits of towers:

```
Definition kcpoLimit (T:Tower kcpoBaseCatType) : Limit T.
```

We then build the bifunctor $F(D, E) = \mathbb{N} + (D \Longrightarrow E_\perp)$ using combinators:

```
Definition FS := biSum (biConst (discrete_cpoType nat)) biFun.
```

And then we construct the solution, defining predomains `DInf` and `VInf` for values:

```
Definition DInf : cpoType := @DInf kcpoBaseCatType kcpoLimit FS leftss.
Definition VInf := (discrete_cpoType nat) + (DInf ⟹ DInf ⊥ ).
Definition Fold : VInf ⟶ DInf ⊥ := Fold kcpoLimit FS leftss.
Definition Unfold : DInf ⟶ VInf ⊥ := Unfold kcpoLimit FS leftss.
```

We notice that for all isomorphisms in the Kleisli category: $f : D \longrightarrow E_\perp$ and $g : E \longrightarrow D_\perp$, $\mathit{kleisli}(f) \circ g = \eta$ and $\mathit{kleisli}(g) \circ f = \eta$, f and g are total functions and we obtain:

```
Lemma foldT : total Fold.
Lemma unfoldT : total Unfold.
Definition Roll : VInf ⟶ DInf := totalL foldT.
Definition Unroll : DInf ⟶ VInf := totalL unfoldT.
Lemma RU_id : Roll ∘ Unroll ≡ Id.
Lemma UR_id : Unroll ∘ Roll ≡ Id.
```

We also get the minimal invariant property:

```
Definition delta:(DInf ⟹ DInf ⊥ )⟶ (DInf ⟹ DInf ⊥ ) := delta kcpoLimit FS leftss.
Lemma id_min : η≡ FIXP delta.
```

Environments are interpreted as n-ary products of `VInf`, with projections:

```
Fixpoint SemEnv E : cpoType := match E with O ⇒ One | S E ⇒ SemEnv E * VInf end.
Fixpoint SemVar E (v : Var E) : SemEnv E ⟶ VInf :=
  match v with
  | ZVAR _  ⇒ π2
```

```
  | SVAR _ v ⇒ SemVar v ∘ π1
  end.
```

The semantics of values and expressions of the unityped language are then defined by:

```
Fixpoint SemVal E (v:Value E) : SemEnv E ⟶ VInf :=
match v return SemEnv E ⟶ VInf with
| INT i ⇒ in1 ∘ const _ i
| VAR m ⇒ SemVar m
| LAMBDA e ⇒ in2 ∘ exp_fun (kleisli (η ∘ Roll) ∘ SemExp e ∘ Id ×Unroll)
end with SemExp E (e:Exp E) : SemEnv E ⟶ VInf ⊥ :=
match e with
| VAL v ⇒ η∘ SemVal v
| APP v1 v2 ⇒ kleisli (η ∘ Unroll) ∘ ev ∘
    ⟪ [ @const _ (exp_cppoType _ _) ⊥, Id] ∘ SemVal v1, Roll ∘ SemVal v2 ⟫
| LET e1 e2 ⇒ ev ∘ ⟨exp_fun (KLEISLIR (SemExp e2)), SemExp e1 ⟩
| OP op v0 v1 ⇒ kleisli (η ∘ in1 ∘ SimpleBOp op) ∘ uncurry (Smash _ _) ∘
                ⟪ [ η, const _ ⊥]∘ SemVal v0, [η, const _ ⊥]∘ SemVal v1⟫
| IFZ v e1 e2 ⇒ ev ∘
  [ [ exp_fun (SemExp e1 ∘ π2), exp_fun (SemExp e2 ∘ π2)] ∘ zeroCase ,
    @const _ (exp_cppoType _ _) ⊥]× Id ∘ ⟪SemVal v, Id⟫
end.
```

Here `zeroCase : nat_cpoType ⟶ One + nat_cpoType` is the 'error-signalling', continuous predecessor function. The expression semantics is defined to yield $\bot$ in case of type errors (e.g. in the `APP` case).

6.2. *Soundness and Adequacy*

As in the typed case, we define semantic substitutions and show a substitution lemma:

```
Lemma SemCommutesWithSub E:
   (∀ (v : Value E) E' (s : Sub E E'), SemVal v ∘ SemSub s ≡ SemVal (subVal s v))
∧ (∀ (e : Exp E) E' (s : Sub E E'), SemExp e ∘ SemSub s ≡ SemExp (subExp s e)).
```

Soundness is then shown using the substitution lemma and the isomorphism of the predomain `DInf` in the case for `APP`. The proof proceeds by induction, using equational reasoning to show that evaluation preserves semantics:

```
Lemma Soundness e v : (e ⇓ v) → SemExp e ≡ η∘ SemVal v.
```

The proof of adequacy again uses a logical relation between syntax and semantics, but this cannot now be defined simply 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 fixed point we seek is not monotone. Following Pitts [1996], 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 $\equiv$-respecting relations between elements of our domain of values `VInf` and closed syntactic values in `Value 0` and show that this is closed under intersection, so admissible relations form a complete lattice, `RelAdm`.

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 `VInf * Value 0` 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 : DInf` $\Longrightarrow$ `DInf` $_\perp$ satisfies the 'logical' property

```
∀ d v, R (d, v) → ∀ d', kleisli (η ∘ Unroll) (f (Roll d)) ≡ Val d' →
  ∃ v', subExp [v'] e ⇓ v2 ∧ S (d',v')
```

It is easy to show that `RelV` maps admissible relations to admissible relations and is contravariant in its first argument and covariant in its second. Hence the function λR : `RelAdm`^{op}. λS : `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 $\Delta^- \equiv \Delta^+$, so we have found a fixed point, `LR` of `RelV`, which is the logical relation needed to prove adequacy.

We extend `LR` to `ELR`, a relation on `VInf` $_\perp$ `* Exp 0`, by `ELR (d,e)` if and only if for all `d'` if `d` $\equiv$ `Val d'` then there exists a value `v` and a derivation `e` $\Downarrow$ `v` such that `LR (d',v)`, and then prove fundamental theorem for these relations by simultaneous structural induction on values and expressions:

```
Theorem FundamentalTheorem n :
(∀  (v:Value n) sl d, LRsubst d sl → (RelV LR LR) (SemVal v d, subVal sl v)) ∧
∀ (e:Exp n) sl d, LRsubst d sl → ELR (SemExp e d) (subExp sl e).
```

Adequacy is then a corollary of the fundamental theorem:

```
Corollary Adequacy (e:Exp 0) d : SemExp e tt ≡ Val d → ∃ v, e ⇓ v.
```

7. Complete Ultra-Metric Space Theory

We continue with a formalization of complete 1-bounded ultrametric spaces. Classically an ultrametric space is a set M equipped with a measure function $d : M \times M \to \mathbb{R}$ such that $d(x, y) \geq 0$ and the following three conditions holds for all x, y, z:

$$d(x, y) = d(y, x) \qquad d(x, y) = 0 \Leftrightarrow x = y \qquad d(x, z) \leq \max(d(x, y), d(y, z))$$

We notice that this axiomization uses neither the additive nor multiplicative structure of the reals, and indeed the ultrametric spaces used to model programming languages are usually *bisected* [Birkedal et al., 2010b]: every non-zero distance is 2^m for some m. It is easy see that the set $\{(x, y) \mid d(x, y) \leq \frac{1}{2^n}\}$ is an equivalence relation for every n, and the $n+1$'th equivalence relation is contained in the n'th. Conversely, given a sequence of equivalence relations D_n on M where $D_{n+1} \subseteq D_n$ (as subsets of $M \times M$), we can define a metric d by $d(x, y) = 0$ if $x = y$ and otherwise $d(x, y) = 2^{-\max\{n | x \ D_n \ y\}}$, and (M, d) is a 1-bounded ultrametric space. On bisected ultrametric spaces the two notations coincide, and for the ease of formalization, we choose to represent a bounded ultrametric space

as a `setoidType` with a sequence of equivalence relations D_n such that $D_{n+1} \subseteq D_n$, and $x\ D_n\ y$ for every n is equivalent to $x \equiv y$:

```
Module Metric. Section Axioms.
  Variable M:setoidType.
  Variable Mrel : ∀ n : nat, M → M → Prop.
  Definition Mrefl x y := ((∀ n, Mrel n x y) ↔ x ≡ y).
  Definition Msym n x y := Mrel n x y → Mrel n y x.
  Definition Mtrans n x y z := Mrel n x y → Mrel n y z → Mrel n x z.
  Definition Mmono x y n := Mrel (S n) x y → Mrel n x y.
  Definition Mbound x y := Mrel O x y.
  Definition axiom := ∀ n x y z, @Mrefl x y ∧ @Msym n x y ∧ @Mtrans n x y z ∧
                                 @Mmono x y n ∧ @Mbound x y.
  End Axioms.
  Record mixin_of (M:setoidType) : Type := Mixin
    { Mrel : ∀ n : nat, M → M → Prop; _ : axiom Mrel }.
  Coercion setoidType cT := Setoid.Pack (class cT) cT.
End Metric.
Notation metricType := Metric.type.
Definition Mrel (m:metricType) : nat → m → m → Prop :=
  let: Metric.Mixin r _ := Metric.met (Metric.class m) in r.
Notation "x '=' n '=' y" := (@Mrel _ n x y) : M_scope.
```

Next we define the morphisms in the category of bounded ultrametric spaces. A function between ultrametric bisected spaces is non-expansive iff it maps n-equivalent elements to n-equivalent results [Birkedal et al., 2010b]. We package up nonexpansive functions, and define the category of bounded ultrametric spaces and nonexpansive functions:

```
Definition nonexpansive (M M':metricType) (f:M → M') : Prop :=
  ∀ (n : nat) (e e' : M), e = n = e' → f e = n = f e'.
Module FMet. Section fmet.
  Variable O1 O2 : metricType.
  Record mixin_of (f:O1 → O2) := Mixin { nonexp :> nonexpansive f }.
  Notation class_of := mixin_of (only parsing).
  Structure type : Type := Pack {sort :> O1 → O2; _ : class_of sort; _ : O1 → O2}.
  End fmet. End FMet.
Notation fmet := FMet.type.
```

We go on to define *complete* bounded ultrametric spaces. We start by defining Cauchy chains as sequences for which all elements from the n'th element onwards are n-equal. In other words, every element after the n'th is in the n'th disc centered around the n'th element. This is a very strict definition of Cauchy chains as we always know from which point on in the sequence the elements are n-related. This strict definition of Cauchy chains weakens the notion of completeness and is used crucially to show the completeness of the space of finite partial nonexpansive functions (see Section 8.2). Hence we end up with this definition:

```
Definition cchainp (M:metricType) (x:nat → M) : Prop :=
```

```
   ∀ n i j, n ⊑ i → n ⊑ j → (x i) = n = (x j).
Record cchain (M:metricType) : Type := mk_cchain
  { tchain :> nat → M; cchain_cauchy : cchainp tchain }.
```

A bounded complete ultrametric space is then defined as an bounded ultrametric space M equipped with a completion operation `comp : cchain M → M` such that every Cauchy chain c converges to `comp c`.

```
Definition mconverge (M:metricType) (c:cchain M) (x:M) : Prop :=
  ∀ n, ∃ m, ∀ i, m ⊑ i → (c i) = n = x.
Module CMetric.
  Definition axiom M (comp:cchain M → M) := ∀ c, mconverge c (comp c).
  Record mixin_of (M : metricType) : Type := Mixin
    { comp : cchain M → M; _ : axiom comp }.
  Structure type : Type := Pack {sort :> Type; _ : class_of sort; _ : Type}.
  Coercion metricType cT := Metric.Pack (class cT) cT.
End CMetric.
Notation cmetricType := CMetric.type.
Definition umet_complete (M:cmetricType) : cchain M → M :=
  CMetric.comp (CMetric.class M).
```

We continue by defining the category of bounded complete ultrametric spaces and nonexpansive functions `cmetricCatType`, and we define the product, sum, and exponential in `cmetricCatType`. As `cmetricCatType` is a full subcategory of the category of bounded ultrametric spaces the morphisms and commuting diagrams are all inherited.

The main reason for looking at bounded complete ultrametric spaces is Banach's fixed point theorem. Given a non-empty bounded complete ultrametric space M, every contractive endomorphism defined on M has a unique fixed point. We say a morphism $f : M \longrightarrow M$ is contractive if for any two points x, y, if $x \overset{n}{=} y$ then $f(x) \overset{n+1}{=} f(y)$. Given $x : M$ (M is non-empty) the sequence where the n'th elements is the n'th iteration of f on x ($\lambda n.\ f^n(x)$) is a Cauchy sequence and thus, by completeness of M, the sequence converges to a point in M. We define:

```
Definition contractive M N (f:fmet M N) : Prop :=
  ∀ n x y, x = n = y → f x = n.+1 = f y.

Fixpoint iter n (M:metricType) (f:M → M) :=
   match n with | 0 ⇒ id | S n ⇒ fun x ⇒ f (iter n f x) end.

Lemma cfixP (M:cmetricType) (f:M ⟶ M) (C:contractive f) x :
                              cchainp (fun n ⇒ iter n f x).
Definition cfix (M:cmetricType) f C x : cchain M := mk_cchain (@cfixP M f C x).
Definition fixp (M:cmetricType) f C x : M := umet_complete (@cfix M f C x).
```

We then show that `fixp f c x` is the unique fixed point of f. The last lemma below shows that taking fixed points of contractive functions is non-expansive, and thus the fixed point operator can be internalized, i.e., there is a morphism *fix* from the bounded complete ultrametric space of contractive endomorphisms on M, to M, and for every contractive f, $\mathit{fix}(f)$ is the unique (up to $\equiv$) fixed point of f.

```
Lemma fixp_eq (M:cmetricType) (f:M ⟶ M) (C:contractive f) (x:M) :
                        fixp C x ≡ f (fixp C x).
Lemma fixp_unique (M:cmetricType) f (C:contractive f) (x y:M) :
                       fixp C x ≡ fixp C y.
Lemma fixp_ne (M:cmetricType) f f' (C:contractive f) (C':contractive f')
  (x x':M) n : f = n = f' → fixp C x = n= fixp C' x'.
```

7.1. *Recursive Domain Equations in M-categories*

Scott's inverse limit construction for solving recursive domain equations was adapted to complete metric spaces by America and Rutten [1989]. We have formalized a recent generalization of the construction to M-categories, that is, categories enriched in complete bounded ultrametric spaces, due to [Birkedal et al., 2009b].

We start by defining an M-category as a category enriched over a bounded complete ultrametric space, with a terminal object, and a composition operation internalized in the metric space. The enrichment is achieved in essentially the same way as we defined cppo-enriched categories in section 5. Similarly, the definition of locally nonexpansive bifunctors closely follows that of locally continuous bifunctors in the cppo-enriched case. In the rest of this section we assume `M` is an M-category and that `F` : $\mathtt{M}^{op} \times \mathtt{M} \to \mathtt{M}$ is locally non-expansive.

An increasing Cauchy towers is a sequence of section/retraction pairs (s, r), where $\lim_{n\to\infty}(s_n \circ r_n) = id$.

```
Definition retract (T0 T1 : M) (f: T0 ⟶ T1) (g: T1 ⟶ T0) := g ∘ f ≡ Id.

Record Tower : Type := mk_tower
{ tobjects : nat → M;
  tmorphisms : ∀ i, tobjects (S i) ⟶ (tobjects i);
  tmorphismsI : ∀ i, (tobjects i) ⟶ (tobjects (S i));
  tretract : ∀ i, retract (tmorphismsI i) (tmorphisms i);
  tlimitD : ∀ n i, n ⊑ i →
    (tmorphismsI i ∘ tmorphisms i : cmetricMorph _ _) = n = Id}.
```

We proceed with definitions of categorical concepts such as cones, limits, cocones, and colimits over increasing Cauchy towers. Here we only show the definition of cones and limits:

```
Record Cone (To:Tower) : Type := mk_basecone
{ tcone :> M;
  mcone : ∀ i, tcone ⟶ (tobjects To i);
  mconeCom : ∀ i, tmorphisms To i ∘ mcone (S i) ≡ mcone i }.

Record Limit (To:Tower) : Type := mk_baselimit
{ lcone :> Cone To;
  limitExists : ∀ (A:Cone To), (tcone A) ⟶ (tcone lcone);
  limitCom : ∀ (A:Cone To), ∀ n, mcone A n ≡ mcone lcone n ∘ limitExists A;
  limitUnique : ∀ (A:Cone To) (h: (tcone A) ⟶ (tcone lcone))
     (C:∀  n, mcone A n ≡ mcone lcone n ∘ h), h ≡ limitExists A }.
```

Following Birkedal et al. [2009b] we require M-categories to have limits of increasing Cauchy towers. Thence we continue by defining a tower `DTower` where the n'th object is the n'th iteration of the bifunctor starting from the terminal object `T` in M. The first retract `F T T` $\longrightarrow$ `T` is given by the fact that `T` is the terminal object. We require a morphism `T` $\longrightarrow$ `F T T`, and that the action of the bifunctor on morphisms is contractive. By contractiveness we then get the limit condition for `DTower`, and we define `DInf` as the limit of `DTower`. It is now a lengthy task to verify that the `DInf` is a solution to the recursive domain equation, finally ending up with constants:

```
Definition Fold : (ob F DInf DInf) ⟶ DInf := ...
Definition Unfold : DInf ⟶ (ob F DInf DInf) := ...
Lemma FU_id : Fold ∘ Unfold ≡ Id.
Lemma UF_id : Unfold ∘ Fold ≡ Id.
```

just as in the cppo-enriched case.

8. A CBV Lambda Calculus with Recursive Types and Higher Order Store

In this section we present an application of our formalization of solutions to recursive domain equations in M-categories by giving a step-indexed semantic model of types for a call-by-value polymorphic lambda calculus $F^{\mu!}$ with recursive types and general references. This model was sketched by Birkedal et al. [2010b]; we begin by recalling the basic ideas. It is a simple unary model, interpreting types as predicates over untyped terms and sufficing to establish a semantic type soundness result, but illustrates many of the core challenges of modelling languages with higher-order store.

Following realizability style models, types will be interpreted by certain 'well-behaved' predicates on the set V of syntactic $F^{\mu!}$ values. We write $UPred(V)$ for the set of all 'well-behaved' predicates. Since $F^{\mu!}$ includes dynamic allocation of general references, the model will be a Kripke model, in which semantic types are indexed by possible worlds that specify which locations are allocated, and what the (semantic) type of the value stored in each allocated location is assumed to be. Thus we end up with recursive equations of roughly this form:

$$\begin{aligned} W &= \mathbb{N} \to_{\text{fin}} T, \\ T &= W \to_m UPred(V). \end{aligned}$$

Here locations are modeled as natural numbers, worlds as finite maps from locations to semantic types, and the m on the function space $\to_m$ refers to the requirement that semantic types should be Kripke monotone in the worlds. Worlds are ordered by the standard inclusion order when we view worlds as (functional) relations.

More precisely, we will solve the following recursive world equation

$$W \simeq \mathbb{N} \to_{\text{fin}} (\tfrac{1}{2}W \to_m UPred(V)) \tag{1}$$

in the category PreCBUlt_{ne} of *preordered* complete bounded non-empty ultrametric spaces. The space $UPred(V)$ is the object of downward closed subsets of $\mathbb{N} \times V$ (if $(k, v) \in UPred(V)$ then $(k', v) \in UPred(V)$ for all $k' < k$), equipped with the natural

metric (see formalization below) and the discrete order. The factor $\frac{1}{2}$ is a so-called shrinking factor, known from earlier work on metric domain theory, which is used to ensure that the functor corresponding to the equation above is locally contractive. Thus $\frac{1}{2}W$ is W with the metric shifted by one (see formalization below).

Having defined the set of semantic types, we can then define a meaning function from syntactic types into semantic types in logical relations style and, finally, prove the fundamental theorem of logical relations, i.e., that any well-typed program is in the interpretation of its type.

This completes the quick overview of the model; we now proceed by presenting the formalization of the syntax of $F^{\mu!}$ and then the semantics.

8.1. *Syntax of* $F^{\mu!}$

The types of $F^{\mu!}$ are defined in Coq as follows.

```
Inductive Ty (n:nat) : Type :=
  | TVar i: i < n → Ty n
  | Int: Ty n | Unit: Ty n
  | Product: Ty n → Ty n → Ty n
  | Sum: Ty n → Ty n → Ty n
  | Mu: Ty (S n) → Ty n
  | All: Ty (S n) → Ty n
  | Arrow: Ty n → Ty n → Ty n
  | Ref: Ty n → Ty n.
```

As with *terms* in the language of Section 6, the *types* here are well-scoped by construction, with μ- and $\forall$-bound type variables represented by de Bruijn indices.

Terms are again split into values and expressions, utilising `LET` for sequencing and `VAL` to embed values into expressions. Well-scoping of type variables alone is by construction, as for types; well-scoping of term variables and typing of terms will be specified by an inductively-defined typing judgment.

We also define a type `cvalue` of closed values with a single constructor `CValue` carrying a value and a proof that it is closed. Likewise we have a type `cexpression` of closed expressions. Substitution is standard and we have constants for type substitutions (`tsubst`, `etsubst`, `vtsubst`) and value substitutions (`csubstV`, `csubstE`) for the different types of syntax. The syntactic store-typings that occur in typing derivations are finite maps, represented using a general sorted-list encoding that is applicable whenever the domain of the map supports a computable order relation. An advantage of this representation is that Leibniz equality coincides with extensional equality on the represented maps.

The typing judgement `i ⊢ env | se ⊢ᵥ v ::: t` means value `v` has type `t` in type environment `env` and syntactic store typing `se`, all the types being well-formed in the context of `i` free type variables; the mutually recursive judgement for expressions is similar. The step-counting natural semantics is as follows:

```
Inductive EV : nat → (Exp 0) → Heap → Value 0 → Heap → Type :=
| EvVAL h v : EV 0 (VAL v) h v h
```

```
| EvFST h v0 v1 : EV 0 (FST (P v0 v1)) h v0 h
| EvSND h v0 v1 : EV 0 (SND (P v0 v1)) h v1 h
| EvOP h op n0 n1 : EV 0 (OP op (P (INT n0) (INT n1))) h (INT (op n0 n1)) h
| EvUNFOLD h v t : EV 1 (UNFOLD (FOLD t v)) h v h
| EvREF (h:Heap) v (l:nat) : l ∉ dom h → EV 1 (REF v) h (LOC l) (updMap l v h)
| EvBANG (h:Heap) v (l:nat) : h l = Some v → EV 1 (BANG (LOC l)) h v h
| EvASSIGN (h:Heap) v (l:nat) : h l → EV 1 (ASSIGN (LOC l) v) h UNIT (updMap l v h)
| EvLET h n0 e0 v0 h0 n1 e1 v h1 : EV n0 e0 h v0 h0 →
   EV n1 (substE (v0::nil) e1) h0 v h1 → EV (n0 + n1) (LET e0 e1) h v h1
| EvAPP h n t0 e v0 v h0 : EV n (substE (v::nil) e) h v0 h0 →
   EV n (APP (LAM t0 e) v) h v0 h0
| EvTAPP h n e t v0 h0 : EV n (etsubst (t::nil) e) h v0 h0 →
   EV n (TAPP (TLAM e) t) h v0 h0
| EvCASEL h n t v e0 e1 v0 h0 : EV n (substE (v::nil) e0) h v0 h0 →
   EV n (CASE (INL t v) e0 e1) h v0 h0
| EvCASER h n t v e0 e1 v0 h0 : EV n (substE (v::nil) e1) h v0 h0 →
   EV n (CASE (INR t v) e0 e1) h v0 h0.
```

$F^{\mu!}$ satisfies the usual substitution and type preservation lemmas.

8.2. *Semantics of* $F^{\mu!}$

We start out by defining the objects of $\mathrm{PreCBUlt}_{ne}$. We define the objects by inheriting the structure of preorders and the structure of bounded complete ultrametric spaces. Furthermore we require the preorder to respect the equivalence relation from the metric space, and the completion process of Cauchy chains must respect the preorder as well.

```
Module PreCBUmet.
  Definition respect (S:setoidType) (le:S → S → Prop) :=
    ∀ s s' t t':S, s ≡ s' → t ≡ t' → le s t → le s' t'.
  Definition axiom (T:cmetricType) (le:T → T → Prop) :=
     respect le ∧ ∀ c c' : cchain T, (∀ i, le (c i) (c' i)) →
                                le (umet_complete c) (umet_complete c').
End PreCBUmet.
```

The morphisms of $\mathrm{PreCBUlt}_{ne}$ are the morphisms of the underlying metric space that are also monotonic with respect to the preorder:

```
Module FPCM. Section fpcm.
  Variable O1 O2 : pcmType.
  Record class_of (f:O1 → O2) :=
    Class { base :> FMet.mixin_of f; ext :> FMon.mixin_of f }.
  Coercion base2 f (c:class_of f) : FMon.mixin_of f := fmonoMixin c.
End fpcm. End FPCM.
Notation fpcm := FPCM.type.
```

Next we define the standard type operators, sums, products, exponentials, etc, and we show that $\mathrm{PreCBUlt}_{ne}$ is an M-category. We show that $\mathrm{PreCBUlt}_{ne}$ has limits of Cauchy towers and thus we obtain solutions to recursive domain equations. We then

build a small library of bifunctor combinators to find a solution to the recursive domain equation mentioned in the overview above. We start with downwards closed sets. Given a type `T` we define:

```
Definition downclosed (p:nat * T → Prop) := ∀ n k t, k < n → p (n,t) → p (k,t).
```

and we give the Σ-type `{ p : nat * T → Prop | downclosed p }` the following structure. Two sets are equivalent iff the sets contain the same elements. Two sets A, B are n-related iff the respective subsets of element (k, e) where $k < n$ are equal. An element (k, e) is in the completion of a Cauchy chain iff (k, e) is in the $(k + 1)$'th element of the chain. The order on these sets is the standard subset order, and it is easy to show the required axioms. We define $UPred(T)$ in $\mathrm{PreCBUlt}_{ne}$ as the object of downward closed subsets of $\mathbb{N} \times T$ and refer to the elements as *uniform predicates* (by analogy with complete uniform PERs [Amadio and Curien, 1998]).

For finite partial maps we define two maps to be equivalent iff they have the same domain, and on their common domain they have equivalent values. Two maps are $(n+1)$-equal iff they have a common domain and their respective values on the elements in the domain are $(n + 1)$-equal (by boundedness any two maps must be 0-related). The completion of a Cauchy chain has the domain of the second element in the chain (notice how the strong definition of Cauchy chains from Section 7 is used), and for each element in this domain, the chain specializes, via application, to a chain in the codomain, whose completion we may take. We have coded up this process and shown that a Cauchy chain of finite partial functions converges to its result. The preorder on finite partial maps is the extension order: a map is less than another map iff the domain of the first map is a subset of the domain of the second map, and on the common part of their domains they have equivalent values. We let `findom_pcmType T T'` denote the object of finite partial maps from `T` to `T'`, and show that it is functorial in `T'` via post composition.

We continue by solving the recursive domain equation (1), and thus obtain the isomorphism `Fold` and `Unfold`:

```
Definition BF : BiFunctor pcmECatType := findomBF
 ((BiComp (halveBF idBF) (constBF (upred_pcmType (cvalue O))) BiArrow))
                                                  [compType of nat].
Definition W : pcmType := @DInf BF morph_contractive.
Definition Unfold := @Unfold BF morph_contractive :
  W ⟶ findom_pcmType [compType of nat]
                     ((halve_pcmType W ⟹ upred_pcmType (cvalue O))).
Definition Fold := @Fold BF morph_contractive :
  findom_pcmType [compType of nat]
                 ((halve_pcmType W ⟹ upred_pcmType (cvalue O))) ⟶ W.
```

Next we give an interpretation of types. We start by interpreting type contexts as products of morphisms in $\mathrm{PreCBUlt}_{ne}$ from $\frac{1}{2}W$ to $UPred(V)$:

```
Definition TV := halve_pcmType W ⟹ upred_pcmType (cvalue O).
Fixpoint TVal (n:nat) : cmetricType :=
match n with | O ⇒ One | S n ⇒ TVal n * TV end.
```

```
Fixpoint pick n j : (j < n) → cmetricCatType (TVal n) TV :=
match n as n0, j as j0 return j0 < n0 → TVal n0 ⟶ TV with
| O,_ ⇒ fun F ⇒ match less_nil F with end
| S n,S j ⇒ fun F ⇒ @pick n j F ∘ π1
| S n, O ⇒ fun F ⇒ π2
end.
```

We then proceed by defining a uniform predicate on closed values for each type constructor. For simple values we simply define a suitable set like this:

```
Lemma upred_int_down :
  downclosed (fun kt ⇒ match snd kt : cvalue O with
                     | (CValue (INT i) _) ⇒ True | _ ⇒ False end).
Definition upred_int : upred_pcmType (cvalue 0) :=
                     exist (@downclosed _) _ upred_int_down.
```

For the recursive types we define an operator `upred_mu` with the intension that if `R` is the meaning of `t` then `FIXP ∘ upred_mu R` is the meaning of `Mu t`. We start by defining the downward closed predicate for the values in the recursive type, and then package it as a morphism.

```
Lemma upred_mu_down n
   (R: TVal n.+1 ⟶ halve_pcmType W ⟹ upred_pcmType (cvalue O)) (s:TVal n)
   (P:(halve_pcmType W ⟶ upred_pcmType (cvalue O)) * halve_pcmType W) :
downclosed (fun kt ⇒ let: CValue v' p' := snd kt in
  match fst kt, v' as v0 return closedV v0 → Prop with
 | O,FOLD t v ⇒ fun X ⇒ True
 | S k,FOLD t v ⇒ fun X ⇒ upred_fun (R ((s,(fst P))) (snd P)) (k,CValue v X)
 | _,_ ⇒ fun X ⇒ False
  end p').


Definition upred_mut n R s w : upred_pcmType (cvalue O) :=
  exist (@downclosed _) _ (@upred_mu_down n R s w).


Definition upred_mu n (R: TVal n.+1 ⟶ halve_pcmType W ⟹ upred_pcmType (cvalue O)) :
   TVal n ⟶ morphc_pcmType TV TV :=
    Eval hnf in mk_fmet (@upred_mun n R).
```

The following relation defines when a heap of closed values, `h`, satisfies the typing constraints of a world, `w`, up to an index, `k`:

```
Definition heap_world k (h:cheap) (w:W) :=
    ∀ j, j < k → dom (heap h) = dom (Unfold w) ∧
    ∀ l (I: l \in dom (Unfold w)) (I':l \in dom (heap h)),
            upred_fun (indom_app I w) (j,CValue (indom_app I') (heap_cl I')).
```

Then we give the definition of when an expression e is in a semantic type f at step-level k and world w. Note how this formal definition closely resembles definitions familiar from step-indexed models, such as that of Ahmed et al. [2009].

```
Definition IExp (f:TV) (k:nat) (e:cexpression 0) (w:W) :=
  ∀ j, (j ⊑ k)%N →
  ∀ (h h':cheap) v (D:EV j e h v h'), heap_world k h w →
   ∃ w':W, w ⊑ w' ∧ heap_world (k - j) h' w' ∧
              upred_fun (f w') (k-j,CValue v (proj2 (cev D))).
```

We then continue with the definitions for the remaining type constructors (for space reasons we only show the definitions for arrow types and ref types here).

```
Lemma upred_arrow_down n
  (R0 R1: TVal n ⟶ halve_pcmType W ⟹ upred_pcmType (cvalue 0))
  (s:TVal n) (w: halve_pcmType W) : downclosed (fun kt ⇒ let: (k,v) := kt in
   match v with
   | CValue (LAM t' e) p ⇒ ∀ w' j (va:cvalue 0), w ⊑ w' → (j ⊑ k)%N →
             upred_fun (R0 s w') (j,va) → IExp (R1 s) j (csubstE [:: va] e) w'
   | _ ⇒ False end).

Definition upred_arrowt n R0 R1 s w : upred_pcmType (cvalue 0) :=
  exist (@downclosed _) _ (@upred_arrow_down n R0 R1 s w).

Definition upred_arrow n
   (R0 R1:TVal n ⟶ halve_pcmType W ⟹ upred_pcmType (cvalue 0)) :
   cmetricCatType (TVal n) (halve_pcmType W ⟹ upred_pcmType (cvalue 0)) :=
   Eval hnf in mk_nemon (upred_arrowN (R0,R1)).

Lemma upred_ref_down n
 (R : TVal n ⟶ halve_pcmType W ⟹ upred_pcmType (cvalue 0))
 (s:TVal n) (w: halve_pcmType W) :

Definition upred_reft n R w : upred_pcmType (cvalue 0) :=
  exist (@downclosed _) _ (@upred_ref_down n R (fst w) (snd w)).

Definition upred_ref n
     (R : TVal n ⟶ halve_pcmType W ⟹ upred_pcmType (cvalue 0)) :
     (TVal n) ⟶ (halve_pcmType W ⟹ upred_pcmType (cvalue 0)) :=
     Eval hnf in mk_nemon (upred_refN R).
```

Now we can give the value interpretation of types. The constants `Pcomp` and `pprod_fun_ne` are internalizations of the composition operation and the universal morphism of the product in PreCBUlt$_{ne}$, respectively.

```
Fixpoint IVal n (t:Ty n) : cmetricCatType (TVal n) TV :=
match t with
| TVar n J ⇒ pick J
| Int ⇒ mconst _ (pconst _ upred_int)
| Unit ⇒ mconst _ (pconst _ upred_unit)
| Mu t ⇒ FIXP ∘ upred_mu (IVal t)
| t ∗ t' ⇒ (exp_fun Pcomp upred_product : metricCatType _ _) ∘ pprod_fun_ne
                                                  ∘ ⟨IVal t,IVal t'⟩
```

```
| Sum t t' ⇒ (exp_fun Pcomp upred_sum : metricCatType _ _) ∘ Pprod_fun
                                                    ∘ ⟨IVal t, IVal t'⟩
| All t ⇒ upred_all (IVal t)
| t --> t' ⇒ upred_arrow (IVal t) (IVal t')
| Ref t ⇒ upred_ref (IVal t)
end.
```

We interpret type environments of size m into uniform predicates on an m-ary product of closed values:

```
Fixpoint IEnv n (e:TypeEnv n) :
  TVal n ⟶ halve_pcmType W ⟹ upred_pcmType (Prod (cvalue 0) (size e)) :=
match e as e0 return
  TVal n ⟶ halve_pcmType W ⟹ upred_pcmType (Prod (cvalue 0) (size e0)) with
| nil ⇒ mconst _ (pconst _ (upred_empty unit))
| t::te ⇒ (pcompM _ _ _ ∘ ppair _ Prod_cons ∘ Pprod_fun) ∘ ⟨IEnv te, IVal t ⟩
end.
```

We interpret store-types into uniform predicates on worlds:

```
Lemma IStore_down n (Se:StoreType n) (s:TVal n) :
  downclosed (fun kt ⇒ ∀ l t, Se l = Some t →
                    upred_fun (IVal (Ref t) s (snd kt)) (fst kt, cLOC _ l)).
Definition IStore n (Se:StoreType n) (s:TVal n) : upred_pcmType W :=
                          exist (@downclosed _) _ (@IStore_down n Se s).
```

Finally, we give a logical relation, show a substitution theorem for the interpretation of types, and show the fundamental theorem of the logical relation, ensuring soundness of the interpretation.

```
Definition VRel n (E:TypeEnv n) (Se:StoreType n) (v:Value n) (t:Ty n) :=
  ∀ k (s:TVal n) (ts:Prod (Ty 0) n) g w,
    upred_fun (IEnv E s w) (k,g) → upred_fun (IStore Se s) (k,w) →
 upred_fun (IVal t s w) (k, csubstV (Prod_subst g) (vtsubst (Prod_subst ts) v)).
Definition ERel n (E:TypeEnv n) (Se:StoreType n) (e:Exp n) (t:Ty n) :=
  ∀ k (s:TVal n) (ts:Prod (Ty 0) n) g w,
     upred_fun (IEnv E s w) (k,g) → upred_fun (IStore Se s) (k,w) →
     IExp (IVal t s) k (csubstE (Prod_subst g) (etsubst (Prod_subst ts) e)) w.

Lemma IVal_subst n (t:Ty n) s m s' (a:seq (Ty m)) :
     (∀ i (P:i < n), pick P s ≡ (IVal (nth Unit a i) s')) →
                       IVal t s ≡ IVal (tsubst a t) s'.

Lemma FT i E S t : (∀ v, i ⊢ E | S ⊢ᵥ v ::: t → VRel E S v t) ∧
                   (∀ e, i ⊢ E | S ⊢ₑ e ::: t → ERel E S e t).
```

9. 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 [Milner, 1972a] falls into the first category, as does Reus's [1999] work on synthetic domain theory in LEGO. HOLCF, originally due to Regensburger [1995] and later reworked by Müller et al. [1999], uses Isabelle's axiomatic type class mechanism to define and prove basic properties of ω-cpos within higher order logic; a datatype package allows certain recursively-defined domains and associated induction and coinduction principles to be introduced, though these are axiomatic rather than definitional. HOL-CPO [Agerholm, 1994a, 1995] was an extension of HOL with similar goals, and basic definitions have also been formalized in PVS [Bartels et al., 1996]. Coq's library includes a formalization by Kahn [1993] of some general theory of dcpos.

HOLCF is probably the most developed of these systems, and has been used for some non-trivial semantic applications [Nipkow, 1998, Varming and Birkedal, 2008]. Recently, Huffman has begun to extend HOLCF significantly, developing the theory of SFP-domains and powerdomains [Huffman, 2008], and a universal domain from which other recursively-defined domains can be presented as retracts [Huffman, 2009]; the intention is that this latter will provide the basis for an improved, definitional datatype package. Huffman's constructions are impressively complex, involving Gödel-numberings of finite posets, for example.

Compared with higher-order logic, working in a rich dependent type theory like that of Coq is clearly a huge advantage. 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 compiler correctness. Secondly, one really needs dependent types to work conveniently with monads and logical relations, or to formalize even concrete cases of the inverse limit construction in a natural way,[†] let alone take the further abstraction step to working with abstract and concrete categories as we do here. It should also be remarked that not only are we making full use of the power of the underlying type theory, but rely heavily on more sophisticated features of Coq's front end, some of which are rather recent additions. Sozeau's [2009] generalized rewriting tactics allow 'diagram-chasing' proofs to translate directly, whilst his `Program` and `dependent destruction` tactics [Sozeau, 2008] are crucial for working with non-trivial dependency. The elegant packaging and inference of many different kinds of structure would be impossible without `Canonical Structure`s, and Garillot et al.'s [2009] development of the pattern to exploit them. (The recently-added type class mechanism [Sozeau and Oury, 2008] might be a viable alternative.)

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

[†] As well as Huffman's work, Agerholm [1994b] has constructed 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 semantics seems to have been done with the model. Petersen [1993] formalized a reflexive cpo based on $P\omega$ in HOL, though this also does not seem to have been applied.

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 is a kind of syntax-free operational semantics, not entirely unlike game semantics; this perspective, and deeper connections with step-indexing and metric models, certainly merit further study.

The revised structuring techniques, using category theory to work at a higher level of abstraction and the packed classes pattern to build hierarchies and infer structure, have been very successful. There is, however, room for further improvement and getting the right level of generality is a delicate matter. For example, the morphisms of a category are currently defined as a `setoidType`, but the objects are not. This is insufficiently general to represent, say, arrow or functor categories, but suffices to abstract the handful of concrete categories with which we have been working. We are *not* aiming for a first-class formalization of category theory for its own sake, merely doing enough to allow us to talk about some particular concrete things in an abstract way. Too much abstraction can easily cause as many problems as too little, especially in mechanized reasoning, but we do expect to generalize the treatment of categories a little further in future.

The Coq development is of a very reasonable size. Categories and the domain theory library, including the theory of recursive domain equations, total around 4400 lines, and the metric extension is around 1400 lines. The formalization of the simply typed language and its soundness and adequacy proofs are around 1000 lines, the unityped language takes around 1200, and the language with recursive types and a higher-order store is around 5100 lines with the syntax taking up over 55% of the lines.

The development has been used in the formalization of some new research [Benton and Hur, 2009] and, although we continue to make improvements, is sufficiently mature to be of use in further non-trivial applications. The semantic model for higher-order store presented in Section 8 is a simple unary model, but involves most of the pieces necessary for the formalization of state-of-the-art relational models for reasoning about equivalence of mutable abstract data types [Ahmed et al., 2009]. The techniques can also be used to show soundness of separation logics for languages with higher order store [Schwinghammer et al., 2010, Birkedal et al., 2010b] or with storable locks [Hobor et al., 2010]. Such models can either be built using step-indexing directly over the operational semantics, as we did in Section 8, or by using a cpo-based model [Birkedal et al., 2009a, Schwinghammer et al., 2010].

References

M. Abadi and G. Plotkin. A model of cooperative threads. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–40, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. .

S. Agerholm. Domain theory in HOL. In *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *LNCS*, 1994a.

S. Agerholm. Formalizing a model of the lambda calculus in HOL-ST. Technical Report 354, University of Cambridge Computer Laboratory, 1994b.

S. Agerholm. LCF examples in HOL. *The Computer Journal*, 38(2), 1995.

A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. In *POPL*, 2009.

R. M. Amadio and P.-L. Curien. *Domains and Lambda-Calculi*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. CUP, 1998.

P. America and J. J. M. M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. *J. Comput. Syst. Sci.*, 39(3):343–375, 1989.

P. Audebaud and C. Paulin-Mohring. Proofs of randomized algorithms in Coq. In *Mathematics of Program Construction*, volume 4014 of *LNCS*, 2006.

F. Bartels, A. Dold, H. Pfeifer, F. W. Von Henke, and H. Rueß. Formalizing fixed-point theory in PVS. Technical report, Universität Ulm, 1996.

G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *J. Funct. Program.*, 13(2): 261–293, 2003.

N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *ACM International Conference on Functional Programming*, 2009.

N. Benton, A. Kennedy, and C. Varming. Some domain theory and denotational semantics in Coq. In *TPHOLs*, volume 5674 of *LNCS*, 2009.

N. Benton, C.-K. Hur, A. Kennedy, and C. McBride. Strongly typed term representations in Coq. *Journal of Automated Reasoning*, 2010. To appear.

L. Birkedal, K. Støvring, and J. Thamsborg. Realizability semantics of parametric polymorphism, general references, and recursive types. In *Proceedings of FOSSACS*, number 5504 in LNCS, pages 456–470, 2009a.

L. Birkedal, K. Støvring, and J. Thamsborg. The category-theoretic solution of recursive metric-space equations. Technical Report ITU-2009-119, IT University of Copenhagen, 2009b.

L. Birkedal, B. Reus, J. Schwinghammer, K. Stovring, J. Thamsborg, and H. Yang. Step-indexed Kripke models over recursive worlds. Submitted for publication, 2010a.

L. Birkedal, K. Støvring, and J. Thamsborg. Kripke models over recursively defined metric worlds: Steps and domains, Jan. 2010b. Submitted for publication. Available at: `http://itu.dk/people/birkedal/papers/krimrd.pdf`.

V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1, 2005.

T. Coquand. Infinite objects in type theory. In *Types for Proofs and Programs*, volume 806 of *LNCS*, 1993.

P. Freyd. Remarks on algebraically compact categories. In *Applications of Categories in Computer Science*, volume 177 of *LMS Lecture Notes*, 1992.

P. Freyd. Recursive types reduced to inductive types. In *IEEE Symposium on Logic in Computer Science*, 1990.

F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. Springer-Verlag, 2009.

A. Hobor, R. Dockins, and A. Appel. A theory of indirection via approximation. In *ACM Symposium on Principles of Programming Languages*, 2010.

B. Huffman. A purely definitional universal domain. In *TPHOLs*, volume 5674 of *LNCS*, 2009.

B. Huffman. Reasoning with powerdomains in Isabelle/HOLCF. In *TPHOLs*, volume 5170 of *LNCS*, 2008.

G. Kahn. Elements of domain theory. In the Coq users' contributions library, 1993.

R. Milner. Implementation and applications of Scott's logic for computable functions. In *ACM Conference on Proving Assertions About Programs*, 1972a.

R. Milner. Logic for computable functions: Description of a machine implementation. Technical Report STAN-CS-72-288, Stanford University, 1972b.

E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *J. Functional Programming*, 9, 1999.

T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10, 1998.

C. Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In *From Semantics to Computer Science. Essays in Honour of G Kahn*. CUP, 2009.

K. D. Petersen. Graph model of LAMBDA in higher order logic. In *Higher-Order Logic Users Group Workshop*, volume 780 of *LNCS*, 1993.

A. M. Pitts. Computational adequacy via 'mixed' inductive definitions. In *Mathematical Foundations of Programming Semantics*, volume 802 of *LNCS*, 1994.

A. M. Pitts. Relational properties of domains. *Inf. Comput.*, 127, 1996.

F. Regensburger. HOLCF: Higher order logic of computable functions. In *Theorem Proving in Higher Order Logics*, volume 971 of *LNCS*, 1995.

B. Reus. Formalizing a variant of synthetic domain theory. *J. Automated Reasoning*, 23, 1999.

J. C. Reynolds. On the relation between direct and continuation semantics. In J. Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 141–156, Berlin, 1974. Springer-Verlag.

J. Schwinghammer, H. Yang, L. Birkedal, and F. P. B. Reus. A semantic foundation for hidden state. In *FOSSACS*, 2010.

M. B. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Comput.*, 11(4):761–783, 1982.

M. Sozeau. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009.

M. Sozeau. *Un environment pour la programmation avec types dépendants*. PhD thesis, Université Paris 11, 2008.

M. Sozeau and N. Oury. First-class type classes. In *TPHOLs*, volume 5170 of *LNCS*, 2008.

C. Varming and L. Birkedal. Higher-order separation logic in Isabelle/HOLCF. In *Mathematical Foundations of Programming Semantics*, 2008.

G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.