

Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the [ENTCS Macro Home Page](#).

Ultrametric Domain Theory and Semantics in Coq

Carsten Varming^{1,2}

*Computer Science Department
Carnegie Mellon University
Pittsburgh PA, USA*

Lars Birkedal³

*IT University of Copenhagen
Copenhagen, Denmark*

Abstract

We present a Coq formalization of ultrametric spaces and of ultrametric-enriched categories, up to and including the construction of solutions to recursive domain equations in ultrametric-enriched categories. We then show how to apply this semantic setup for giving semantics to a programming language with higher-order store. Specifically, we define a step counting operational semantics for a functional language with recursive types and higher-order store; show how to define an interpretation of the types of the programming language via a Kripke logical relation over the operational semantics, using a recursively defined metric space of worlds; and prove the fundamental theorem of logical relations.

Keywords: Semantics, higher-order store, recursive types, ultra-metric spaces, Coq

1 Introduction

Various kinds of metric spaces have been used in semantics for many years, in particular to give semantics of concurrent and non-deterministic languages, but also to give semantics of PCF, see, e.g., [12,3,11,22,14] and the references therein. Recently, the second author and colleagues have shown how a wide

¹ NSF Grant: XX YY

² Email: varming@cmu.edu

³ Email: birkedal@itu.dk

variety of models of programming languages with higher-order store⁴ can usefully be seen as Kripke models over recursively-defined complete ultrametric spaces of worlds [9,19,10]. In several of these applications, the recursively-defined complete ultrametric space also comes equipped with an order relation (for capturing the idea of a future world in the Kripke model). This led Birkedal, Støvring, and Thamsborg [8] to the development of a formulation of the inverse-limit construction of solutions to recursive domain equations based on categories enriched in complete ultrametric spaces (called M -categories), a generalization of the classical work for complete ultrametric spaces [12,3].

In this paper we present a formalization of complete bounded ultrametric spaces and the solution of recursive domain equations in M -categories. The formalization is done in constructive type theory, using the Coq theorem prover.

As shown in [10], the approach to modeling higher-order store via Kripke models over recursively-defined worlds applies both to situations where the programming language is modeled using denotational semantics via standard Scott domains and to situations where the programming language is modeled using step-indexing directly over the operational semantics [5].

To verify the applicability of our formalization of recursive domain equations in M -categories, we present an example application, namely the formalization of a step-indexed model of a programming language with impredicative polymorphism, references, and recursive types. This application was sketched in [10], but here it is fully formalized in Coq. We relate it to recent formalizations of step-indexed models by Hobor et. al. [16,4] in Section 5.

The work presented in this paper can be seen as a continuation of earlier work of Benton et. al. [7], who showed how to solve recursive domain equations in standard categories of domains in Coq and used it to give the first mechanization of a denotational semantics of a higher-order programming language without store. The techniques we present here can be used in concert with the techniques in [7] to formalize models of higher-order store based on domain-theoretic models of the programming language, e.g., following [9].

Our Coq formalization uses no axioms other than what is standard in the Coq theorem prover, and in this sense it is a constructive type theoretic formalization. In particular, we show how to endow Coq types with bounded ultrametric structure and functions computing the limit of Cauchy chains, and thus end up with bounded complete ultrametric types. Our formalization makes use of ideas from `ssreflect` [15] to simplify reasoning about equality and to make the formalization modular such that it, for example, is easy to form types equipped with both a metric and an order relation. For reasons of space

⁴ Higher-order store refers to the possibility of storing computations in the store and is a feature found in most modern programming languages, e.g., in ML.

some definitions and proofs have been omitted; referees can find some of them in the appendices.

Outline

In Section 2 we present our formalization of complete bounded ultrametric spaces and in Section 3 we present the formalization of the solution to recursive domain equations in M -categories. The application to semantics of a programming language with higher-order store is presented in Section 4, and we conclude and discuss further related work in Section 5. You can find a copy of the Coq source files at <http://www.cs.cmu.edu/~cvarming/papers/archive/metric.tgz>

2 Formalization of CBUlt Spaces

The formalization of complete 1-bounded ultrametric spaces starts out with a standard formalization of the category of total setoids and equivalence-respecting functions [17,6]. The only difference between our framework and the formalization in [6] is the adoption of the ideas of [15] to support a complex mathematical hierarchy.

We denote the equivalence relation of a Setoid with \equiv and declared it as a parametric `Setoid` relation. Most of the constructions that follows are proved and declared to be morphisms that respect \equiv ; this enables convenient equational rewriting.

We also define a number of types as Setoids, e.g., products, sums, exponentials, options, and we define *Props* as the setoid of propositions in which two elements are related iff they are interprovable. The exponential in Setoids is the set of equivalence respecting functions which we denote with \rightarrow_s .

2.1 Ultrametric Spaces

In classical presentations of metric spaces the metric is given as a function from pairs of elements to the reals. We cannot expect to use a naive representation of such a function in Coq since the distance between two processes cannot be expected to be computable (though a co-inductive definition of the reals may make such a solution possible). An alternative is to use a constructive variant of metric spaces, but since we are only going to use ultrametric 1-bounded spaces we can do something much simpler. Indeed, none of the axioms of ultrametric spaces uses the additive or the multiplicative structure of the reals, and what is important for us, is just the notion of convergence. Moreover, in our intended applications, all distances are of the form 2^{-n} , and thus we may replace the reals with the natural numbers. In summary, we end up formalizing ultrametric spaces that are also *bisected* [10], i.e., with

distances of the form 2^{-n} , as setoids equipped with a natural number indexed family of equivalence relations:

```

Module METRIC.
Section Axioms.
Variable M:setoidType.
Variable Mrel : ∀ n : nat, M →s M →s Props.
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 →s M →s Props;   _ : axiom Mrel }.
Record class_of (T:Type) : Type :=
  Class { base :> Setoid.class_of T ;   met :> mixin_of (Setoid.pack base) }.
Structure type : 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 repack cT : _ → Type → type := let k T c p := p c in unpack k cT.
Definition pack := let k T c m := Pack (@Class T c m) T in Setoid.unpack k.
Coercion setoidType cT := Setoid.Pack (class cT) cT.
End METRIC.
Notation metricType := Metric.type.
Notation MetricMixin := Metric.Mixin.
Notation MetricType := Metric.pack.
Delimit Scope M_scope with Metric.
Open Scope M_scope.
Canonical Structure Metric.setoidType.
Definition Mrel (m:metricType) : nat → m →s m →s Props :=
  let: Metric.Mixin r _ := Metric.met (Metric.class m) in r.
Implicit Arguments Mrel [m].
Notation "x  $\stackrel{n}{\equiv}$  y" := (Mrel n x y) : M_scope.
    
```

The second half of the definition above is the cruft needed to support a mathematical hierarchy of structures, see [15]. Every structure we define from now on has such a part, but later on we will omit it to save space. Furthermore, we define a new scope for interpreting symbols such that we can reuse symbols for different theories:

The coercion *setoidType* is declared as a canonical structure such that Coq will infer the setoid structure of an ultrametric space when needed. The use of coercions is ubiquitous, e.g., `:>` in record and structure definitions results in implicit coercions.

Notice that in this formulation the ultrametric triangle inequality is equivalent to the transitivity axiom given above.

Next we define the exponential in the category of ultrametric spaces. Classically, a function between ultrametric bisected spaces is non-expansive iff it maps n -equivalent elements to n -equivalent results [10]. Thus we here define a function to be non-expansive if it has this property and is setoid-respecting.

We show that the resulting space of non-expansive functions is one of our metric spaces.

```

Definition nonexpansive (M M':metricType) (f:M → M') : Prop :=
  ∀ (n : nat) (e e' : M), e ≐n e' → f e ≐n f e'.

Record metric_morphism (M M':metricType) := mk_umet_morphism
{ smet_morph :> M →s M'; smet_morph_exp : nonexpansive smet_morph }.

Definition metric_morph_fun M M' : metric_morphism M M' → M → M' :=
  fun f => smet_morph f.

Coercion metric_morph_fun : metric_morphism >-> Funclass.

Lemma metric_morphSP (T T' : metricType) :
  Setoid.axiom (fun (f g:metric_morphism T T') => ∀ x:T, f x ≐ g x).

Canonical Structure morph_metricSMixin (T T':metricType) :=
  Setoid.Mixin (metric_morphSP T T').

Canonical Structure morph_metricSType T T' :=
  Eval hnf in SetoidType (morph_metricSMixin T T').

Lemma metric_morphP (M M':metricType) :
  Metric.axiom (fun n : nat => setoid2_morph
    (f:=fun f g : morph_metricSType M M' => ∀ x : M, f x ≐n g x:Props) ...).

Canonical Structure morph_metricMixin (T T':metricType) :=
  Metric.Mixin (@metric_morphP T T').

Canonical Structure morph_metricType T T' :=
  Eval hnf in MetricType (morph_metricMixin T T').

Infix "→n" := morph_metricType (at level 45, right associativity) : M_scope.
    
```

The use of `>` in the definition of the record of morphisms results in an implicit coercion from morphisms to setoid morphisms, and we further declare an implicit coercion from morphisms to the Coq function space. The use of `Eval hnf` in the definition of the metric space of morphisms evaluates the definition to head normal form, i.e., the structure *Pack* defined above. See [15] for details.

We also define products, sums, options, and so forth, and show that the category of bounded ultrametric spaces is cartesian closed.

Next 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 definition of completeness and is used crucially to show the completeness of the space of finite partial nonexpansive functions (see Section 4.2). 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 complete ultrametric space is then defined as an ultrametric space M equipped with a completion operation $comp : cchain M \rightarrow M$ 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  $M$  ( $comp:cchain\ M \rightarrow M$ ) :=  $\forall\ c, mconverge\ c\ (comp\ c)$ .
Record mixin_of ( $M : metricType$ ) : Type := Mixin
{  $comp : cchain\ M \rightarrow M$ ;  $_ : axiom\ comp$  }.
Record class_of ( $T : Type$ ) : Type := Class
{  $base :> Metric.class\_of\ T$ ;  $mixin :> mixin\_of\ (Metric.Pack\ base\ T)$  }.
Structure type : Type := Pack { $sort :> Type$ ;  $_ : class\_of\ sort$ ;  $_ : Type$ }.
...
Definition setoidType  $cT$  := Setoid.Pack ( $class\ cT$ )  $cT$ .
Coercion metricType  $cT$  := Metric.Pack ( $class\ cT$ )  $cT$ .
End CMETRIC.
Notation cmetricType := CMetric.type.
Notation CMetricMixin := CMetric.Mixin.
Notation CMetricType := CMetric.pack.
Canonical Structure CMetric.setoidType.
Canonical Structure CMetric.metricType.
Definition umet_complete ( $M:cmetricType$ ) :  $cchain\ M \rightarrow M$  :=
CMetric.comp (CMetric.class  $M$ ).

```

Next we define the product, sum, and exponential of bounded complete ultrametric spaces. As the category of bounded complete ultrametric spaces 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 complete ultrametric spaces is Banach's fixed point theorem. Given a non-empty complete ultrametric space M , every contractive endomorphism defined on M has a unique fixed point. We say a function $f : M \rightarrow_n M$ is contractive if for any two points x, y , if $x \stackrel{n}{=} y$ then $f(x) \stackrel{n+1}{=} f(y)$. 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  $M\ N$  ( $f:M \rightarrow_n N$ ) : Prop :=  $\forall\ n\ x\ y, x \stackrel{n}{=} y \rightarrow f\ x \stackrel{n+1}{=} f\ y$ .
Fixpoint iter  $n$  ( $M:cmetricType$ ) ( $f:M \rightarrow M$ ) :=
match  $n$  with |  $O \Rightarrow id$  |  $S\ n \Rightarrow fun\ x \Rightarrow f\ (iter\ n\ f\ x)$  end.
Lemma cfixP ( $M:cmetricType$ ) ( $f:M \rightarrow_n M$ ) ( $C:contractive\ f$ )  $x$  :  $cchainp\ (fun\ n \Rightarrow iter\ n\ f\ x)$ .
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  $M\ f\ C\ x$ ).
Lemma fixp_eq ( $M:cmetricType$ ) ( $f:M \rightarrow_n M$ ) ( $C:contractive\ f$ ) ( $x:M$ ) :  $fixp\ C\ x \equiv f\ (fixp\ C\ x)$ .
Lemma fixp_unique ( $M:cmetricType$ )  $f$  ( $C:contractive\ f$ ) ( $x\ y:M$ ) :  $fixp\ C\ x \equiv fixp\ C\ y$ .
Lemma fixp_ne ( $M:cmetricType$ ) ( $f\ f':M \rightarrow_n M$ ) ( $C:contractive\ f$ ) ( $C':contractive\ f'$ )  $x\ x'\ n$  :
 $f \stackrel{n}{=} f' \rightarrow fixp\ C\ x \stackrel{n}{=} fixp\ C'\ x'$ .

```

The last lemma above 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 , $fix(f)$ is the unique (up to \equiv) fixed point of f .

3 Recursive Domain Equations in M -categories

In this section we outline the formalization of solutions to recursive domain equations in M -categories. The solutions are obtained via Scott's inverse limit

construction, which was adopted to categories of complete metric spaces in [3]. As mentioned in the introduction, our formalization follows a recent generalization to M -categories, that is, categories enriched in complete bounded ultrametric spaces [8].

In earlier work, the first author and colleagues [7] formalized the solution to recursive domain equations in categories of complete partial orders. The present formalization is more general since it applies to arbitrary M -categories and, as shown in [8], solutions to recursive domain equations in Smyth and Plotkin's classical O -categories [21] (hence, in particular, in categories of complete partial orders), can be obtained via the solution in M -categories.

We start by defining the type of M -categories as a record with a type for objects, a bounded complete ultrametric space for morphisms, an identity morphism, a composition operation internalized in the metric space, a terminal object, and proofs of the associated commuting diagrams.

```

Record BaseCat : Type := mk_basecat
{ tcat :> Type;
  tmorph: tcat → tcat → cmetricType;
  tid : ∀ T0, tmorph T0 T0;
  tcomp : ∀ T0 T1 T2, tmorph T1 T2 →ntmorph T0 T1 →ntmorph T0 T2;
  tto : tcat;
  tto_terminalExists : ∀ T, tmorph T tto;
  tcomp_assoc : ∀ T0 T1 T2 T3 (f:tmorph T2 T3) (g:tmorph T1 T2) (h:tmorph T0 T1),
    (tcomp - - - f (tcomp - - - g h)) ≡ (tcomp - - - (tcomp - - - f g) h);
  tid_left : ∀ T0 T1 (f:tmorph T0 T1), tcomp - - - (tid T1) f ≡ f;
  tid_right : ∀ T0 T1 (f:tmorph T0 T1), tcomp - - - f (tid T0) ≡ f;
  tto_terminalUnique : ∀ T (f g : tmorph T tto), f ≡ g }.
    
```

In the rest of this section we assume we are given a category $M : \text{BaseCat}$. We use the section mechanism in Coq to avoid repeating the assumption of M .

Next we give the type of mixed variance locally non-expansive bifunctors on M .

```

Record BiFunctor : Type := mk_functor
{ ob : tcat M → tcat M → tcat M ;
  morph : ∀ (T0 T1 T2 T3 : tcat M),
    (tmorph T1 T0) × (tmorph T2 T3) →n(tmorph (ob T0 T2) (ob T1 T3)) ;
  morph_comp: ∀ T0 T1 T2 T3 T4 T5
    (f:tmorph T4 T1) (g:tmorph T3 T5) (h:tmorph T1 T0) (k:tmorph T2 T3),
    (morph T1 T4 T3 T5 (f,g)) ∘ morph T0 T1 T2 T3 (h, k) ≡ morph - - - (h ∘ f, g ∘ k) ;
  morph_id : ∀ T0 T1, morph - - - (@Tid T0, @Tid T1) ≡ Tid}.
    
```

In the definition above Tid is the identity morphism in M .

The above definition of bifunctors has the same form as the definition of bifunctors in [7], and the same principles for constructing them applies. Hence, for any M -category you build up a small library of combinators to define domain equations. We have done this for the category of preordered bounded complete ultrametric spaces, see Section 4.

We continue by assuming that we are given a bifunctor. We then define increasing Cauchy towers as sequences of section / retraction pairs (s, r) , where $\lim_{n \rightarrow \infty} (s_n \circ r_n) = id$.

Definition *retract* $T_0 T_1$ $(f:tmorph T_0 T_1)$ $(g:tmorph T_1 T_0) := g \circ f \equiv Tid$.

```

Record Tower : Type := mk_tower
{ tobjects : nat → tcat M ;
    
```

```

tmorphisms : ∀ i, tmorph (objects (S i)) (objects i);
tmorphismsI : ∀ i, tmorph (objects i) (objects (S i));
tretract : ∀ i, retract (tmorphismsI i) (tmorphisms i);
tlimitD : ∀ n i, n ≤ i → tmorphismsI i ∘ tmorphisms i ≐ Tid }.

```

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 :> tcat M;
  mccone : ∀ i, tmorph tcone (objects To i);
  mconeCom : ∀ i, tmorphisms To i ∘ mccone (S i) ≐ mccone i }.

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

```

Following [8] we require M -categories to have limits of increasing Cauchy towers. Thence we continue by defining a tower where the n 'th object is the n 'th iteration of the bifunctor starting from the terminal object in M . The first retract $F T T \rightarrow_n T$ is given by the fact that T is the terminal object. We assume there exists a morphisms $T \rightarrow_n F T T$, and that the action of the bifunctor on morphisms is contractive. By contractiveness we then get the limit condition for the tower. It is now a tedious task to verify that the limit of this tower is a solution to the recursive domain equation. In the end we end up with constants:

```

Definition D∞ : tcat M := ...
Definition Fold : tmorph (ob lc D∞ D∞) D∞ := ...
Definition Unfold : tmorph D∞ (ob lc D∞ D∞) := ...
Lemma FU_id : Fold ∘ Unfold ≐ Tid.
Lemma UF_id : Unfold ∘ Fold ≐ Tid.

```

4 Semantics of Language with 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 model of a polymorphic lambda calculus $F^{\mu^!}$ with recursive types, general reference, and a standard call-by-value operational semantics. This model was sketched in [10] and we start out by recalling the basic ideas. It is a simple unary model, which in the end just serves to prove type soundness and which has been chosen to illustrate some of the core challenges of modeling higher-order store. In Section 5 we discuss the prospects of formalizing more advanced models for reasoning about equivalence of programs with mutable abstract data types.

Following realizability style models, types will be modeled as certain predicates on the set V of syntactic $F^{\mu^!}$ values. We refer to the set of such predicates as $UPred(V)$. Since $F^{\mu^!}$ includes dynamic allocation of general references, the model will be a Kripke model, in which semantic types are indexed by worlds

describing which locations are allocated. Because locations may store values of arbitrary types it is also necessary to record the semantic types of the locations in the worlds. Thus we end up with recursive equations of roughly this form:

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

Here locations are modeled as natural numbers, worlds as finite maps from locations to semantic types, and the m on the function space \rightarrow_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

$$W \simeq \mathbb{N} \rightarrow_{\text{fin}} (\tfrac{1}{2}W \rightarrow_m \text{UPred}(V))$$

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

4.1 Syntax of $F^{\mu^!}$

We model the syntax of $F^{\mu^!}$ in Coq via a deep embedding. Types are defined intrinsically as an inductive type family indexed by natural numbers. Binding is encoded in the standard way by de-bruijn indices, and the index of the family captures the type context.

```

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

Notice that the less-than operation in the variable case is a function into the type *bool* and, as equality of bools is decidable, we have proof irrelevance, i.e., there is at most one proof of $i < n$. This implies that Leibniz equality on this type is our standard notion of syntactic equality. We show this and

declare $ExpType\ n$ as an equality type (see [15] for details on equality types).

We separate the syntax of values and expressions into two mutually inductive types and restrict it to ANF (administrative normal form). Effects are sequenced by let expressions and values are embedded into the type of expressions via a val expression. Binding is encoded by de-bruijn indices. We give a standard extrinsically typed version of the syntax as it is simpler to define than an intrinsically typed version.

```

Inductive Value (n:nat) : Set :=
| VAR: nat → Value n | LOC: nat → Value n | INT: nat → Value n
| TLAM: Exp (S n) → Value n | LAM: ExpType n → Exp n → Value n | UNIT : Value n
| P: Value n → Value n → Value n | INL : ExpType n → Value n → Value n
| INR : ExpType n → Value n → Value n | FOLD : ExpType (S n) → Value n → Value n
with Exp (n:nat) : Set :=
| VAL: Value n → Exp n | FST: Value n → Exp n | SND: Value n → Exp n
| OP: (nat → nat → nat) → Value n → Exp n | UNFOLD: Value n → Exp n
| REF: Value n → Exp n | BANG: Value n → Exp n | ASSIGN: Value n → Value n → Exp n
| LET: Exp n → Exp n → Exp n | APP: Value n → Value n → Exp n
| TAPP: Value n → ExpType n → Exp n | CASE: Value n → Exp n → Exp n → Exp n.
    
```

We have also defined 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 ($tsubst$, $etsubst$, $vsubst$) and value substitutions ($csubstV$, $csubstE$) for the different types of syntax. See Appendix B for these definitions.

For the typing derivations we need an encoding of store-typings. We could use the Coq function space suitable restricted, but provable equality would be different from extensional equality and thus we would end up with an unfamiliar typing derivation. Instead we define a structure of finite maps in such a way that extensional equality coincides with provable equality.

We say a type T is comparable if and only if there is a computable linear order on T , and we define a new sort $compType$ (in the sense of [15]) of comparable types. Given $(T : compType)$ we can represent the graphs of finite partial functions with domain T and codomain T' uniquely as lists of $T \times T'$ pairs, where the first projection mapped on the list is sorted and duplicate free.

```

Record FinDom : Type := mk_findom {
  findom_t : seq (T × T');
  findom_P : sorted (map (@fst T T') findom_t) && uniq (map (@fst T T') findom_t) }.
    
```

As $findom_P$ is a decidable predicate (a function into $bool$) we have proof irrelevance, and thus provable equality on $FinDom$ is provable equality of the graph of the function, and by the given property this becomes extensional equality. We define a coercion from $FinDom\ T\ T'$ to $T \rightarrow option(T')$ to regard elements of this type as partial functions.

We define two mutually inductive type families encoding typing derivations (denoted by: $- \vdash - \mid - \vdash_v - : -$, and $- \vdash - \mid - \vdash_e - : -$), a natural step counting operational semantics (denoted by EV), and show standard substitutions theorems and preservation of types.

4.2 Semantics of F^{μ}

We start out by defining the objects of PreCBUlt_{ne} . We define the objects by inheriting a structure of preorders (defined in [7], and changed to fit our packaging structure) 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').
Record mixin_of (T:cmetricType) : Type := Mixin
{ pre :> PreOrd.class_of T; _ : axiom (PreOrd.Ole pre); _ : T }.
Record class_of (T:Type) := Class
{ base :> CMetric.class_of T; mixin :> mixin_of (CMetric.Pack base T) }.
Structure type : Type := Pack {sort :> Type; _ : class_of sort; _ : Type}.

...
Coercion ordType (cT:type) := PreOrd.Pack (class cT) cT.
Coercion cmetricType (cT:type) := CMetric.Pack (class cT) cT.
Definition metricType (cT:type) := Metric.Pack (class cT) cT.
Definition setoidType (cT:type) := Setoid.Pack (class cT) cT.
Definition getelem (cT:type) : cT := let: Mixin _ _ e := mixin (class cT) in e.
End PRECBUMET.

Notation PCMMixin := PreCBUmet.Mixin.
Notation pcmType := PreCBUmet.type.
Notation PCMType := PreCBUmet.pack.

Canonical Structure PreCBUmet.ordType.
Canonical Structure PreCBUmet.cmetricType.
Canonical Structure PreCBUmet.metricType.
Canonical Structure PreCBUmet.setoidType.
    
```

The morphisms of PreCBUlt_{ne} are the morphisms of the underlying metric space that are also monotonic with respect to the preorder:

```

Record pcm_morphism (T T':pcmType) : Type := mk_pcm_morphism
{ pcm_morph :> T →n T'; pcm_monotonic: monotonic pcm_morph }.
    
```

Next we define the standard type operators, sums, products, exponentials, etc, and we show that PreCBUlt_{ne} is an M -category. We show that 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** $\text{downclosed} (p:\text{nat} \times T \rightarrow \text{Prop}) := \forall n k t, k < n \rightarrow p (n,t) \rightarrow p (k,t)$ and we give the Sigma type $\{p:\text{nat} \times T \rightarrow \text{Prop} \mid \text{downclosed } p\}$ (with the single constructor exist) 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 $\text{UPred}(T)$ in PreCBUlt_{ne} as the object of downward closed subsets of $\mathbb{N} \times T$ and refer to the elements as *uniform predicates* (by analogy to complete uniform pers [2]).

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 2.1 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 $fndom_pcmType\ T\ T'$ denote the object of finite partial maps from T to T' , and show that it is functorial in T' via post composition.

Next we give an interpretation of types. We start by interpreting type contexts as products of morphisms in $PreCBUlt_{ne}$ from $\frac{1}{2}W$ to $UPred(V)$ (the subscripts p and n on the function arrows means that it is the function space in $PreCBUlt_{ne}$ and in the underlying metric space, respectively):

Definition $TV := \frac{1}{2}W \rightarrow_p UPred\ (cvalue\ O)$.

Fixpoint $TVal\ (n:nat) : cmetricType :=$
match n **with** $| O \Rightarrow unit_cmetricType\ | S\ n \Rightarrow TVal\ n \times TV\ \text{end}$.

Lemma $less_nil\ j\ (P:j < 0) : False$.

Fixpoint $pick\ n\ j : (j < n) \rightarrow TVal\ n \rightarrow_n TV :=$
match n **as** n_1, j **as** j_1 **return** $j_1 < n_1 \rightarrow TVal\ n_1 \rightarrow_n TV$ **with**
 $| O, _ \Rightarrow \text{fun } F \Rightarrow \text{match } less_nil\ F \text{ with } \text{end}$
 $| S\ n, S\ j \Rightarrow \text{fun } F \Rightarrow @pick\ n\ j\ F \circ \pi_1$
 $| S\ n, O \Rightarrow \text{fun } F \Rightarrow \pi_2$
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 $upred_int_down : downclosed$
 $(\text{fun } kt \Rightarrow \text{match } snd\ kt : cvalue\ O \text{ with } | (CValue\ (INT\ i)\ _) \Rightarrow True\ | _ \Rightarrow False\ \text{end})$.

Definition $upred_int : UPred\ (cvalue\ 0) := exist\ (@downclosed\ _) _ upred_int_down$.

For more complicated inductive type constructors we try to use the letter R for the meaning of the sub-component(s). For instance, the meaning of a recursive type $Mu\ t$ is a fixed point and we use R for the meaning of t and P for the meaning of the argument of the functional of which we will take a fixed point. We then show the necessary properties of the definition and end up with a suitable constant $upred_mu$.

Lemma $upred_mu_down\ n\ (R:TVal\ n.+1 \rightarrow_n \frac{1}{2}W \rightarrow_p UPred\ (cvalue\ O))\ (s:TVal\ n)$
 $(P:(\frac{1}{2}W \rightarrow_p UPred\ (cvalue\ O)) \times \frac{1}{2}W) :$
 $downclosed\ (\text{fun } kt \Rightarrow \text{let: } CValue\ v'\ p' := snd\ kt \text{ in}$
 $\text{match } fst\ kt, v' \text{ as } v0 \text{ return } closedV\ v0 \rightarrow Prop \text{ with}$
 $| O, FOLD\ t\ v \Rightarrow \text{fun } X \Rightarrow True$
 $| S\ k, FOLD\ t\ v \Rightarrow \text{fun } X \Rightarrow upred_fun\ (R\ ((s, (fst\ P)))\ (snd\ P))\ (k, CValue\ v\ X)$
 $| _, _ \Rightarrow \text{fun } X \Rightarrow False$
 $\text{end } p')$.

Definition $upred_mut\ n\ R\ s\ P : UPred\ (cvalue\ O) :=$
 $exist\ (@downclosed\ _) - (@upred_mu_down\ n\ R\ s\ P).$

Definition $upred_mu\ n\ R :$
 $TVal\ n \rightarrow_n (\frac{1}{2} W \rightarrow_p UPred\ (cvalue\ O)) \rightarrow_c (\frac{1}{2} W \rightarrow_p UPred\ (cvalue\ O)) := \dots$

In the definition above the \rightarrow_c arrow is the object of contractive morphisms.

The following definitions of sets involves suspended computations and thus we start by defining a relation between heaps of closed values and worlds.

Definition $heap_world\ k\ (h:cheap)\ (w:W) := \forall j, j < k \rightarrow dom\ (heap\ h) = dom\ (Unfold\ w) \wedge$
 $\forall l\ (I: l \in dom\ (Unfold\ w))\ (I': l \in dom\ (heap\ h)),$
 $upred_fun\ (indom_app\ I\ w)\ (j, CValue\ (indom_app\ I')\ (heap_cl\ I')).$

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, e.g., [1].

Definition $IExp\ (f:TV)\ (k:nat)\ (e:cexpression\ 0)\ (w:W) :=$
 $\forall j, (j \leq k) \% N \rightarrow \forall (h\ h':cheap)\ v\ (D:EV\ j\ e\ h\ v\ h'), heap_world\ k\ h\ w \rightarrow$
 $\exists w':W, w \sqsubseteq w' \wedge heap_world\ (k-j)\ h'\ w' \wedge upred_fun\ (f\ w')\ (k-j, CValue\ v \dots).$

We then continue with the definitions for the remaining type constructors:

Lemma $upred_arrow_down\ n\ (R_0\ R_1:TVal\ n \rightarrow_n \frac{1}{2} W \rightarrow_p UPred\ (cvalue\ 0))\ (s:TVal\ n)\ (w:\frac{1}{2} W) :$

$downclosed\ (fun\ kt \Rightarrow let: (k,v) := kt\ in$
 $match\ v\ with$
 $| CValue\ (LAM\ t'\ e)\ p \Rightarrow \forall w'\ j\ (va:cvalue\ O), w \sqsubseteq w' \rightarrow (j \leq k) \% N \rightarrow$
 $upred_fun\ (R_0\ s\ w')\ (j, va) \rightarrow IExp\ (R_1\ s)\ j\ (csubstE\ [::\ va]\ e)\ w'$
 $| _ \Rightarrow False\ end).$

Definition $upred_arrowt\ n\ R_0\ R_1\ s\ w : UPred\ (cvalue\ 0) :=$
 $exist\ (@downclosed\ _) - (@upred_arrow_down\ n\ R_0\ R_1\ s\ w).$

Definition $upred_arrow\ n\ (R_0\ R_1:TVal\ n \rightarrow_n \frac{1}{2} W \rightarrow_p UPred\ (cvalue\ 0)) :$
 $(TVal\ n \rightarrow_n \frac{1}{2} W \rightarrow_p UPred\ (cvalue\ 0)).$

Lemma $upred_all_down\ n\ (R:TVal\ n.+1 \rightarrow_n \frac{1}{2} W \rightarrow_p UPred\ (cvalue\ 0))\ (s:TVal\ n \times \frac{1}{2} W) :$

$downclosed\ (fun\ kt \Rightarrow let: (k,v) := kt\ in$
 $match\ @cval\ O\ v\ as\ v0\ return\ closedV\ v0 \rightarrow Prop\ with$
 $| TLAM\ v \Rightarrow fun\ C \Rightarrow \forall (t:ExpType\ 0)\ (r:TV)\ (w':W)\ j, (j \leq k) \% N \rightarrow snd\ s \sqsubseteq w' \rightarrow$
 $(IExp\ (R\ (fst\ s,r))\ j\ (CExp\ (etsubst\ [::\ t]\ v)\ (@closed_tsubst\ _ \ v\ C\ _ [::\ t]))\ w')$
 $| _ \Rightarrow fun\ _ \Rightarrow False$
 $end\ (cvalueP\ v)).$

Definition $upred_allt\ n\ R\ s : UPred\ (cvalue\ 0) :=$
 $exist\ (@downclosed\ _) - (@upred_all_down\ n\ R\ s).$

Definition $upred_all\ n\ (R : TVal\ n.+1 \rightarrow_n \frac{1}{2} W \rightarrow_p UPred\ (cvalue\ 0)) :$
 $TVal\ n \rightarrow_n \frac{1}{2} W \rightarrow_p UPred\ (cvalue\ 0).$

Lemma $upred_ref_down\ n\ (R : TVal\ n \rightarrow_n \frac{1}{2} W \rightarrow_p UPred\ (cvalue\ 0))\ (s:TVal\ n)\ (w: \frac{1}{2} W) :$

$downclosed\ (fun\ kt \Rightarrow let: (k,v) := kt\ in$
 $match\ k,(v:cvalue\ O)\ with$
 $| O, CValue\ (LOC\ l)\ _ \Rightarrow True$
 $| S\ k, CValue\ (LOC\ l)\ _ \Rightarrow \{P:l \in dom\ (Unfold\ w) \ \&\ (indom_app\ P)\}^{k,+1}\ R\ s\}$
 $| _, _ \Rightarrow False$
 $end).$

Definition $upred_reft\ n\ R\ s\ w : UPred\ (cvalue\ 0) :=$
 $exist\ (@downclosed\ _) - (@upred_ref_down\ n\ R\ s\ w).$

Definition $upred_ref\ n\ (R : TVal\ n \rightarrow_n \frac{1}{2} W \rightarrow_p UPred\ (cvalue\ 0)) :$
 $TVal\ n \rightarrow_n \frac{1}{2} W \rightarrow_p UPred\ (cvalue\ 0).$

Now we can give the value interpretation of types. The constants $Premet.Pcomp$ and $Pprod_fun$ are internalizations of the composition operation and the universal morphism of the product in $PreCBUlt_{ne}$, respectively. **Fixpoint** $IVal\ n\ (t:ExpType\ n) : TVal\ n \rightarrow_n TV :=$

```

match t with
| TVar n J ⇒ pick J
| Int ⇒ mconst _ (pconst _ upred_int)
| Unit ⇒ mconst _ (pconst _ upred_unit)
| t × t' ⇒ (Premet.Pcomp - - - upred_product ∘ Pprod_fun - - -) ∘⟨ IVal t, IVal t' ⟩
| t + t' ⇒ (Premet.Pcomp - - - upred_sum ∘ Pprod_fun - - -) ∘⟨ IVal t, IVal t' ⟩
| Mu t ⇒ FIXP ∘ upred_mu (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 →n( $\frac{1}{2}$ W →pUPred (Prod (cvalue 0) (size e)))%PCM
:=
match e as e0 return TVal n →n( $\frac{1}{2}$ W →pUPred (Prod (cvalue 0) (size e0)))%PCM with
| nil ⇒ mconst _ (pconst _ (upred_empty unit))
| t::te ⇒ (Premet.Pcomp - - - 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 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:ExpType n) :=
  ∀ k (s:TVal n) (ts:Prod (ExpType 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) (vsubst (Prod_subst ts) v)).

```

```

Definition ERel n (E:TypeEnv n) (Se:StoreType n) (e:Exp n) (t:ExpType n) :=
  ∀ k (s:TVal n) (ts:Prod (ExpType 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:ExpType n) s m s' (a:seq (ExpType 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).

```

5 Discussion

We have presented a Coq formalization of solutions to recursive domain equations in M -categories and shown how to apply it to give semantics to a programming language with higher-order store.

The semantic model presented in Section 4 is a simple unary model, but we believe it shows the efficacy of our approach. Now it should be possible to formalize state-of-the-art relational models for reasoning about equivalence of mutable abstract data types [1,13]. The techniques can also be used to show soundness of separation logics for languages with higher order store [19,20,10] or with storable locks [16]. Such models can either be built using step-indexing directly over the operational semantics, as we did in Section 4, or by using a standard cpo-based semantics of the programming language, as in [9,19,20].

To formalize the latter approach one may use the formalization of domain theory based on complete partial orders in [7].

Several groups of researchers have formalized parts of classical or synthetic domain theory, see [7] and the discussion therein. We are not aware of any earlier formalizations of ultrametric domain theory.

Other related work includes the formalization of step-indexed models via indirection theory by Hobor et. al. [16,4]. The main difference is that we use the metric approach which applies both to step-indexed models but also (as mentioned above) to models based on standard domain theory; see [10] for a detailed explanation of how the metric approach can be specialized to indirection theory. With regard to the approaches to formalization, one difference is that in [4] a few axioms are assumed in addition to Coq’s standard type theory, in particular extensionality of functions and propositions, and also dependent unique choice, whereas in our approach we do not assume any additional axioms, but rather work with setoids, etc., to ensure that the structures we work with have the right extensionality properties. Another difference is that we have strived to follow the ideas of [15] to (1) support a mathematical hierarchy of structures and (2) to make reasoning about equality in Coq easier (hence our focus on using decidable properties, using *bool* rather than *Prop*). Indeed, the size of the formalization appears to be reasonable: the two tables below show for each file used in the formalization how many lines of specifications (definitions and lemmas) and how many lines of proof there are. The table on the left contains the files used in the formalization of ultrametric spaces and solutions to recursive domain equations in M -categories. The table on the right shows the files used in the application (and here we see that there is, as one might expect, quite a bit of work involved in the deep embedding of the syntax of the rich programming language we have considered).

Framework		
Spec	Proof	Filename
149	110	NSetoid.v
230	512	Finmap.v
337	446	MetricCore.v
189	473	MetricRec.v
683	472	PredomCore.v
287	246	PredomProd.v
1776	2261	Total

Application		
Spec	Proof	Filename
544	1692	Syntax.v
60	318	Operational.v
231	867	Sem.v
835	2877	Total

References

- [1] Ahmed, A., D. Dreyer and A. Rossberg, *State-dependent representation independence*, in: *POPL*, 2009.
- [2] Amadio, R. M. and P.-L. Curien, “Domains and Lambda-Calculi,” Cambridge Tracts in Theoretical Computer Science **46**, Cambridge University Press, Cambridge, 1998.

- [3] America, P. and J. J. M. M. Rutten, *Solving reflexive domain equations in a category of complete metric spaces*, J. Comput. Syst. Sci. **39** (1989), pp. 343–375.
- [4] Appel, A., R. Dockins and A. Hobor, *Mechanized semantic library*, <http://msl.cs.princeton.edu/> (2009).
- [5] Appel, A. W. and D. A. McAllester, *An indexed model of recursive types for foundational proof-carrying code*, ACM Trans. Program. Lang. Syst. **23** (2001), pp. 657–683.
- [6] Barthe, G., V. Capretta and O. Pons, *Setoids in type theory*, J. Funct. Program. **13** (2003), pp. 261–293.
- [7] Benton, N., A. Kennedy and C. Varming, *Some domain theory and denotational semantics in Coq*, in: *TPHOLS*, 2009, pp. 115–130.
- [8] Birkedal, L., K. Støvring and J. Thamsborg, *The category-theoretic solution of recursive metric-space quations*, Technical Report ITU-2009-119, IT University of Copenhagen (2009).
- [9] Birkedal, L., K. Støvring and J. Thamsborg, *Realizability semantics of parametric polymorphism, general references, and recursive types*, in: *Proceedings of FOSSACS*, number 5504 in LNCS, 2009, pp. 456–470.
- [10] Birkedal, L., K. Støvring and J. Thamsborg, *Kripke models over recursively defined metric worlds: Steps and domains*, 2010, submitted for publication. Available at: <http://itu.dk/people/birkedal/papers/krimrd.pdf>.
- [11] de Bakker, J. and E. de Vink, “Control flow semantics,” MIT Press, Cambridge, MA, USA, 1996.
- [12] de Bakker, J. W. and J. Zucker, *Processes and the denotational semantics of concurrency*, Information and Control **54** (1982), pp. 70–120.
- [13] Dreyer, D., G. Neis, A. Rossberg and L. Birkedal, *A relational modal logic for higher-order stateful adts*, in: *POPL*, 2010.
- [14] Escardó, M., *A metric model of PCF* (1998), laboratory for Foundations of Computer Science, University of Edinburgh. Presented at the Workshop on Realizability Semantics and Applications, June 30–July 1, 1999 (associated to the Federated Logic Conference, held in Trento, June 29–July 12, 1999).
- [15] Garillot, F., G. Gonthier, A. Mahboubi and L. Rideau, *Packaging mathematical structures*, in: *TPHOLS '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (2009), pp. 327–342.
- [16] Hobor, A., R. Dockins and A. W. Appel, *A theory of indirection via approximation*, in: *37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010)*, 2010, p. to appear.
URL <http://msl.cs.princeton.edu/indirection.pdf>
- [17] Hofmann, M., “Extensional Concepts in Intensional Type Theory,” Ph.D. thesis, LFCS, Univ. of Edinburgh (1995).
- [18] McBride, C. and J. McKinna, *The view from the left*, J. Funct. Program. **14** (2004), pp. 69–111.
- [19] Schwinghammer, J., L. Birkedal, B. Reus and H. Yang, *Nested Hoare triples and frame rules for higher-order store*, in: *Proceedings of CSL*, number 5771 in LNCS, 2009, pp. 440–454.
- [20] Schwinghammer, J., H. Yang, L. Birkedal and F. P. B. Reus, *A semantic foundation for hidden state*, in: *FOSSACS*, 2010.
- [21] Smyth, M. and G. Plotkin, *The category-theoretic solution of recursive domain equations*, SIAM Journal on Computing **11** (1982), pp. 761–783.
- [22] van Breugel, F., *An introduction to metric semantics: Operational and denotational semantics for programming and specification languages*, Theoretical Computer Science **258** (2001), pp. 1–98.

A Setoids

A.1 Setoids

A setoid is a type, a binary relation, and a proof that the relation is an equivalence relation.

```

Module Setoid.
Definition axiom (T:Type) (e:T → T → Prop) := equiv _ e.
Record mixin_of (T:Type) : Type := Mixin
{ set_eq : T → T → Prop;
  set_equiv : axiom set_eq
}.
Notation class_of := mixin_of (only parsing).
Structure type : 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 repack cT : _ → Type → type := let k T c p := p c in unpack k cT.
Definition pack T c := @Pack T c T.
End Setoid.

Notation setoidType := Setoid.type.
Notation SetoidMixin := Setoid.Mixin.
Notation SetoidType := Setoid.pack.

```

The second half of the definition above is the cruft needed to support a mathematical hierarchy of structures, see [15]. Every structure we define from now on has such a part, but later on we will omit it to save space. Furthermore, we define a new scope for interpreting symbols such that we can reuse symbols for different theories:

```

Definition tset_eq := fun T => Setoid.set_eq (Setoid.class T).
Delimit Scope S_scope with SET.
Open Scope S_scope.
Infix "≡" := tset_eq (at level 70) : S_scope.

```

Below we define the exponential in Setoid, namely the setoid of equivalence respecting functions (\rightarrow_s).

```

Definition setoid_respect (T T':setoidType) (f: T → T') :=
  ∀ x y, x ≡ y → (f x) ≡ (f y).
Record setoid_morphism (T T':setoidType) := mk_setoid_morphism
{ tset_morph :> T → T';
  tset_morph_respect : setoid_respect tset_morph }.
Lemma setoid_morphP (T T' : setoidType) :
  Setoid.axiom (fun (f g:setoid_morphism T T') => ∀ x:T, (f x) ≡ (g x)).
Canonical Structure morph_setoidMixin (T T':setoidType) :=
  Setoid.Mixin (setoid_morphP T T').
Canonical Structure morph_setoidType T T' :=
  Eval hnf in SetoidType (morph_setoidMixin T T').
Infix "→s" := morph_setoidType (at level 45, right associativity) : S_scope.

```

B Syntax of $F^{\mu!}$

Fixpoint $tshiftR\ k\ m\ n\ (t:ExpType\ n) : ExpType\ (addn\ n\ m) :=$
match t **with**
 $| TVar\ i\ l \Rightarrow$ **if** $i < k$ **then** $@TVar\ _ i\ (ssrnat.leq_trans\ l\ (leq_addr\ _ _))$ **else** $@TVar\ _ (i+m)$
 $(leq_add\ l\ (leqnn\ _)) : (i+m < n + m)$
 $| Int \Rightarrow Int$
 $| Unit \Rightarrow Unit$
 $| Product\ t\ t' \Rightarrow Product\ (tshiftR\ k\ m\ t)\ (tshiftR\ k\ m\ t')$
 $t + t' \Rightarrow +\ (tshiftR\ k\ m\ t)\ (tshiftR\ k\ m\ t')$
 $| Mu\ t \Rightarrow Mu\ (tshiftR\ (S\ k)\ m\ t)$
 $All\ t \Rightarrow All\ (tshiftR\ (S\ k)\ m\ t)$
 $| Arrow\ t\ t' \Rightarrow Arrow\ (tshiftR\ k\ m\ t)\ (tshiftR\ k\ m\ t')$
 $| Ref\ t \Rightarrow Ref\ (tshiftR\ k\ m\ t)$
end.

Definition $tshiftL\ k\ m\ n\ (t:ExpType\ n) : ExpType\ (m+n) := eq_rect\ _ ExpType\ (tshiftR\ k\ m\ t)$
 $_ (addnC\ n\ m).$

Fixpoint $tsubst\ n\ (s:seq\ (ExpType\ n))\ m\ (t:ExpType\ m) : ExpType\ n :=$
match t **with**
 $| TVar\ i\ l \Rightarrow nth\ Unit\ s\ i$
 $| Int \Rightarrow Int$
 $| Unit \Rightarrow Unit$
 $| Product\ t\ t' \Rightarrow Product\ (tsubst\ s\ t)\ (tsubst\ s\ t')$
 $t + t' \Rightarrow +\ (tsubst\ s\ t)\ (tsubst\ s\ t')$
 $| Mu\ t \Rightarrow Mu\ (tsubst\ (TVar\ (ltn0Sn\ _)) :: (map\ (@tshiftL\ 0\ 1\ n)\ s))\ t)$
 $All\ t \Rightarrow All\ (tsubst\ (TVar\ (ltn0Sn\ _)) :: (map\ (@tshiftL\ 0\ 1\ n)\ s))\ t)$
 $| Arrow\ t\ t' \Rightarrow Arrow\ (tsubst\ s\ t)\ (tsubst\ s\ t')$
 $| Ref\ t \Rightarrow Ref\ (tsubst\ s\ t)$
end.

Definition $TypeEnv\ n := seq\ (ExpType\ n).$

Definition $StoreType\ i := FinDom\ [compType\ of\ nat]\ (ExpType\ i).$

Inductive $VTyping\ (i:nat)\ (env:TypeEnv\ i)\ (se:StoreType\ i)\ (t:ExpType\ i) : Value\ i \rightarrow Type :=$
 $| TvVAR: \forall\ m,\ nth_error\ env\ m = Some\ t \rightarrow i \vdash env \mid se \vdash_v (VAR\ m) : t$
 $| TvLOC: \forall\ t', l, se\ l = Some\ t' \rightarrow t = Ref\ t' \rightarrow i \vdash env \mid se \vdash_v LOC\ l : t$
 $| TvINT: \forall\ n, t = Int \rightarrow i \vdash env \mid se \vdash_v (INT\ n) : t$
 $| TvTLAM: \forall\ e\ t', t = All\ t' \rightarrow i.+1 \vdash map\ (@tshiftL\ 0\ 1\ i)\ env \mid post_compt\ (@tshiftL\ 0\ 1\ i)$
 $se \vdash_e e : t' \rightarrow i \vdash env \mid se \vdash_v (TLAM\ e) : t$
 $| TvLAMBDA: \forall\ a\ b\ body, t = a \rightarrow b \rightarrow i \vdash a :: env \mid se \vdash_e body : b \rightarrow$
 $i \vdash env \mid se \vdash_v (LAM\ a\ body) : t$
 $| TvUNIT: t = Unit \rightarrow i \vdash env \mid se \vdash_v UNIT : t$
 $| TvP: \forall\ a\ b\ e1\ e2, t = a \times b \rightarrow i \vdash env \mid se \vdash_v e1 : a \rightarrow$
 $i \vdash env \mid se \vdash_v e2 : b \rightarrow i \vdash env \mid se \vdash_v (P\ e1\ e2) : t$
 $| TvINL: \forall\ a\ b\ e, i \vdash env \mid se \vdash_v e : a \rightarrow t = Sum\ a\ b \rightarrow i \vdash env \mid se \vdash_v (INL\ b\ e) : t$
 $| TvINR: \forall\ a\ b\ e, i \vdash env \mid se \vdash_v e : b \rightarrow t = Sum\ a\ b \rightarrow i \vdash env \mid se \vdash_v (INR\ a\ e) : t$
 $| TvFOLD: \forall\ a\ e, i \vdash env \mid se \vdash_v e : tsubst\ (Mu\ a::idsub\ 0\ i)\ a \rightarrow t = Mu\ a \rightarrow i \vdash env \mid se \vdash_v$
 $FOLD\ a\ e : t$

where $"i \vdash env \mid se \vdash_v exp : type" := (@VTyping\ i\ env\ se\ type\ exp)$

with $ETyping\ (i:nat)\ (env:TypeEnv\ i)\ (se:StoreType\ i)\ (t:ExpType\ i) : Exp\ i \rightarrow Type :=$
 $| TeVAL: \forall\ v, i \vdash env \mid se \vdash_v v : t \rightarrow i \vdash env \mid se \vdash_e (VAL\ v) :: t$
 $| TeLET: \forall\ e\ e', i \vdash env \mid se \vdash_e e' : t' \rightarrow i \vdash t' :: env \mid se \vdash_e e : t \rightarrow$
 $i \vdash env \mid se \vdash_e (LET\ e'\ e) : t$
 $| TvFST: \forall\ b\ e, i \vdash env \mid se \vdash_v e : (t \times b) \rightarrow i \vdash env \mid se \vdash_e (FST\ e) : t$
 $| TvSND: \forall\ a\ e, i \vdash env \mid se \vdash_v e : (a \times t) \rightarrow i \vdash env \mid se \vdash_e (SND\ e) : t$
 $| TvOP: \forall\ op\ exp, t = Int \rightarrow i \vdash env \mid se \vdash_v exp : Product\ Int\ Int \rightarrow i \vdash env \mid se \vdash_e (OP\ op$
 $exp) : t$
 $| TvUNFOLD: \forall\ a\ e, i \vdash env \mid se \vdash_v e : Mu\ a \rightarrow t = tsubst\ (Mu\ a::idsub\ 0\ i)\ a \rightarrow i \vdash env \mid se$
 $\vdash_e UNFOLD\ e : t$
 $| TvREF: \forall\ e\ a, i \vdash env \mid se \vdash_v e : a \rightarrow t = Ref\ a \rightarrow i \vdash env \mid se \vdash_e REF\ e : t$
 $| TvBANG: \forall\ e, i \vdash env \mid se \vdash_v e : Ref\ t \rightarrow i \vdash env \mid se \vdash_e BANG\ e : t$
 $| TvASSIGN: \forall\ l\ e\ a, i \vdash env \mid se \vdash_v l : Ref\ a \rightarrow i \vdash env \mid se \vdash_v e : a \rightarrow t = Unit \rightarrow i \vdash env \mid$
 $se \vdash_e ASSIGN\ l\ e : t$
 $| TeAPP: \forall\ a\ rator\ rand, i \vdash env \mid se \vdash_v rator : a \rightarrow t \rightarrow$
 $i \vdash env \mid se \vdash_v rand : a \rightarrow$
 $i \vdash env \mid se \vdash_e (APP\ rator\ rand) : t$
 $| TeTAPP: \forall\ e\ b\ a, i \vdash env \mid se \vdash_v e : All\ b \rightarrow t = tsubst\ (a::idsub\ 0\ i)\ b \rightarrow i \vdash env \mid se \vdash_e$
 $TAPP\ e\ a : t$

| *TeCASE*: $\forall v e e' a b, i \vdash env \mid se \vdash_v v : Sum a b \rightarrow i \vdash a :: env \mid se \vdash_e e : t \rightarrow i \vdash b :: env \mid se \vdash_e e' : t \rightarrow$

$i \vdash env \mid se \vdash_e CASE v e e' : t$
 where "i $\vdash env \mid se \vdash_e exp : type$ " := ($@ETyping i env se type exp$).

Fixpoint *shiftV* $k i n (v:Value n) : Value n :=$

```

match v with
| VAR j  $\Rightarrow$  if j j k then VAR j else VAR (j+i)
| LOC j  $\Rightarrow$  LOC j
| INT j  $\Rightarrow$  INT j
| TLAM e  $\Rightarrow$  TLAM (shiftE k i e)
| LAM t e  $\Rightarrow$  LAM t (shiftE k.+1 i e)
| UNIT  $\Rightarrow$  UNIT
| P v0 v1  $\Rightarrow$  P (shiftV k i v0) (shiftV k i v1)
| INL t v  $\Rightarrow$  INL t (shiftV k i v)
| INR t v  $\Rightarrow$  INR t (shiftV k i v)
| FOLD t v  $\Rightarrow$  FOLD t (shiftV k i v)

```

end

with *shiftE* $k i n (e:Exp n) : Exp n :=$

```

match e with
| VAL v  $\Rightarrow$  VAL (shiftV k i v)
| LET e0 e1  $\Rightarrow$  LET (shiftE k i e0) (shiftE k.+1 i e1)
| FST v  $\Rightarrow$  FST (shiftV k i v)
| SND v  $\Rightarrow$  SND (shiftV k i v)
| OP op v  $\Rightarrow$  OP op (shiftV k i v)
| UNFOLD v  $\Rightarrow$  UNFOLD (shiftV k i v)
| REF v  $\Rightarrow$  REF (shiftV k i v)
| BANG v  $\Rightarrow$  BANG (shiftV k i v)
| ASSIGN v0 v1  $\Rightarrow$  ASSIGN (shiftV k i v0) (shiftV k i v1)
| APP v0 v1  $\Rightarrow$  APP (shiftV k i v0) (shiftV k i v1)
| TAPP v t  $\Rightarrow$  TAPP (shiftV k i v) t
| CASE v e0 e1  $\Rightarrow$  CASE (shiftV k i v) (shiftE k.+1 i e0) (shiftE k.+1 i e1)

```

end.

Fixpoint *substV* $i (s:seq (Value i)) (v:Value i) : Value i :=$

```

match v with
| VAR j  $\Rightarrow$  nth (@UNIT -) s j
| LOC j  $\Rightarrow$  LOC j
| INT j  $\Rightarrow$  INT j
| TLAM e  $\Rightarrow$  TLAM (substE (map (@vsubst - (drop 1 (idsub 0 i.+1)) -) s) e)
| LAM t e  $\Rightarrow$  LAM t (substE (VAR 0 :: (map (@shiftV 0 1 -) s)) e)
| UNIT  $\Rightarrow$  UNIT
| P v0 v1  $\Rightarrow$  P (substV s v0) (substV s v1)
| INL t v  $\Rightarrow$  INL t (substV s v)
| INR t v  $\Rightarrow$  INR t (substV s v)
| FOLD t v  $\Rightarrow$  FOLD t (substV s v)

```

end

with *substE* $i (s:seq (Value i)) (e:Exp i) : Exp i :=$

```

match e with
| VAL v  $\Rightarrow$  VAL (substV s v)
| LET e0 e1  $\Rightarrow$  LET (substE s e0) (substE (VAR 0 :: map (@shiftV 0 1 -) s) e1)
| FST v  $\Rightarrow$  FST (substV s v)
| SND v  $\Rightarrow$  SND (substV s v)
| OP op v  $\Rightarrow$  OP op (substV s v)
| UNFOLD v  $\Rightarrow$  UNFOLD (substV s v)
| REF v  $\Rightarrow$  REF (substV s v)
| BANG v  $\Rightarrow$  BANG (substV s v)
| ASSIGN v0 v1  $\Rightarrow$  ASSIGN (substV s v0) (substV s v1)
| APP v0 v1  $\Rightarrow$  APP (substV s v0) (substV s v1)
| TAPP v t  $\Rightarrow$  TAPP (substV s v) t
| CASE v e0 e1  $\Rightarrow$  CASE (substV s v) (substE (VAR 0 :: map (@shiftV 0 1 -) s) e0) (substE (VAR 0 :: map (@shiftV 0 1 -) s) e1)

```

end.

Definition *Heap* := *FinDom* [*compType* of *nat*] (*Value O*).

Inductive *EV* : *nat* \rightarrow (*Exp 0*) \rightarrow *Heap* \rightarrow *Value O* \rightarrow *Heap* \rightarrow **Type** :=

$| \text{EvVAL } h \ v : \text{EV } O \ (\text{VAL } v) \ h \ v \ h$
 $| \text{EvFST } h \ v0 \ v1 : \text{EV } O \ (\text{coqconstructorrefSyntax.FSTFST } (P \ v0 \ v1)) \ h \ v0 \ h$
 $| \text{EvSND } h \ v0 \ v1 : \text{EV } O \ (\text{SND } (P \ v0 \ v1)) \ h \ v1 \ h$
 $| \text{EvOP } h \ op \ n0 \ n1 : \text{EV } O \ (\text{OP } op \ (P \ (\text{INT } n0) \ (\text{INT } n1))) \ h \ (\text{INT } (op \ n0 \ n1)) \ h$
 $| \text{EvUNFOLD } h \ v \ t : \text{EV } 1 \ (\text{UNFOLD } (\text{FOLD } t \ v)) \ h \ v \ h$
 $| \text{EvREF } (h:\text{Heap}) \ v \ (l:\text{nat}) : l \ \text{“notin dom } h \rightarrow \text{EV } 1 \ (\text{REF } v) \ h \ (\text{LOC } l) \ (\text{updMap } l \ v \ h)$
 $| \text{EvBANG } (h:\text{Heap}) \ v \ (l:\text{nat}) : h \ l = \text{Some } v \rightarrow \text{EV } 1 \ (\text{BANG } (\text{LOC } l)) \ h \ v \ h$
 $| \text{EvASSIGN } (h:\text{Heap}) \ v \ (l:\text{nat}) : h \ l \rightarrow \text{EV } 1 \ (\text{ASSIGN } (\text{LOC } l) \ v) \ h \ \text{UNIT } (\text{updMap } l \ v \ h)$
 $| \text{EvLET } h \ n0 \ e0 \ v0 \ h0 \ n1 \ e1 \ v \ h1 : \text{EV } n0 \ e0 \ h \ v0 \ h0 \rightarrow \text{EV } n1 \ (\text{substE } [:: \ v0] \ e1) \ h0 \ v \ h1$
 $\rightarrow \text{EV } (n0 + n1) \ (\text{LET } e0 \ e1) \ h \ v \ h1$
 $| \text{EvAPP } h \ n \ t0 \ e \ v0 \ v \ h0 : \text{EV } n \ (\text{substE } (v :: \ \text{nil}) \ e) \ h \ v0 \ h0 \rightarrow \text{EV } n \ (\text{APP } (\text{LAM } t0 \ e) \ v)$
 $h \ v0 \ h0$

 $| \text{EvTAPP } h \ n \ e \ t \ v0 \ h0 : \text{EV } n \ (\text{etsubst } [:: \ t] \ e) \ h \ v0 \ h0 \rightarrow \text{EV } n \ (\text{TAPP } (\text{TLAM } e) \ t) \ h \ v0 \ h0$
 $| \text{EvCASEL } h \ n \ t \ v \ e0 \ e1 \ v0 \ h0 : \text{EV } n \ (\text{substE } [:: \ v] \ e0) \ h \ v0 \ h0 \rightarrow \text{EV } n \ (\text{CASE } (\text{INL } t \ v)$
 $e0 \ e1) \ h \ v0 \ h0$
 $| \text{EvCASER } h \ n \ t \ v \ e0 \ e1 \ v0 \ h0 : \text{EV } n \ (\text{substE } [:: \ v] \ e1) \ h \ v0 \ h0 \rightarrow \text{EV } n \ (\text{CASE } (\text{INR } t \ v)$
 $e0 \ e1) \ h \ v0 \ h0.$