ModuRes: a Coq Library for Modular Reasoning about Concurrent Higher-Order Imperative Programming Languages

Filip Sieczkowski, Aleš Bizjak, and Lars Birkedal

Department of Computer Science, Aarhus University
{filips,abizjak,birkedal}@cs.au.dk

Abstract. It is well-known that it is challenging to build semantic models of type systems or logics for reasoning about concurrent higher-order imperative programming languages. One of the key challenges is that such semantic models often involve constructing solutions to certain kinds of recursive domain equations, which in practice has been a barrier to formalization efforts. Here we present the ModuRes Coq library, which provides an easy way to solve such equations. We show how the library can be used to construct models of type systems and logics for reasoning about concurrent higher-order imperative programming languages.

1 Introduction

In recent years we have seen a marked progress in techniques for reasoning about higher-order, effectful programming languages. However, many of the resulting models and logics have eschewed formal verification. The main reason seems to be the rising complexity of these theories, as well as the increasing use of sophisticated mathematical structures, which impose a substantial barrier to entry for potential formalization efforts. One of the crucial features that is difficult to model is circularity, in its various shapes. Its prototypical form stems from higher-order store, where we can store functions and references as well as first-order data, but circularities can arise in many other ways. One that is commonly encountered in program logics for concurrency, for instance, is shared invariants, which can be accessed by any one thread of computation — provided that the access preserves the invariant. Since the state that a particular invariant describes could well refer to other invariants, a circularity arises. This circularity is quite similar in nature to the one that arises when one attempts to interpret types of higher-order references: the semantic description of the store has to include the interpretation of the type for each location — but the interpretation of the type itself depends on the description of the store.

In this paper we present the ModuRes Coq library[1], which provides an easy way to solve these kinds of circularities, and help use the solutions to build models of programming languages and program logics.

[1] Available at `http://cs.au.dk/~birke/modures/tutorial`.

Example: Consider a simply-typed lambda-calculus extended with ML-style higher-order references. As discussed in [2,1,12,18,9] it is natural to try to give the semantic interpretation of its types in the following (flawed) way:

$$\mathsf{type} = \mathsf{world} \to \mathcal{P}(\mathsf{value}) \qquad\qquad \mathsf{world} = \mathsf{loc} \rightharpoonup_{\mathsf{fin}} \mathsf{type}. \tag{1}$$

The idea is that the world associates with each location its semantic type, and the semantic types use the world to give the interpretation to the reference types: the interpretation of a reference type should only contain locations that are associated with the appropriate type by the world. However, the recursive dependency between type and world is not well-founded and thus a simple set-theoretic solution to such an equation does not exist in general (for a more detailed treatment of this circularity see e.g. [9]).

Various solutions to this problem have been proposed, using tools like explicit step-indexing [2,1], Hobor *et al.*'s indirection theory, which also comes with an associated Coq formalization [18], and ultrametric spaces [12,9], among others. In our library, we formalize a variant of the ultrametric-space approach described by Birkedal et al. [12,9,11], that not only gives us a general way of solving the recursive domain equations, but puts it in a larger context that allows us to easily utilize it to build higher-order logics with recursively defined predicates. In the following, we explain some of the basics of our approach, and why we believe that it forms a low-boilerplate and powerful tool, and present how it can be used to model and solve equations like (1). We also report on a recent succesful application of the ModuRes Coq library to formalize a state-of-the-art program logic for a concurrent higher-order imperative programming language [21].

2 COFE in Coq

As mentioned above, the recursive domain equations that often arise when modeling sophisticated type systems and logics for advanced programming languages, in general do not have a solution using standard sets and functions. In the ModuRes library we use the approach of Birkedal *et al.* [11] and move from the category **Set** of sets and functions to categories enriched over certain kinds of ultrametric spaces. For formalization purposes, it is important that this move is not too cumbersome in practical use. The ModuRes library and this paper therefore makes use of an alternative, simpler, presentation of the necessary ultrametric spaces, known as complete ordered families of equivalences, or COFEs [17]. A tutorial presentation of COFEs and their application to models of higher-order logics can be found in Birkedal and Bizjak [8]. In this section we focus on our formalization in Coq, and introduce the types that are necessary to understand the statement and solution of recursive domain equations.

2.1 Using Type Classes: Types with Equality

COFEs will be represented in Coq as types enriched with some additional information. To make it easier to work with such enriched types, we make use of

Coq's type classes [24]. In this subsection we explain the general approach we follow, by first considering how to represent types with an equality provided by the user (known in the Coq standard library as a setoid).

We begin by calling to mind the setoid definition from the standard library:

```
Class Setoid A := {
  equiv : relation A ;
  setoid_equiv :> Equivalence equiv }.
```

This definition means that for any type T for which we can find an instance of the Setoid class, we have a relation, and we know that this relation is an equivalence relation. The type class system allows us to build many generic constructions on setoids, such as products and subset types. Consider, for instance the product of two setoids, with the usual, pointwise equality. The following snippet gives the definition, with the proof that the relation is indeed an equivalence elided (recall == is the standard library infix notation for equiv).

```
Context `{eqU : Setoid U} `{eqV : Setoid V}.
Definition prod_equiv (p1 p2 : U * V) :=
  fst p1 == fst p2 /\ snd p1 == snd p2.
Global Program Instance prod_setoid : Setoid (U * V) :=
  Build_Setoid (equiv := prod_equiv) _.
Next Obligation. ... Qed.
```

The principal strength of the type class mechanism as applied as above is in the lightweight usage: any type T * U can be considered as a setoid as long as we can find a proof that both arguments have a setoid structure too.

We can build many general constructions on types of equality, but one of particular importance is the function space on equality types. As with many other mathematical structures, the general functions between the underlying types are too loose: we need a space of "good" functions. In the case of setoids, the appropriate notion of goodness is, unsurprisingly, preservation of equality. We can define equality-preserving maps as follows:

```
Record morphism S T `{eqS : Setoid S} `{eqT : Setoid T} :=
  mkMorph {
    morph :> U -> V;
    morph_resp : Proper (equiv ==> equiv) morph}.
Infix "-=>" := morphism (at level 45, right associativity).
```

Several things are worth noting here. Firstly, an equality preserving function is actually a *record* that carries with itself the proof of that property. Note that while we could declare it a typeclass, this would not give us much, since finding a proof of preservation of equality is beyond the type-directed proof search that the type class mechanism employs. Secondly, we are using the standard library Proper class, which expresses the general notion of a relation being preserved by a function symbol — in our case, that the function morph preserves setoid equality, i.e., takes equiv arguments to equiv results. Finally, in contrast to the general

function spaces, we can define the setoid structure for the equality preserving maps, as illustrated by the following snippet:

```
Context `{eqT : Setoid T} `{eqU : Setoid U}.
Definition morph_equiv (f g : T -=> U) := forall t, f t == g t.
Global Program Instance morph_setoid : Setoid (T -=> U) :=
  Build_Setoid (equiv := morph_equiv) _.
Next Obligation. ... Qed.
```

This is the gist of the pattern of enriching the types that we follow throughout the library. It is reminiscent and indeed inspired by the approach of Spitters and van der Weegen [25]; we discuss the relationship in more detail in Section 5. The library provides many additional constructions on setoids, as well as useful definitions and lemmas. However, the most important aspect is that we provide a lightweight way of enriching the types, particularly the function spaces.

2.2 Complete Ordered Families of Equivalences

After seeing the general pattern we use to enrich types, we can move to defining the actual COFEs. Conceptually, ordered families of equivalences are an extension of setoids: where the setoid provides one notion of equality for a given type, an ordered family of equivalences (OFE) provides a whole family of equivalences that *approximate* equality. Formally, we define them as follows:

```
Class ofe T {eqT : Setoid T} :=
  { dist : nat -> T -> T -> Prop;
    dist_morph n :> Proper (equiv ==> equiv ==> iff) (dist n);
    dist_refl : forall x y, (forall n, dist n x y) <-> x == y;
    dist_sym n :> Symmetric (dist n);
    dist_trans n :> Transitive (dist n);
    dist_mono : forall n x y, dist (S n) x y -> dist n x y;
    dist_bound : forall x y, dist 0 x y}.
```

As we can see, in addition to a setoid structure, an ofe has a relation for each natural number. These are *ordered*, in the sense that $\mathsf{dist}_{\mathsf{mono}}$ implies: a relation at level n is included in all smaller levels. Furthermore, the relation is trivial at level 0, and the family's limit is the setoid equality — i.e., two elements of an OFE are considered equal iff they are approximately equal at all levels, as enforced by $\mathsf{dist}_{\mathsf{refl}}$. The somewhat cryptic statement of $\mathsf{dist}_{\mathsf{morph}}$ ensures that the approximate equality respects the setoid equality, i.e., that one can freely substitute equal arguments of dist. We use the following notation for the approximate equality:

```
Notation "x ' = ' n ' = ' y" := (dist n x y).
```

For the OFEs to serve their purpose, we need one extra property: completeness of all Cauchy chains. Cauchy chains are sequences (or chains) whose elements get arbitrarily close together. An OFE T is *complete* (a **C**OFE) if all Cauchy chains *converge* to an element of T: for any level of approximation n there is a tail of the chain whose elements are all n-equal to the limit. Formally

```
Definition chain (T : Type) := nat -> T.
Class cchain `{ofeT : ofe T} (c : chain T) :=
  chain_cauchy : forall n i j (HLei : n <= i) (HLej : n <= j), (c i) = n = (c j).
Definition converges (c : chain T) (m : T) :=
  forall n, exists k, forall i (HLe : k <= i), m = n = (c i)
Class cofe T `{ofeT : ofe T} :=
  { compl : forall c {cc : cchain c}, T;
    conv_cauchy : forall c {cc : cchain c}, converges c (compl c)}.
```

Constructions on COFEs Much like in the case of types with equality, we can provide various standard constructions on COFEs. For most standard constructions, such as products and indexed products, or subset types, it suffices to define the approximation pointwise.[2] However, we can also define another simple, but important space: step-indexed propositions, or, *uniform predicates*. The (slightly simplified) definition is as follows:

```
Record UPred T :=
  mkUPred {p :> nat -> T -> Prop;
           uni_pred : forall n m t (HLe : m <= n), p n t -> p m t}.
```

As we can see, these are families of predicates over some type T, such that the predicates are *downwards-closed* in the natural number component. This makes it easy to define non-trivial approximate equality on them:

```
Definition up_dist {T} n (p q : UPred T) :=
    forall m t, m < n -> (p m t <-> q m t).
```

Note that we require the propositions to be equivalent only for *strictly smaller* indices: this ensures that any two uniform predicates are equal at level 0.

Non-expansive maps For function spaces between COFEs, we can proceed in a manner similar to the equality types. Here, the appropriate notion of a function space consists of the non-expansive functions, i.e., the equality preserving maps that also preserve all the approximations:

```
Record ofe_morphism T U `{oT : ofe T} `{oU : ofe U} :=
  mkUMorph { ofe_morph :> T -=> U;
             ofe_morph_nonexp n : Proper (dist n ==> dist n) met_morph}.
Infix "→ne" := ofe_morphism (at level 45, right associativity).
```

Following the pattern set by the equality types, we can show that this notion of function space between COFEs itself forms a COFE with the usual application and abstraction properties. In other words, the category with objects COFEs and with morphisms non-expansive functions is cartesian closed.

[2] Although in the case of subset types an extra condition is required to make sure the subset is complete.

2.3 Contractiveness and Fixed Points

A *contractive* function is a non-expansive function for which the level of approximate equality between the results of function application not only persists, but actually increases. Formally, we can define it as

Class contractive (f : T $\to_{\mathsf{ne}}$ U) := contr n : Proper (dist n ==> dist (S n)) f.

Observe, that if we have a contractive endofunction f on some space T, the results of iterating f on any two elements get progressively more and more equal — any two elements of T are 0-equal, so the results of applying f to them are 1-equal, and so on. This results in a version of Banach's fixed-point theorem for contractive maps on COFEs, i.e., we get the following properties:

Definition fixp (f : T $\to_{\mathsf{ne}}$ T) {HC : contractive f} (x : T) : T := ...
Lemma fixp_eq f x {HC : contractive f} : fixp f x == f (fixp f x).
Lemma fixp_unique f x y {HC : contractive f} : fixp f x == fixp f y.

The term fixp constructs the fixed point of a contractive function f by iterating f starting at x and taking the limit (hence the need for completeness). The lemma fixp_eq expresses that fixp f x is indeed a fixed point of f for any x and lemma fixp_unique shows that the starting point of the iteration is irrelevant.

When using the library to build models of type systems and logics, fixed-point properties allow us to interpret recursively defined types and predicates.

3 Solving the Recursive Domain Equation

Following Birkedal *et al.* [11], the ModuRes library provides solutions to recursive domain equations in categories enriched over COFEs. Here, for simplicity of presentation, we just present our approach to recursive domain equations for the concrete category of COFEs,[3] which suffices for many applications.

We first describe the interface of the input to the part of the library solving recursive domain equations, then describe the interface of the provided solution, and finally describe how one may use the solution.

3.1 Interface of the Recursive Domain Equation

The ModuRes library provides solutions to recursive domain equations in the category of COFEs. A recursive domain equation must be specified by a suitable functor F on the category of COFEs. To accomodate mixed-variance recursive domain equations, the functor F must be a bifunctor, such that positive and negative occurrences of the recursion variable are split apart. In the example from the Introduction, we were seeking a solution to the equation

$$\mathsf{T} \simeq (\mathsf{Loc} \rightharpoonup_{\mathsf{fin}} \mathsf{T}) \to \mathsf{Pred}(\mathsf{Val})$$

[3] Since the category of COFEs itself is cartesian closed it is indeed enriched over itself, and hence it is a special case of the general approach provided by the library.

In this case, there are no positive occurrences, and just one negative one. Thus, in this case, our functor F can be defined by

$$\mathsf{F}(\mathsf{T}^-, \mathsf{T}^+) = (\mathsf{Loc} \rightharpoonup_{\mathsf{fin}} \mathsf{T}^-) \to_{\mathsf{ne}} \mathsf{UPred}(\mathsf{Val}).$$

Note that since the result needs to be a COFE, we need to use non-expansive function spaces ($\to_{\mathsf{ne}}$) and the step-indexed predicate space, UPred.[4]

The key parts of the interface to a recursive domain equation are:

```
Module Type SimplInput.
  ...
  Parameter F : COFE -> COFE -> COFE.
  Parameter FArr : BiFMap F.
  Parameter FFun : BiFunctor F.
  Parameter F_ne : unit →ne F unit unit.
End SimplInput.
```

The interface consists of several components. The function F works on type COFE — a record that packs a type together with a proof that it is indeed a COFE. We have to enforce that F is not just any function on COFEs, but a *functor*. To this end, we need to provide a definition of how it transforms non-expansive functions on its arguments into non-expansive functions on the result: a requirement expressed by the BiFMap typeclass. Moreover, the result has to be contravariant in the first argument, and covariant in the second, as shown in the following formulation, specialized to the category of COFEs

```
Class BiFMap (F : COFE -> COFE -> COFE) :=
  fmorph : forall {t t' u u'}, (t' →ne t) * (u →ne u') →ne (F t u →ne F t' u').
```

In our running example, this means that for any spaces T_1^-, T_1^+, T_2^-, T_2^+, and any functions $\mathsf{f} : \mathsf{T}_2^- \to_{\mathsf{ne}} \mathsf{T}_1^-$, $\mathsf{g} : \mathsf{T}_1^+ \to_{\mathsf{ne}} \mathsf{T}_2^+$ we need to build – in a non-expansive way – a function of type

$$((\mathsf{Loc} \rightharpoonup_{\mathsf{fin}} \mathsf{T}_1^-) \to_{\mathsf{ne}} \mathsf{UPred}(\mathsf{Val})) \to_{\mathsf{ne}} (\mathsf{Loc} \rightharpoonup_{\mathsf{fin}} \mathsf{T}_2^-) \to_{\mathsf{ne}} \mathsf{UPred}(\mathsf{Val}).$$

As is usual in such cases, there is only one sensible definition: we define FArr as

$$\begin{aligned}&\mathsf{FArr}\ (\mathsf{f} : \mathsf{T}_2^- \to_{\mathsf{ne}} \mathsf{T}_1^-, \mathsf{g} : \mathsf{T}_1^+ \to_{\mathsf{ne}} \mathsf{T}_2^+)\ (\mathsf{P} : (\mathsf{Loc} \rightharpoonup_{\mathsf{fin}} \mathsf{T}_1^-) \to_{\mathsf{ne}} \mathsf{UPred}(\mathsf{Val}))\\ &\quad (\mathsf{w} : \mathsf{Loc} \rightharpoonup_{\mathsf{fin}} \mathsf{T}_2^-) = \mathsf{P}(\mathsf{map}\ \mathsf{f}\ \mathsf{w}),\end{aligned}$$

i.e., use the function f to map over the finite map that describes the world, and finish the definition by using the predicate we took as an argument. Of course, we also need to check that this definition is non-expansive in all of the arguments, which is a simple excercise.

To prove that F actually forms a functor, we need one more check: the definition of FArr has to preserve compositions of non-expansive maps, as well as the

[4] For the actual application to modeling ML-style reference types, the functions should not only be non-expansive, but also monotone wrt. a suitable extension ordering on the worlds. The ModuRes library includes support for such, but we omit that here.

identity function. This is expressed using the BiFunctor typeclass. Again, these conditions typically amount to simple proofs.

The final ingredient in the interface is that we should provide a proof that the type produced by F, if it is applied to the singleton space, is inhabited. Obviously, this is easy to achieve for our running example: any constant uniform predicate is a good candidate, and we have the freedom to pick any one.

3.2 Interface of the Solution

Now that we know how to represent a recursive domain equation, we can look at what our general solution provides. The signature looks as follows (here, again, specialised to the type of COFEs, rather than more general enriched categories).

```
Module Type SolutionType(InputF : SimplInput).
  Import InputF.
  Axiom TInf : COFE.
  Axiom Fold : ▷ (F TInf TInf) →ne TInf.
  Axiom Unfold : TInf →ne ▷ (F TInf TInf).
  Axiom FU_id : Fold ∘ Unfold == id TInf.
  Axiom UF_id : Unfold ∘ Fold == id (▷ (F TInf TInf)).
  ...
End SolutionType.
```

First of all, the solution provides a COFE by the name of TInf. This is the solution to the recursive domain equation, an abstract complete, ordered family of equivalences. Since we don't know its definition (the definition is not provided as part of the interface) — and, indeed, we do not need to know it — we need some other ways to use it. This is provided by the two dual functions: Fold and Unfold, that convert between TInf and the input functor that defines the recursive domain equation, F.

The Unfold function takes an object of type TInf to (F TInf TInf): but there is a twist, namely, the ▷ operator, usually called "later". This operator acts on the distances of the spaces that is its argument, bringing them "one step down". That is, if we have $\mathsf{m}_1 = \mathsf{n} + 1 = \mathsf{m}_2$ in some space M, we only have $\mathsf{m}_1 = \mathsf{n} = \mathsf{m}_2$ in the space ▷ M. This has consequences for the Unfold function: assume we have some elements t_1, t_2 of our solution TInf, and that $\mathsf{t}_1 = \mathsf{n} + 1 = \mathsf{t}_2$. In such a case, we can learn something about the structure of these elements by applying Unfold (remember that we know the definition of F), but we can only establish that Unfold $\mathsf{t}_1 = \mathsf{n} =$ Unfold t_2. We will see in the following section that while this affects the way we use the solution, it does not give rise to any problems.

While Unfold provides us with a way of extracting information from the solution, Fold does the converse: it gives us a way to convert an object of type (F TInf TInf) into a TInf. Similarly to Unfold, there is a later operator, although in this case it allows us to *increase* the level of approximation, since it appears on the argument type.

Finally, the solution provides two equations that describe the effects of Fold and Unfold. These effects are simple: if one follows Fold with an Unfold, the

resulting function is equivalent to an indentity; similarly if we follow unfolding with folding. This provides us with the only way to eliminate the calls to Unfold and Fold that can appear in the definitions that use the solution.

3.3 Using the Solution

As we can see, apart for the later operators appearing in some places, the solution interface is remarkably simple. However, this leads to the question of how we can use it in practice. To address this question, in this section we take the solution that arises from our running example, and show some key cases of a simple unary logical relation that uses the solution as a foundation.

In Section 3.1 we have defined the functor F that describes the recursive domain equation that arises from the presence of reference types. The solution of this equation has given us a type TInf. The final types we need to define now are the semantic types that we will use to interpret the syntax (the type T), and the useful helper type of worlds W.

Definition T := F TInf TInf.
Definition W := Loc $\rightharpoonup_{\mathsf{fin}}$ TInf.

The easiest way to define a logical relation in Coq is by first defining semantic operators that will be used to interpret the programming language type constructors. Particularly interesting is the semantic operator for interpreting reference types. Naturally, the operator, call it sref, should map semantic types to semantic types: sref : T $\rightarrow_{\mathsf{ne}}$ T. The idea is that if R is a semantic type interpreting a programming language type τ then sref(R) should interpret the programming language type ref τ, and thus, loosely speaking, sref(R) should, given a world w consist of those locations in the world whose type matches R. Now, if we unroll the definitions of T and F, we can see that there are a lot of side conditions about non-expansiveness that we will need to prove when defining sref. Thus, it is easiest to start by building the actual predicate. This can be done roughly as follows (explanation follows below):

```
Inductive refR : T -> W -> nat -> val -> Prop :=
| RRef (w : W) (R : T) (Rw : TInf) l n (HLup : w l = Some Rw)
       (HEqn : R = n = Unfold Rw) : refR R w n (vLoc l).
```

The next step is to show that refR is in reality a uniform predicate, and that the remaining function spaces are non-expansive. We elide these straightforward proofs, and instead examine the definition itself.

Let us look at the arguments of refR. The first, R, is the semantic type that models the argument of the reference type. The following, w, is the world followed by the step index of the uniform predicate, and the value that inhabits our type. Obviously, this value should be a location, but what makes a location a proper value of a reference type? The answer lies in the assumptions: HLup ensures that the location is defined in the world, and gives us a type Rw. However, Rw is of the type TInf, and R is at type T. This is where the conversion functions from the solution interface come in: we can use Unfold to cast Rw to type T (with the

equalities brought "one step down"), which allows us to compare it to R with the HEqn assumption: we claim that R is indistinguishable from Rw at the given approximation level.

The other place in the definition of the logical relation that needs to use the world in a nontrivial way is the interpretation of computations. To interpret computations we need to take a heap that *matches the world* and, in addition to ensuring that the resulting value is in the interpretation of the type, we need to check that the resulting heap also matches the world (allowing for its evolution). What does it mean for a heap to match the world? Clearly, all the locations in the world should be defined in the heap. Additionally, the values stored in the heap should belong to the semantic types stored in the world. This gives us the following definition:

```
Definition interpR (w : W) (n : nat) (h : heap) :=
  forall l R (HLu : w k = Some R), exists v, h k = Some v /\ Unfold R w n v.
```

Like in the definition of refR, we use the Unfold function to map the element stored in the world back to type T, in this case in order to apply it to arguments.

One may be concerned that we only use the Unfold function in these definitions: clearly there should be places in the logical relations argument where we need to use the Fold function. The answer is, that Fold is used in the proof of compatibility of the allocation rule. This is the one rule that forces us to extend the world, and to do this we need to come up with an object of type TInf — which we can only procure by using Fold. This is also the only place where we need to use the proof that Fold and Unfold compose to identity, to ensure that the resulting heap matches the extended world.

3.4 Summary

In summary, we have now seen how a user of the ModuRes library can obtain a solution to a recursive domain equation in the category of COFEs by providing a suitable bifunctor. The library then provides a solution with associated isomorphisms, which the user of the library can use to build, e.g., a logical relations interpretation of programming language reference types.

The user of the ModuRes library does not need to understand how the solution is constructed. The construction in Coq follows the proof in Birkedal *et al.* [11].

4 Building Models of Higher-Order Logics

In this section we explain how the ModuRes library can be used to build models of higher-order logics for reasoning about concurrent higher-order imperative programs. Concretely, we describe the core part of the model of Iris, a recently published state-of-the-art program logic [21]. For reasons of space, we cannot explain in detail here *why* the core part of Iris is defined as it is; for that we refer the reader to *loc. cit.* Instead, we aim at showing how the library supports

working with a model of higher-order logic using a recursively defined space of truth values.

To reason about mutable state, Iris follows earlier work on separation logic where a variety of structures with which models of propositions can be built have been proposed [14,16,23,13], see [20, Chapter 7] for a recent review. Iris settles on a simple yet expressive choice to model propositions as suitable subsets of a partial commutative monoid. In the simplest case, the partial commutative monoid can be the one of heaps, as used in classical separation logic, but in general it is useful to allow for much more elaborate partial commutative monoids [21]. To reason about concurrent programs, Iris propositions are indexed over named invariants, which are used to describe how some shared state may evolve. There has been a lot of research on developing a rich language for describing invariants (*aka* protocols); one of the observations of the work on Iris is that it suffices to describe an invariant by a predicate itself, when the predicate is over a suitably rich notion of partial commutative monoid. In other words, the invariants in Iris are described by worlds, which map numbers (names of invariants) to propositions, and a proposition is a map from worlds to (a suitable) powerset of a partial commutative monoid M. Thus, to model Iris propositions using the ModuRes library, we use as input the following bifunctor:

$$\mathsf{F}(\mathsf{P}^-, \mathsf{P}^+) = (\mathbb{N} \rightharpoonup_{\mathsf{fin}} \mathsf{P}^-) \to_{\mathsf{ne,mon}} \mathsf{UPred}(\mathsf{M}),$$

Note that the functor is very similar to the one for modeling ML reference types. Here $\mathsf{UPred}(\mathsf{M})$ is the COFE object consisting of uniform predicates over the partial commutative monoid M, and $\to_{\mathsf{ne,mon}}$ denotes the set of non-expansive and monotone functions. For monotonicity, the order on $(\mathbb{N} \rightharpoonup_{\mathsf{fin}} \mathsf{P}^-)$ is given by the extension ordering (inclusion of graphs of finite functions) and the order on $\mathsf{UPred}(\mathsf{M})$ is just the ordinary subset ordering. Using the library, we obtain a solution, which we call $\mathsf{PreProp}$. Then we define

$$\mathsf{Wld} = \mathbb{N} \rightharpoonup_{\mathsf{fin}} \mathsf{PreProp} \qquad\qquad \mathsf{Props} = \mathsf{Wld} \to_{\mathsf{ne,mon}} \mathsf{UPred}(\mathsf{M}).$$

The type Props serves as the semantic space of Iris propositions. Recall that the interface of the solution gives us an isomorphism between $\mathsf{PreProp}$ and $\triangleright\mathsf{Props}$ — a fact that is not necessary to show Props form a general separation logic, but crucial when we want to make use of the invariants in the logic-specific development of Iris.

Let us take stock. What we have achieved so far is to define the propositions (the object of truth values) of Iris, as a complete ordered family of equivalences, and we obtained it by solving a recursive domain equation. We want to use this to get a model of higher-order separation logic. In Iris, types are modeled by complete ordered families of equivalences, terms by non-expansive functions, and the type of propositions is modeled by Props. We can show on paper (see the tutorial [8]) that this is an example of a so-called BI-hyperdoctrine, a categorical notion of model of higher-order separation logic [7]. In the ModuRes library we do not provide a general formalization of BI-hyperdoctrines. Instead, we have formalized a particular class of BI-hyperdoctrines, namely those where types and

terms are modeled by COFEs and non-expansive functions and propositions are modeled by a preordered COFE, the preorder modeling logical entailment. We now describe how that is done, and later we return to the instantiation to Iris.

Let T be a preordered COFE — think of T as the object of truth values. We write $\sqsubseteq$ for the ordering relation. Logical connectives will be described by the laws they need to satisfy. Most of the connectives are relatively easy to model: the usual binary connectives of separation logic are maps of type $\mathsf{T} \to_{\mathsf{ne}} \mathsf{T} \to_{\mathsf{ne}} \mathsf{T}$ that satisfy certain laws. For instance, implication is defined by the two axioms

```
and_impl : forall P Q R, and P Q ⊑ R <-> P ⊑ impl Q R;
impl_pord :> Proper (pord --> pord ++> pord) impl;
```

The former of these establishes the adjoint correspondence between conjunction and implication, and the latter ensures that implication is contravariant in its first argument and covariant in its second argument, with respect to the entailment relation (pord denotes the pre-order for which $\sqsubseteq$ is the infix notation).

This leaves us with the issue of quantifiers. Since the library provides the notion of BI-hyperdoctrine that is modeled by complete ordered families of equivalences, the quantifiers should work only for functions between COFEs — indeed, only for functions that are nonexpansive. This gives rise to the following types of the quantifiers:

```
all : forall {U} `{cofeU : cofe U}, (U →ne T) →ne T;
xist : forall {U} `{cofeU : cofe U}, (U →ne T) →ne T;
```

As with the other connectives, we require certain laws to hold. For example, below are the laws for the existential quantifier. The first law expresses that existential quantification is left adjoint to reindexing, while the second law expresses functoriality of existential quantification.

```
xist_L U `{cU : cofe U} :
  forall (P : U →ne T) Q, (forall u, P u ⊑ Q) <-> xist P ⊑ Q;
xist_pord U `{cU : cofe U} (P Q : U →ne T) :
  (forall u, P u ⊑ Q u) -> xist P ⊑ xist Q
```

Now that we have a description of what a model of higher-order separation logic is, we can proceed to show that if we let $\mathsf{T} = \mathsf{Props}$, then all the required properties hold. With the help of the ModuRes library we can establish this automatically, and in a modular fashion. Firstly, the library contains a fact that UPred(M), the space of uniform predicates over a partial commutative monoid M, forms a model of higher-order separation logic. Moreover, it also shows that the set of monotone and non-expansive maps from a preordered COFE, such as Wld, to any structure that models a higher-order separation logic itself models higher-order separation logic. Thus, it follows easily — and, thanks to the use of typeclasses, automatically — that the space of Props also satisfies the rules of the logic. In the development process, this allows us to focus on the application-specific parts of the logic, rather than on the general BI structure.

Recursive Predicates To reason about recursive programs, Iris includes guarded recursively defined predicates, inspired by earlier work of Appel et. al. [3]. In the Coq formalization they are modeled roughly as follows. First, we show that there is an endo-function on $\mathsf{UPred}(\mathsf{M})$, also denoted $\triangleright$ and also called "later" (though it is different from the later functor on COFEs we described earlier). It is defined by

```
Definition laterF (p : nat -> T -> Prop) n t :=
  match n with O => True
             | S n => p n t
  end.
```

This later operator extends pointwise to Props. Now if we have a non-expansive function $\varphi : (\mathsf{U} \to_{\mathsf{ne}} \mathsf{Props}) \to_{\mathsf{ne}} (\mathsf{U} \to_{\mathsf{ne}} \mathsf{Props})$, then if we compose it with the later operator on $\mathsf{U} \to_{\mathsf{ne}} \mathsf{Props}$ we can show that the resulting function is contractive, and hence we get a fixed-point, as described in Section 2.3.

The combination of higher-order separation logic with guarded recursively defined predicates makes it possible to give abstract specifications of layered and recursive program modules [26]. Indeed, it is to facilitate this in full generality that we model the types and terms of Iris by COFEs and non-expansive functions, rather than simply by Coq types and Coq functions.

In the formalization of Iris, we also use this facility to define the meaning of Hoare triples via a fixed-point of a contractive function.

We have explained how some of the key features of Iris are modeled using the ModuRes library. The actual formalization of Iris also includes a treatment of the specific program logic rules that Iris includes. They are modeled and proved sound in our formalization of Iris, but we do not discuss that part here, since that part only involves features of the library that we have already discussed.

5 Related Work

Benton *et al.* [6] and Huffman [19] provide libraries for constructing solutions to recursive domain equations using classical domain theory. In recent work on program logics the spaces involved are naturally equipped with a metric structure and the functions needed are suitably non-expansive, a fact used extensively in the ModuRes library. In contrast solutions using domain theory do not appear to provide the necessary properties for modeling such higher-order program logics.

The line of work that is possibly the closest related to the ModuRes library is Hobor *et al.*'s theory of indirection [18] and their associated Mechanized Semantic Library in Coq [4]. It provides an *approximate* step-indexed solution of a recursive domain equation expressed as a functor in the category of sets. The explicit aim of indirection theory is to obtain the approximations in a *simple* setting, and it manages to do so: in the Mechanized Semantic Library, the approximate solution is represented as a usual Coq type. Thus, one can use standard Coq function spaces and standard higher-order quantification, over Coq types.

As argued in detail in [10], the ultrametric — or, equivalently, COFE — treatment that the ModuRes library implements subsumes indirection theory. What's

more, it is not necessarily limited to step-indexing over operational models, but can also be readily used over classical domain-theoretic denotational semantics. Generally, it also seems easier to enrich the constructions with additional structure, such as the preorder in Section 4, or build recursive objects and predicates through the use of Banach's fixed-point operator.

Another important aspect is pointed out in the recent work by Svendsen and Birkedal [26], where they explore program logis for reasoning about layered and recursive abstractions. To this end, they need to quantify over higher-order predicates, and use them within definitions of other, recursive, predicates. In general this can lead to problems with well-definedness. However, since in their setup all maps are non-expansive by default, the recursive predicate can be defined without problems. In contrast, if the model admitted other functions, the question would be far from certain, and probably require explicit restrictions to non-expansive maps in certain places, and more frequent reasoning explicitly in the model. Thus to model higher-order logics with guarded recursion in general, functions should be modeled by non-expansive functions (rather than all functions). This is the approach taken by the ModuRes library. See Section 4 (Exercise 4.17) and Section 5 (Corollary 5.22) of the tutorial [8] for more details.

Throughout the development, we intended the ModuRes library to provide a readable and easy to grasp set of tools. This would not be possible without some recent developments in proof engineering. Throughout the development we heavily use the type class feature of Coq, due to Sozeau and Oury [24] to abstract from the particular kind of enriched structure we are working in and present a simple and meaningful context to the user. In setting up the pattern for the hierarchy of classes that forms the backbone of the development — the types with equality, OFEs and COFEs — we follow closely Spitters and van der Weegen's work [25], which inspired our attempt to set up this hierarchy in a clean and readable fashion. We deviate somewhat from their pattern, however, in that we package the operations together with the proofs (see dist in the definition of ofe in Section 2), where their approach would separate them from the propositional content. This works because our hierarchy does not contain problematic diamond inheritance patterns, and we found that it improves performance.

6 Conclusions and Future Work

A question one could ask of any development in this line of work is, how easy the library is to adopt. Our early experiences seem encouraging in this regard. An intern, who recently worked with us on applying the library, was able to pick it up with little trouble. Moreover, in the development and formalization of Iris [21] this was also our experience. We believe that the abstract presentation of the solution of the recursive domain equation, and the structured construction of the spaces that the library supports will allow this experience to scale. However, since the authors were involved in both of these projects, it is not a certainty. To make the library more accessible to others, we have started developing a tutorial for the ModuRes library. While still a work in progress, we believe it can already

be helpful for potential users of the library. It can be found, along with the library, online at `http://cs.au.dk/~birke/modures/tutorial`. It features, in particular, a detailed tutorial treatment of the running example we used in this paper, a logical relations interpretation of an ML-like language with general references.

Outside the tutorial context, we see the future of the ModuRes library as serving in the development of formalized models of programming languages and program logics. It would be of particular interest to try to extend the simpler models currently used by tools that attempt to verify programs *within* Coq, such as Bedrock [15] or Charge! [5], in order to enhance the expressive power of their respective logics. The proof-engineering effort required to achieve this in an efficient and scalable is non-trivial, however. We hope that the recent progress by Malecha and Bengtson on reflective tactics can help in this regard [22].

In conclusion, we have presented a library that provides a powerful, low-boilerplate set of tools that can serve to build models of type systems and logics for reasoning about concurrent higher-order imperative programing langauges. The structures and properties provided by the library help the user focus on the issues specific to the model being constructed, while the well-definedness of the model and some of the common patterns can be handled automatically. Thus, we believe that this line of work can significantly lower the barrier to entry when it comes to formalizing the complex models of programming languages of today.

Acknowledgements

The formalization of the general solution of the recursive domain equation is inspired by an earlier, unpublished development by Varming and Birkedal. While both the proof engineering methods used and the scope of the ModuRes library differ significantly from this earlier effort, some of the setup is borrowed from that. Yannick Zakowski was the first user of the library, providing important feedback, as well as a formalization of the example used in Section 3. We thank the anonymous reviewers for their comments.

This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

References

1. Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.
2. Amal Ahmed, Andrew W. Appel, and Roberto Virga. A stratified semantics of general references embeddable in higher-order logic. In *LICS*, 2002.
3. Andrew Appel, Paul-André Melliès, Christopher Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *POPL*, 2007.
4. Andrew W. Appel, Robert Dockins, and Aquinas Hobor. `http://vst.cs.princeton.edu/msl/`, 2009.

5. Jesper Bengtson, Jonas B. Jensen, and Lars Birkedal. Charge! - A framework for higher-order separation logic in coq. In *ITP*, 2012.
6. Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in coq. In *TPHOL*, 2009.
7. Bodil Biering, Lars Birkedal, and Noah Torp-Smith. Bi hyperdoctrines and higher-order separation logic. In *ESOP*, 2005.
8. Lars Birkedal and Aleš Bizjak. A taste of categorical logic – tutorial notes. `http://cs.au.dk/~birke/modures/tutorial/categorical-logic-tutorial-notes.pdf`, 2014.
9. Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed kripke models over recursive worlds. In *POPL*, 2011.
10. Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed Kripke models over recursive worlds. In *POPL*, 2011.
11. Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theor. Comput. Sci.*, 411(47):4102–4122, 2010.
12. Lars Birkedal, Jacob Thamsborg, and Kristian Støvring. Realizability semantics of parametric polymorphism, general references, and recursive types. In *FOSSACS*, 2009.
13. James Brotherston and Jules Villard. Parametric completeness for separation theories. In *POPL*, 2014.
14. Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS*, 2007.
15. Adam Chlipala. The bedrock structured programming system: Combining generative metaprogramming and hoare logic in an extensible program verifier. In *ICFP*, 2013.
16. Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *APLAS*, 2009.
17. Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In *TYPES*, 2002.
18. Aquinas Hobor, Robert Dockins, and Andrew Appel. A theory of indirection via approximation. In *POPL*, 2010.
19. Brian Huffman. A purely definitional universal domain. In *TPHOL*, 2009.
20. Jonas B. Jensen. *Enabling Concise and Modular Specifications in Separation Logic*. PhD thesis, IT University of Copenhagen, 2014.
21. Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, 2015.
22. Gregory Malecha and Jesper Bengtson. Rtac — a reflective tactic language for Coq. Submitted for publication, 2015.
23. François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *JFP*, 23(1):38–144, 2013.
24. Matthieu Sozeau and Nicolas Oury. First-class type classes. In *TPHOLs*, 2008.
25. Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.
26. Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, 2014.