

Effectful Programming across Heterogeneous Computations

— Work in Progress

Jean Abou Samra¹, Martin Bodin², and Yannick Zakowski³

¹ ENS, Paris, France

`jean@abou-samra.fr`

² Inria, Grenoble, France

`martin.bodin@inria.fr`

³ Inria, Lyon, France

`yannick.zakowski@inria.fr`

Keywords: Formal Semantics, Free Monad, Coq, Algebraic Effects.

Abstract

Monadic programming is a popular way to embed effectful computations in purely functional languages. In particular, the so-called free-monad comes with the promise of extensibility and modularity: computations are seen as syntax arising from a signature of operations they may perform. Popularized in a programming context, the approach is nowadays used for verification in proof assistants as well, as witnessed by frameworks such as FreeSpec or Interaction Trees.

In this work in progress, we investigate the following question. Can we type each sub-computation with their minimal valid operation signature, and seamlessly combine all these monadic computations of different natures? Furthermore, can we leverage this additional precision in typing to derive monadic invariants for free?

We answer positively by suggesting two simple ideas. First, a bind operation between computations living in distinct monads can be defined by transporting them through monad morphisms. Second, to give more structure to the monads we manipulate by indexing them by a semi-lattice as a means to automatically infer the adequate morphisms. We illustrate the benefits on a minimal Coq example: a computation interacting with a memory cell.

1 Introduction

Monads were initially introduced as a semantic model for effectful computations [11]. Quickly, they have spread beyond this formal setup to become a popular programming device allowing for safely encapsulating effects in functional programs [4, 13], taking most famously a central role in Haskell’s design.

Under the lens of programming, a given monad consists of three core components.¹ First, a family of types specifies a domain of computations: the `Option` type constructor captures potentially failing computations, the `List` type constructor can be seen as specifying non-deterministic computations, state-threading functions may encapsulate stateful computations, and so on. Monads are furthermore equipped with two operations: the `ret` construct describes how to embed a pure computation into the domain of computations, while the `bind` operation describes how computations can be sequenced (hence its notation ‘`x ← m ; k`’). Figure 1 illustrates the Coq definition of a stateful monad over a single memory `cell` storing a natural number.

¹In general, monads should also come equipped with a notion of equivalence of computation. We brush this detail under the rug in this presentation, pretending that we may work with Coq’s equality `eq` everywhere. We refer the interested reader to our formal development for a proper setoid-based approach.

```

Definition monad := Type → Type.

Definition cell : Type := nat.
Definition state : monad := fun X => cell → cell * X.
Definition state_ret : ∀ X, X → state X := fun X x c => (c, x).
Definition state_bind : ∀ X Y, state X → (X → state Y) → state Y :=
  fun X Y m k c => let (c', x) := m c in k x c'.

```

Figure 1: The state monad (👉)

```

Definition signature := Type → Type.
Notation "M ~> N" := (∀ X, M X → N X).
Inductive free (E : signature) (X : Type) :=
  | pure (x : X)
  | op Y (e : E Y) (k : Y → free E X).

Variant Rd : signature :=
  | rd : Rd cell.
Variant Wr : signature :=
  | wr (c : cell) : Wr unit.

Fixpoint interp E M (MM: Monad M)
  (h : E ~> M) X (m : free E X) : M X :=
  match m with
  | pure x => ret x
  | op e k =>
    bind (h e) (fun x => interp h (k x))
  end.

Definition h_state
  : Rd +' Wr ~> state :=
  fun _ e c => match e with
  | inl1 rd => (c,c)
  | inr1 (wr c') => (c',tt)
  end.

```

Figure 2: The free monad (👉) and its use for stateful computations (👉)

Perhaps more surprisingly, monads offer a solution to Wadler’s expression problem. The *free monad* (Figure 2) acts as an extensible syntax [12] for effectful computations: effects are thought of as arising from the use of effectful operations described via an interface E . A computation is then essentially encoded as a tree where leaves contain pure computations, and nodes contains operations. The relationship between a node and its children arises from a continuation indexed by the type of answer expected from a valid implementation of the operation. This tree structure is naturally equipped with a monadic structure: leaves directly act as `ret` operations, while `bind` attaches the appropriate continuation to the leaves of the first part of the computation (👉).

The right hand-part of Figure 2 illustrates the approach when declaring computations manipulating the previously mentioned cell. The `rd` event specifies the *read* operation: the fact that one expects to get back the content of the cell following this operation is reflected in its type `rd : Rd cell`. In contrast, a *write* event `wr c` is intended to update the content of the cell, but does not entail any informative answer: its return type is merely an acknowledgement, embodied by the *unit* type. From there, one can write rich programs manipulating this memory cell as elements of the monad `free (Rd +' Wr)` where `+'` is the disjoint sum of both signatures. Read operations give rise to an infinite branching, with a branch per natural number, while write operations have a single child. In concrete syntax, a computation doubling the current content of the cell becomes:

```

Definition double : free (Rd +' Wr) unit := n ← rd;; wr (2 * n).

```

The final crucial ingredient to our story is hidden behind the *free* monad’s name: except for their return type, the operations are free of constraints, no semantics is attached to them! Because of this freedom, one can plug’n’play the monadic implementation of one’s choice: by folding over the tree, substituting nodes for their implementations, the `interp` function lifts

monadic implementations of operations into implementations of computations. For instance, `h_state` provides a typical stateful implementation to read and write operations, such that `interp h_state double` becomes the expected executable function of an initial cell.

Monadic interpreters built atop of the free monad are an increasingly popular way to formalize the semantics of programming languages in proof assistants based on dependent typed theories, such as Coq for instance. Indeed, via variants of the *delay* monad [2, 10], divergence can be internalized, freeing us from Coq’s termination checker when writing monadic definitional interpreters. When applicable, the approach offers the benefits of compositionality — in the traditional sense of denotational semantics —, of modularity — effects are specified in isolation and modularly composed —, and of executability — through extraction to OCaml for instance, testing is made possible, including when dealing with non-terminating programs.

Specifically in the Coq ecosystem, the approach has been implemented in the FreeSpec library [7, 6, 8] and the Interaction Tree (ITree) project [14, 15]. The viability of the method has been shown to scale to realistic settings such as for modelling LLVM IR [16], or to support non-determinism and model concurrency [5, 3]. In this context, *formal reasoning* about these monadic computations takes back a central place, combining the algebraic reasoning enabled by the monadic setup to unary or relational Hoare style reasoning.

The extensibility of the free approach is central to these large scale projects to enable modularity: orthogonal effects are specified in isolation, and theories can be reused across projects. It is additionally used as a means to painlessly link computations arising from different sources: one can always *translate* free computations into free computations over larger interfaces. However, in our experience, these projects have vastly shied away from leveraging the extensibility as a means to facilitate reasoning. Indeed, signatures can be thought of as a primitive type and effect system. As such, one could play the game of typing as precisely as possible every computation. Coming back to our cell example, we should benefit when possible from reflecting in the type of a computation that the cell is never written to, by typing the computation at type `free Rd X` rather than the larger `free (Rd +' Wr) X`. But we should be able to go further: if considering now a stateful computation over several cells, one should be able to reflect in the signature an over-approximation of the read and write location sites of the computation. Speculating even further, invariants could be encoded in dependent types, specifying for instance that the cell may be written to, but only in an increasing fashion.

In this work in progress, we ask the question: is this ambition practical? Can we meaningfully and painlessly combine a zoo of computations of different natures, both as free computations over diverse signatures, but also as their semantic implementations in monads of diverse complexities? Can we leverage this additional typing information to ease the reasoning in two directions: transporting invariants for free, and allowing for reasoning in simpler monadic structures?

We introduce the following early contributions that lead us believe these question should find a positive answer:

- we observe that monadic computations of distinct natures can be bound, provided there is a pair of monad morphisms into a common target (Section 3.1);
- we propose to index the subset of monads of interest by first a partial order (Section 3.2), then a directed set (Section 3.3), in order to remove the need for outrageous explicit type annotations;
- moving to a setup based on the free monad, we propose an axiomatized interface to program with the free and concrete views of the indexing domain of computations (Section 4);

```

Definition read : monad := fun X => data -> X.
Definition read_ret X (x : X) := fun c => x.
Definition read_bind X Y (m : read X) (k : X -> read Y) : read Y :=
  fun c => k (m c) c.

Definition write : monad := fun X => X * option data.
Definition write_ret X (x : X) := (x, None).
Definition write_bind X Y (m : write X) (k : X -> write Y) : write Y :=
  let '(x,mw) := m in
  match k x with
  | (y, None) -> (y, mw)
  | (y, Some w) -> (y, Some w)
  end.

Definition pure : monad := fun X => X.
Definition pure_ret X (x : X) := x.
Definition pure_bind X Y (m : pure X) (k : X -> pure Y) : pure Y := k m.

```

Figure 3: Read only, write only, and pure monads (🍷)

- we provide a minimal instance of this interface to the case of a stateful computation over a cell (Section 2), and offer some thoughts as to how this work can lead to nicer reasoning principles (Section 5).

All our results are formalized in Coq². This paper can naturally be read as plain text, but we additionally provide hyperlinks to our source code as a support for the interested reader: those are indicated by a ‘🍷’ symbol.

2 Running Example

As a means to guide our intuition, we consider as a minimal concrete example stateful computations over a single memory cell as introduced in Section 1. To work with such effects, one would typically (1) assemble pieces of computations written in the `free (Rd + Wr)` monad, (2) interpret these computations into the `state` monad, and (3) perform any reasoning, of functional correctness for instance, in this structure.

We propose ourselves to leverage three simple pieces of static information we may collect: a computation might only read the cell, it might only write to the cell, or it might perform neither operation. On the syntactic side, a sufficient condition to ensure these invariants is to type a piece of computation in respectively the `free Rd`, `free Wr`, or `free \emptyset` ³ monad. We hence have naturally four signatures in mind, organized into a diamond w.r.t. a form of signature inclusion.

So far, little differs from previous work: we could translate the simpler signatures into the largest one when combining computations. However, this would mean that semantically, we still see all computations through the lens of the overly general state monad. Having made the existence of these four kinds of typed computations explicit, the temptation is great to instead interpret each of them into the simplest structure possible. Figure 3 suggests such structures: read only computations should not need to return an updated cell; write only ones should not

²<https://gitlab.inria.fr/yzakowsk/ordered-signatures/-/tree/jfla23/theories>

³Where \emptyset is the empty signature.

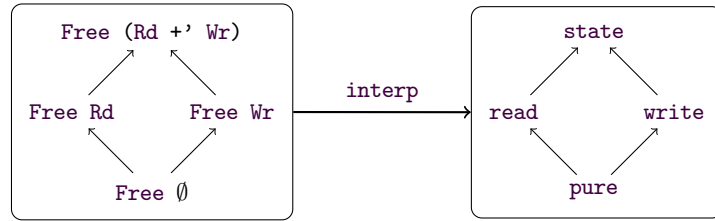


Figure 4: Flavors of stateful computations: syntax and semantics

have to take the initial cell as argument; and pure computation should live in the identity monad. We omit the formal definitions here (👉) but analogues to `h_state` can be defined to interpret each signature at play into its corresponding implementation monad.

We hence have a diamond of free monads, upon which we can climb by translation, and an isomorphic diamond of concrete monads. Although less explicit, the intuition according to which monads get “more general” when climbing in the diamond is valid in the concrete one as well. Here, the arrow between two monads `M` and `N` represents a monad morphism, i.e., an operation `fmap : M ~> N` commuting with `ret` and `bind`. For instance, the morphism between the `read` monad and the `state` monad simply exposes that the cell has been left untouched:

Definition `read_state_map`: `read ~> state := fun X m c => (c, m c)`. 👉

Figure 4 sums up the eight monads we are tempted to manipulate, four on the syntactic side, connected via interpretation to the four on the semantic side. We aim to identify the right interface necessary to program with no overhead with this heterogeneous syntax, as well as to benefit from the additional typing annotations to compose pieces of reasoning in the simplest possible structures. As a simplistic but illustrative example, we consider the following program.

```
Variable init : cell → free Wr unit.
Variable fetch : free Rd cell.
Definition main (n : cell) : free (Rd +' Wr) bool :=
  init n ;;
  v1 ← fetch ;;
  v2 ← fetch ;;
  ret (v1 =? v2).
```

The goal is twofold. Can we write this syntax, despite `init` and `fetch` living in different syntax than `main`? And second, can we frame the right theory allowing us to leverage the typing information to prove that `main n` will always return `true` while keeping the definitions of `init` and `fetch` completely opaque? We already intuitively observe that the second goal requires us to know how to combine heterogeneous computations not only across the syntactic diamond, but also across the semantic one: the interpretation of a heterogeneous bind should commute into a heterogeneous bind of interpretations into distinct monads.

3 Monadic Programming

We set aside the free monad temporarily, and first focus on developing the right mathematical framework to painlessly program with monadic computations living in different structures. This amounts to binding together computations from different monads: a value `m : M X` in a first monad `M` and a continuation `k : X → N Y` in a second monad `N`. In other words, we are

trying to define a practical structure with a heterogeneous bind operator with the following type, for several monads M , N , and T .

```
bind : ∀ X Y, M X → (X → N Y) → T Y
```

We furthermore ask this heterogeneous bind operator extends the usual monadic laws: `ret` should be a unit for `bind` on both sides, and `bind` should be associative.

3.1 Monad Morphisms to Transport

We start with a very simple observation: two computations can always be sequenced as soon as we know how to transport both arguments into a common monad. We hence request the existence of monad morphisms from M to T and from N to T and compose `bind` with the appropriate `fmaps`. Coq automatically infers the type of these `fmaps`, but for better readability, we annotate them in grey in the code.

```
Definition bindH M N T (MM : Monad M) (MN : Monad N) (MT : Monad T)
  (MMT : MonadMorphism M T) (MNT : MonadMorphism N T)
  : ∀ X Y, M X → (X → N Y) → T Y :=
  fun X Y m k => v ← fmapM↔T m; fmapN↔T (k v).
```

Where `Monad M` is a type class constraint asserting the existence of the monadic operations over M , and `MonadMorphism N T` a type class constraint ensuring the existence of a morphism `fmapN↔T` from N to T . In the rest of this subsection, we omit these constraints to lighten the presentation.

By identifying all three monads and using the identity morphism, we show that the heterogeneous bind is indeed an extension of the usual monadic bind.

```
Lemma bindH_extends_bind : ∀ M X Y (m : M X) (k : X → M Y),
  bindHM X → (X → M Y) → M Y m k = bind m k.
```

Furthermore, the monadic laws remain valid, once appropriately generalized. The left and right `ret` laws simply have to manually `fmap` the unbound pure computations.

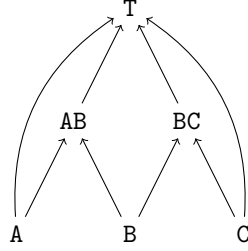
```
Lemma bindH_ret_l : ∀ M N T X Y (x : X) (k : X → N Y),
  bindHM X → (X → N Y) → T Y (ret x) k = fmapN↔T (k x).
```

```
Lemma bindH_ret_r : ∀ M N T X (c : M X),
  bindHM X → (X → N X) → T X c (fun x => ret x) = fmapM↔T c.
```

The associativity of the heterogeneous bind operator is a bit more of a mouthful. We need to consider six monads: three “base” monads A , B , and C for the arguments of the bind (`mx`, `kxy`, and `kyz`), one general monad into which the final result is computed, and two pivot monads AB and BC for the intermediary results. Figure 5 represents these structures, as well as the various morphisms each bind operator requires to state the lemma. We furthermore assume that the three subdiagrams of Figure 5 commute.

```
Lemma bindH_bindH : ∀ A B C AB BC T
  (commAB : ∀ X (m : A X), fmapAB↔T (fmapA↔AB m) = fmapA↔T m)
  (commBC : ∀ X (m : C X), fmapBC↔T (fmapC↔BC m) = fmapC↔T m)
  (commBT : ∀ X (m : B X), fmapAB↔T (fmapB↔AB m) = fmapBC↔T (fmapB↔BC m)),
  ∀ X Y Z (mx : A X) (kxy : X → B Y) (kyz : Y → C Z),
  bindH (bindH mx kxy) kyz = bindH mx (fun x => bindH (kxy x) kyz).
```

One may ponder: have we already solved our programming challenge, can we not write `main` using `bindH`? Well yes, but at great cost! Looking at the type of `bindH`, the unifying monad T


 Figure 5: Monads and their morphisms used to state `bindH_bindH`

```

Variable D : Type.
Variable reify : D → monad.
Notation "'[[ i ]]'" := (reify i).
Hypothesis reify_into_monads : ∀ i, Monad [[i]].

Variable reify_le : ∀ i1 i2, i1 ⊆ i2 → MonadMorphism [[i1]] [[i2]].
Definition climbPO i1 i2 (I12 : i1 ⊆ i2) : ∀ X, [[i1]] X → [[i2]] X :=
  fmap (MonadMorphism := reify_le I12).

Hypothesis reify_le_trans :
  ∀ i1 i2 i3 (I12 : i1 ⊆ i2) (I13 : i1 ⊆ i3) (I23 : i2 ⊆ i3) X (m : [[i1]] X),
    climbPO I23 (climbPO I12 m) = climbPO I13 m.
    
```

Figure 6: Partial Order interface 🍷

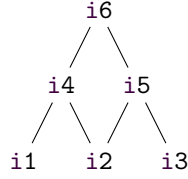
cannot be inferred from the type of the computations `m` and `k`: the programmer would essentially have to annotate it manually at every binding point. Furthermore, while the inference of the morphisms involved via instance declarations would be possible, this definition provides no safeguard to guide the programmer, they would have to foresee those they will need. We can do better.

3.2 A Partial Order to Index

While the `bind` operator from Section 3.1 captures semantically what we are looking for, its generality, working over arbitrary monads, leaves both our type checker and our programmer quite in the dark. Looking back at Figure 4, we had identified the constellation of monads at play starting from observing that we might work with some signatures $(\emptyset, \text{Rd}, \text{Wr}, \text{Rd} + \text{Wr})$, and that these signatures are naturally ordered by inclusion.

We follow this intuition by requesting the user to provide the interface described in Figure 6. We assume a domain of indices `D` equipped with a partial order \subseteq — for our running example, we build the four valued diamond $\{\mathbf{v}, \mathbf{r}, \mathbf{w}, \mathbf{rw}\}$. Indices are mapped to the monads of interest via a `reify` function (written $[[_]]$).

Intuitively, the ordering placed over our indices abstracts the ability to transport computations across monadic structures. We formalize this requisite in the `reify_le` field: we request the user to explicit how to map proofs of inequality between indices to monad morphisms between their respective reification. We define `climbPO` as the function associating the corresponding `fmap` to a given inequality. Finally, we must ensure some coherence: given two monads `M` and

Figure 7: Hasse Diagram of the order used in `bindPO_bindPO`

\mathbb{N} , we intuitively want to consider at most one way to transport computations from \mathbb{M} to \mathbb{N} . In particular, the provided morphisms must be compatible with the associativity of the partial order, as spelled out in `reify_le_trans`.

Given such an interface, we can provide a specialization of Section 3.1’s operation: rather than provide a common target monad and morphisms, all we need is to pick an index above the two indexes involved in the computations being sequenced.

```

Definition bindPO i1 i2 i3 (LT1 : i1 ⊆ i3) (LT2 : i2 ⊆ i3) :
  ∀ X Y, [[i1]] X → (X → [[i2]] Y) → [[i3]] Y :=
  bindH (MMT := reify_le LT1) (MNT := reify_le LT2).

```

As before, the extended monadic laws remain valid. But where they used to depend on arbitrary morphisms, they are now specialized to ordered indices: everything relies on the canonical morphism specified by `reify_le`.

```

Lemma bindPO_ret_l : ∀ i1 i2 i3 (I13 : i1 ⊆ i3) (I23 : i2 ⊆ i3),
  ∀ X Y (k : X → [[i2]] Y) (x : X),
  bindPO I13 I23 (ret x) k = climbPO I23 (k x).

```

```

Lemma bindPO_ret_r : ∀ i1 i2 i3 (I13 : i1 ⊆ i3) (I23 : i2 ⊆ i3),
  ∀ X (m : [[i1]] X),
  bindPO I13 I23 m (ret (X := X)) = climbPO I13 m.

```

```

Lemma bindPO_bindPO : ∀ i1 i2 i3 i4 i5 i6,
  (I16 : i1 ⊆ i6) (I46 : i4 ⊆ i6) (I56 : i5 ⊆ i6) (I36 : i3 ⊆ i6)
  (I14 : i1 ⊆ i4) (I24 : i2 ⊆ i4) (I25 : i2 ⊆ i5) (I35 : i3 ⊆ i5),
  ∀ X Y Z (m : [[i1]] X) (k1 : X → [[i2]] Y) (k2 : Y → [[i3]] Z),
  bindPO I46 I36 (bindPO I14 I24 m k1) k2
  = bindPO I16 I56 m (fun x ⇒ bindPO I25 I35 (k1 x) k2).

```

As for `bindH_bindH`, the associativity law is based on a hierarchy of monads. But this time, their relation is expressed as an ordering of their indices, represented in Figure 7. Interestingly, the hypotheses of commutativity are now systematic consequences of `reify_le_trans`, and can therefore be omitted.

At its core, no work has of course been saved, all the morphisms at play must still be provided. But the necessary proof obligations are now cleanly identified, and the arguments to `bind` and to the monadic laws are now drawn from a much simpler structure.

We have shed some light on how to structure the constructions at hand, but have not addressed the other running issue: the unifying monad \mathbb{T} from Section 3.1 has now become a unifying index `i3`, but it still needs to be provided. We remove this thorn in the side by adding slightly more structure to our indexing domain.

3.3 A Directed Set to Resolve Indecisiveness

The heterogeneous bind still has too much freedom: from having the luxury of picking the target monad T of its choice, it can now pick its favourite target index $i3$. As a consequence, programmers have to either manually specify $i3$ for each call of `bindPO`, or to explicit the return type of each bind operation.

This freedom feels superfluous when coding: taking inspiration from our running example once again, it seems always safe to look at our index domain as having the structure of a semi-lattice and taking as target the join of the indices involved. In practice, picking the smallest upper bound does not matter for soundness, all we care about is to have a canonical way to chose a binary upper bound: we request the user to additionally equip their partial order with the structure of a directed set.

```
Variable join : D → D → D.
Infix "⊔" := join.
Hypothesis join_le_l : ∀ i1 i2, i1 ⊑ i1 ⊔ i2.
Hypothesis join_le_r : ∀ i1 i2, i2 ⊑ i1 ⊔ i2.
```

The heterogeneous bind specialises `bindPO` with the two inequalities about \sqcup . This time Coq can infer all type arguments from $(m : \llbracket i1 \rrbracket X)$ and $(k : X \rightarrow \llbracket i2 \rrbracket Y)$, discharging a lot of annotations from the programmer.

```
Definition bindL i1 i2 :
  ∀ X Y,  $\llbracket i1 \rrbracket X \rightarrow (X \rightarrow \llbracket i2 \rrbracket Y) \rightarrow \llbracket i1 \sqcup i2 \rrbracket Y :=$ 
  bindPO (join_le_l i1 i2) (join_le_r i1 i2).
```

The main function constituting our running example can thus be written with the desired syntax, with no additional notation, provided that the extended interface is provided. We furthermore observe that the user can provide a reification of the indexing diamond into either the free diamond (left part in Figure 4) or the semantic diamond (right part in Figure 4).

Depending on their choice, the exact same syntax, depicted to the right will be valid against opaque computations `init` and `fetch` living in respectively `free Wr` and `free Rd`, or in `write` and `read`. Its return type will be directly inferred as respectively `free (Rd +' Wr)` or `state` from the joins involved.

```
Definition main (n : data) :=
  init n ;;
  v1 ← fetch ;;
  v2 ← fetch ;;
  ret (v1 =? v2).
```

We have overlooked one detail in the code above: the typing of `ret (v1 =? v2)` remains ambiguous. However this is easily resolved. By definition, `ret` performs no effect in the monads, hence if the lattice is equipped with a \perp element (typically reified into the `pure` monad), then it is safe to always use this element as an index for `ret`: we can thus redefine `ret` to always live in this “minimal” monad.

In order to guide the type inference when performing a bind, we have added a computational device, `join`, to compute the index governing the monadic type of the result. Pandora’s box is open, we are performing non-trivial dependent programming. The consequences show up the moment we turn our attention to the monadic laws for `bindL`. Things remain straightforward for the two `ret` laws:


```
Lemma bindL_ret_l : ∀ i1 i2,
  ∀ X Y (k : X →  $\llbracket i2 \rrbracket Y$ ) (x : X),
  bindL (ret (M :=  $\llbracket i1 \rrbracket$ ) x) k = climbPO (join_le_r i1 i2) (k x).
```

```
Lemma bindL_ret_r : ∀ i1 i2,
  ∀ X (m :  $\llbracket i1 \rrbracket X$ ),
  bindL m (ret (M :=  $\llbracket i2 \rrbracket$ )) = climbPO (join_le_l i1 i2) m.
```


But the obvious statement for associativity does not even type check!


```
Fail Lemma bindL_bindL_fail : ∀ i1 i2 i3,
  ∀ X Y Z (m :  $\llbracket i1 \rrbracket$  X) (k1 : X →  $\llbracket i2 \rrbracket$  Y) (k2 : Y →  $\llbracket i3 \rrbracket$  Z),
  bindL (bindL m k1) k2 = bindL m (fun x ⇒ bindL (k1 x) k2).
```

Indeed, the expression `bindL (bindL m k1) k2` has type $\llbracket (i1 \sqcup i2) \sqcup i3 \rrbracket$ Z while the expression `bindL m (fun x ⇒ bindL (k1 x) k2)` has type $\llbracket i1 \sqcup (i2 \sqcup i3) \rrbracket$ Z. In other words, we need to assume the associativity of our domain of indices to express the associativity of our heterogeneous bind.

```
Hypothesis join_assoc : ∀ i1 i2 i3, i1  $\sqcup$  (i2  $\sqcup$  i3) = (i1  $\sqcup$  i2)  $\sqcup$  i3. 
```

Luckily, `climbP0` allows us to follow any inequality proof along our index domain. It can hence in particular be used to transport equality proofs in our types: by instantiating it over `join_assoc`, we get the `climb_assoc` transport function (we omit its code here) needed to state the bind associativity.

```
Definition climb_assoc : ∀ i1 i2 i3 :  $\llbracket i1 \sqcup (i2 \sqcup i3) \rrbracket$  ~  $\llbracket (i1 \sqcup i2) \sqcup i3 \rrbracket$ . 
```

```
Lemma bindL_bindL : ∀ i1 i2 i3, 
  ∀ X Y Z (m :  $\llbracket i1 \rrbracket$  X) (k1 : X →  $\llbracket i2 \rrbracket$  Y) (k2 : Y →  $\llbracket i3 \rrbracket$  Z),
  bindL (bindL m k1) k2 = climb_assoc (bindL m (fun x ⇒ bindL (k1 x) k2)).
```

At this point, we have reached a first milestone: we have identified a suitable interface to program in a heterogeneous way across a set of monads organised as the reification of a pointed set. We now turn our attention back to the free monad.

4 Support for the Free Monad

In the previous section, we have seen that indexing monads by a directed set whose order maps to monad morphisms between the reified monad provides an adequate interface to support heterogeneous programming among those monads. Looking once again back at Figure 4, one gets the feeling that we did not quite strike the bullseye. First, we had not one but two structured sets of monads in mind — the syntactic world with the various free monads, and the semantic counterpart with their interpretations. Furthermore, signatures themselves seemed to play a specific role. In this section, we provide yet another interface, from which we will derive two instances of the directed one, respectively for the syntax and the semantics. The full interface is depicted on Figure 8: we motivate and walk through its components in this section. As an additional visual support, Figure 9 depicts the data contained in the interface when instantiated to our running example.

A first intuition could be to consider that signatures themselves should form the indexing structure. After all, the disjoint sum (`+`) could be a candidate for `join`, while signature inclusion, written `<` and defined by the existence of an injection between the signatures, would constitute a valid order. However, disjoint sum is much too crude a `join` — at the very least, we would like to look more closely to a set-theoretical join — and we therefore still need to introduce a proper pointed set `D`. But on the syntactic side, we are indeed indexing signatures rather than monads: this is the purpose of the part of Figure 8 describing the first layer of reification. In contrast to the previous case, two ordered indices must here map to reified signatures such that the smallest one can be injected into the largest one, as witnessed by `Sinj`. We impose three coherence properties on this first layer of reification: the injections should be proof irrelevant, proofs of self relation should map to the identity injection, and proofs by transitivity should map to the composition of their respective injections.

```

(* The directed domain by which we index *)
Variable D : Type.
Djoin_le_l : ∀ d1 d2, d1 ⊆ d1 ⊔ d2;
Djoin_le_r : ∀ d1 d2, d2 ⊆ d1 ⊔ d2;
Djoin_assoc : ∀ d1 d2 d3, d1 ⊔ (d2 ⊔ d3) = d1 ⊔ d2 ⊔ d3;

(* First layer of reification: in terms of signatures *)
Sreify : D → signature;
Notation "'[[ i ]]" := (Sreify i);
(* To the order must correspond injections *)
Sinj : ∀ d1 d2, d1 ⊆ d2 → [[d1]] -< [[d2]];
(* Coherence properties of these injections *)
Sinj_proper : ∀ d1 d2 (I12 I12' : d1 ⊆ d2), Sinj I12 = Sinj I12';
Sinj_refl : ∀ d (I : d ⊆ d), Sinj I = inject_id;
Sinj_trans : ∀ d1 d2 d3 (I12 : d1 ⊆ d2) (I23 : d2 ⊆ d3),
  (Sinj I12) ∘ (Sinj I23) = Sinj (PreOrder_Transitive _ _ I12 I23);

(* Via the free construction, D reifies into the corresponding free monads *)
Freify d := free [[d]];
Notation "'{ i }'" := (Freify i);
Ffmap d1 d2 (I : d1 ⊆ d2) : {d1} ~> {d2} := fun m => translate (Sinj I) m;

(* Second layer of reification: in terms of concrete monads *)
Mreify : D → monad;
Notation "'<< i >>'" := (Mreify i).
Mmonad : ∀ d, Monad <<d>>;
(* To the order must correspond morphisms *)
Mmorph : ∀ d1 d2, d1 ⊆ d2 → MonadMorphism <<d1>> <<d2>>;
(* Coherence properties of these morphisms *)
Mfmap {d1 d2} (I : d1 ⊆ d2) X : <<d1>> ~> <<d2>> :=
  fmap <<d1>> ~> <<d2>> (MonadMorphism := Morph (Mmorph I)) X;
Mmorph_proper : ∀ d1 d2 (I J : d1 ⊆ d2), (Mfmap I) = (Mfmap J);
Mmorph_refl : ∀ d (I : d ⊆ d) X (m : <<d>> X), Mfmap I m = m;
Mmorph_trans : ∀ d1 d2 d3 (I12 : d1 ⊆ d2) (I23 : d2 ⊆ d3),
  (Mfmap I12) ∘ (Mfmap I23) = Mfmap (PreOrder_Transitive I12 I23);

(* Both layers are finally connected by handlers *)
Dh : ∀ d, [[d]] ~> <<d>>;
I d : {d} ~> <<d>> := interp (Dh d);
(* Climbing in the syntax and then interpreting or interpreting and then
climbing in the semantics should commute *)
I_commut : ∀ d1 d2 (I : d1 ⊆ d2) X (m : {d1} X),
  I (Ffmap I m) = Mfmap I (I m)

```

Figure 8: Interface for free monadic computations (👉)

Turning our eye to Figure 9, we have so far covered the upper part of the figure. The indexing diamond maps to the one containing the four signatures introduced in Section 2 via `Sreify` (also noted `[[_]]`). By the `free` construction, signatures get mapped to all the free monads at which we might want to write programs — hence by composition, reification into signature induces a reification `Freify` (noted `{_}`) into free monads. Non-explicitly represented are the nature and relationship between the arrows in each of these three boxes: to a proof of inequality corresponds an injection of signatures, to which correspond a monad morphism between the corresponding free monads — the latter is derived for free.

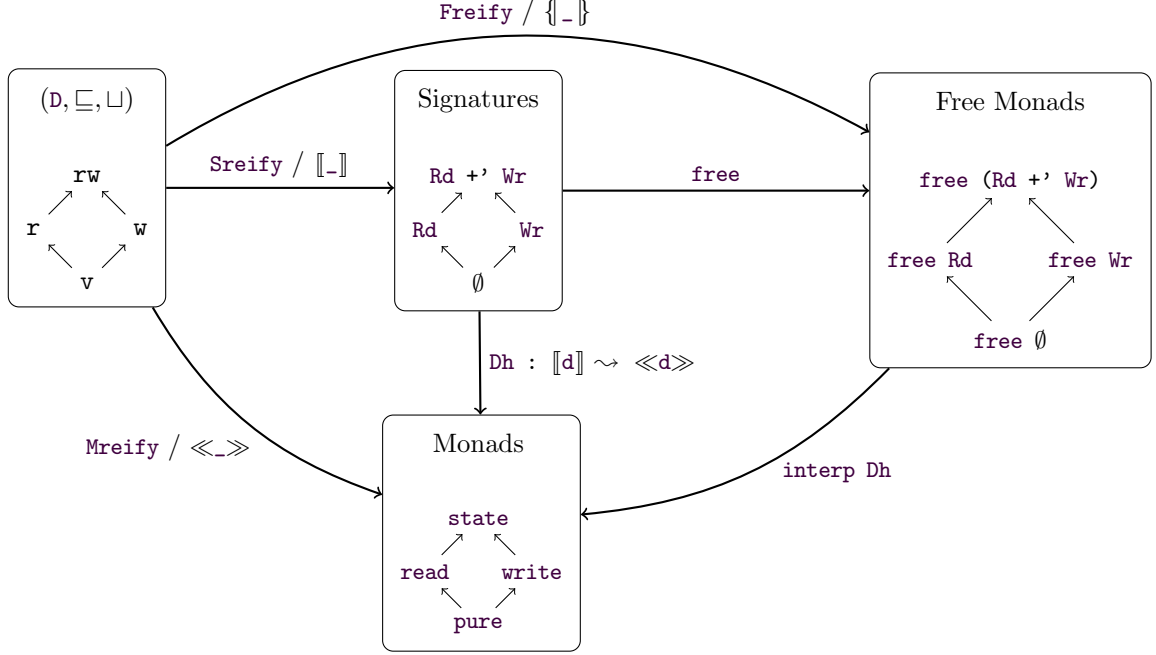


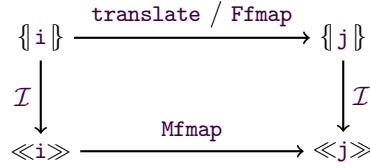
Figure 9: The full picture instantiated on our running example (🔥)

We now bring into the picture the monads in which we concretely implement our effectful operations. This is the purpose of the second layer of reification from Figure 8: `Mreify` (also noted \ll_\gg) maps the same domain of indices \mathbb{D} to concrete monads. This part of the interface is very close to the one developed in Section 3.3: indices must be reified into monads, and proofs of inequality must map to monad morphisms. The coherence properties are identical to the ones for the first layer of reification. On Figure 9, we have now introduced a fourth diamond, containing the state monad and its simplified versions, and the left arrow.

It remains only to relate the “free” diamond to the concrete one: we need to explain how we can implement the relevant signatures — this is captured by Dh . The user must provide at each index d a handler ($Dh\ d : \ll d \gg \rightsquigarrow \ll d \gg$): the `Rd` events must be implemented in the `read` monad, the `Rd +' Wr` ones in the state monad, and so on. Much like the reification into signatures gives rise to a reification into the free monads, this family of handlers entails a bridge \mathcal{I} between the diamond of free monads and the one of concrete monads via the `interp` function, completing the picture.

There remains one unstated coherence condition. Given two indices i and j such that $i \sqsubseteq j$, one may now follow two paths, represented in Figure 10. On one hand, we can translate the signature $\ll i \gg$ into $\ll j \gg$ (property `Sinj` in Figure 8), and thus translate any tree in $\{i\}$ into $\{j\}$ —this is the role of the `translate` function in the definition of `Ffmap`. We can then interpret the resulting computation into $\ll j \gg$. On the other hand, we may interpret $\ll i \gg$ into $\ll i \gg$, and then transport $\ll i \gg$ into $\ll j \gg$. Condition `I_commut` ensures that this diagram commutes.

From an instance of the interface from Figure 8, we derive three essential facts. The first is an instance of the directed interface indexed by \mathbb{D} w.r.t. `Freify`. The second is an instance of the directed interface indexed by \mathbb{D} w.r.t. `Mreify`. We can hence sequence heterogeneous com-


 Figure 10: Representation of the $\mathcal{I}_{\text{commut}}$ property

putations in both the syntactic and semantic world: let us write these operations respectively `Fbind` and `Mbind`. The last important result we derive is a theorem for commuting interpretation with binds in this melting pot:

```
Theorem bind_I : ∀ d1 d2 X Y (m : free [d1] X) (k : X → free [d2] Y),
  I (Fbind m k) = Mbind (I m) (fun x => I(k x)).
```

We have fulfilled the core of our contract by axiomatizing an interface allowing us to write the tightly typed version of `main` without additional crust. But even more importantly, it allows us to switch the perspective we have on the semantics of a compound computation, obtained by monadic interpretation, into the composition of the generally simpler implementation of its sub-components. Equipped with the minimal ingredients necessary to express and leverage effect typing information, we now sketch some early avenues to try and take advantage of it.

5 Reasoning

Sections 3 and 4 have depicted the core of our current contribution: a clean specification of the necessary interface to enable (1) programming in the free monad over a variety of signatures, (2) mapping each computation to an adequate structure of implementation, (3) retaining equational reasoning through notably generalized monadic laws and commutation of interpretation with heterogeneous binds. We now reach the current limits of this work in progress: the principled way to leverage typing information has yet to be identified, and no case study at sufficient scale has yet been performed to back up a particular ad-hoc approach. We hence only modestly present a few directions we have been exploring, and ornate them with preliminary remarks.

Recall Section 2's task: can we prove that `main n` will always return `true`. This could be specified extensionally, breaking the monadic abstraction: $\forall c, \text{snd } (\mathcal{I} (\text{call } n)) = \text{true}$. It seems more appealing to introduce a unary Hoare logic over `state`.

```
Definition pre_state := (cell → Prop).
Definition post_state X := (X → cell → Prop).
Definition Hoare_state X : pre_state → state X → post_state X → Prop :=
  fun P m Q => ∀ c, P c → let '(c', x) := m c in Q x c'.
```

Our goal hence become:

```
Theorem main_correct : ∀ (n : nat),
  Hoare_state (fun _ => True) (I (main n)) (fun v _ => v = true).
```

Using `bind_I`, the computation becomes explicitly the sequence of a computation in `write` with a computation in `read`. Using a cut rule for our Hoare logic, we can eliminate the former at the trivial postcondition, and are hence left with reasoning about the remaining read only computation. A quite adhoc way to conclude is to prove a general law of read only computations: when called twice in sequence, they lead to the same result.

```

Lemma read_twice (m : free Rd nat) :
  ∀ c,  $\mathcal{I}(v1 \leftarrow m ;;$ 
    v2  $\leftarrow m ;;$ 
    ret (v1 =? v2)) c = true.

```

This lemma breaks the abstraction introduced by `read`, but solves our goal immediately. It leverages our ability to keep track of the type of `fetch` to derive a generic invariant of the computation. From this perspective, it is a satisfying result, and the one we currently propose in our formal development.

In the remaining of this section, we discuss potential avenues to go further.

Finer grain invariants breaking abstraction. Although `read_twice` is generic over arbitrary computations (`m : free Rd nat`), it still fills handcrafted specifically for our problem. One may wonder whether the primitive fact expressing that read only computations may not affect the cell may suffice. Similarly, pure and write-only computations can be explicitly proved to not depend on the initial state.

```

Lemma read_invariant_explicit : ∀ X (m : read X) c,
  let (cf,x) := fmapread state m c in c = cf.

```

```

Lemma write_invariant_explicit : ∀ X (m : write X) c c',
  let (cf,x) := fmapwrite state m c in
  let (cf',x') := fmapwrite state m c' in
  x = x' ∧ ((cf = c ∧ cf' = c') ∨ cf = cf').

```

```

Lemma pure_invariant_explicit : ∀ X (m : pure X) c c',
  let (cf,x) := fmappure state m c in
  let (cf',x') := fmappure state m c' in
  x = x' ∧ cf = c ∧ cf' = c'.

```

However, these invariants require to completely break the abstractions in order to be useful. On a simple example such as `main`, managing to reduce enough the computation to exhibit the sub-parts needed without simply computing the whole thing has proved unreasonably cumbersome for the expected benefit. It would nonetheless be interesting to see if it can be done, and may be worth it when working with coinductive computations such as the ones generated by `ITrees`, as (internal to `Coq`) reduction is not possible in this context.

Internalizing `read_state_invariant_explicit` into `hoare_state`. Rather than explicitly exposing the underlying computation, one could try to express the invariants as `state_hoare` triplets valid for any computation lifted from a given monad. Expressing that the state is left unchanged in this style is however quite annoying. One option is to change the postcondition so that it quantifies additionally over the initial state before the execution as well. Another is to use a meta-variable in order to bind as an equality in the precondition the initial value of the cell, and use the same meta-variable in the postcondition.

```

Lemma read_state_invariant : ∀ X v (m : read X),
  Hoare_state (fun c => c = v) (fmapread state m) (fun _ c => c = v)

```

However, in both cases, there seems to be no obvious way to leverage such a rule to ensure that the second call to `fetch` would return the same value.

Furthermore, it is unclear how one would internalize in `state_hoare` the write only invariant `write_invariant_explicit`.

```

Definition post_pure X := (X → Prop).
Definition Hoare_pure X : pure X → post_pure X → Prop :=
  fun m Q ⇒ Q m.

Definition pre_read := (cell → Prop).
Definition post_read X := (X → Prop).
Definition Hoare_read X : pre_read → read X → post_read X → Prop :=
  fun P m Q ⇒ ∀ c, P c → let x := m c in Q x.

Definition post_write X := ((X → Prop) * (cell → Prop)).
Definition Hoare_write X : write X → post_write X → Prop :=
  fun m '(Qx, Qc) ⇒ let '(x, mc) := m in
    match mc with
    | None ⇒ Qx x
    | Some c ⇒ Qx x ∧ Qc c
  end.

```

Figure 11: Hoare-style proof system for pure, read only and write only monads

Hoare-style abstraction at each level. A strong temptation is to avoid breaking any abstraction. We have worked hard to preserve information about the structure in which each computation has been built, in a way compatible with equational reasoning. We would like to follow the usual approach and pair it cleanly with Hoare-style reasoning in the appropriate structure.

Naturally, this suggests that we should associate a Hoare-style proof system to each monad at play. For our running case study, we hence need a diamond of hoare systems: the missing components can be found in Figure 11. While more appealing, similar difficulties arise: for instance, read only computations seem to require a meta-variable to be lifted.

```

Lemma read_invariant : ∀ X v (m : read X) (P : pre_read) (Q : post_read X),
  Hoare_read P m Q →
  Hoare_state (fun c ⇒ P c ∧ c = v) (fmapread state m) (fun x c ⇒ c = v ∧ Q x).

```

Furthermore, we are here working in an ad-hoc fashion, handcrafting proof systems for our case study. In order to state generic laws about these systems, and notably w.r.t. their interactions with `fmap`, we would need to move them to the interface. However identifying a universal shape to monadic deductive systems is non-trivial: it would likely have to rely on porting Maillard et al.'s work on Dijkstra monads [9].

We plan on conducting additional case studies at medium scale to better identify the kind of reasoning we may want to perform, and hopefully derive the adequate framework to do so.

6 Discussion and Future Work

The initial motivation for this work emerged from a Coq formalisation of R [1]. This formalisation is a large monadic interpreter of roughly 18,000 lines of code, with about two thousand `bind` applications. Proving some safety invariant within this formalisation (for instance memory safety) is doable, but requires a substantial proof effort due to the size of the interpreter: invariant lemmas have to be stated and proven for every function. Frustratingly, a lot of these lemmas felt obvious, as most functions only use a small subset of effects.

To illustrate this intuition, we categorised the interpreter functions involved in the formalization by the effects they used: Figure 12 shows the different categories that we identified.

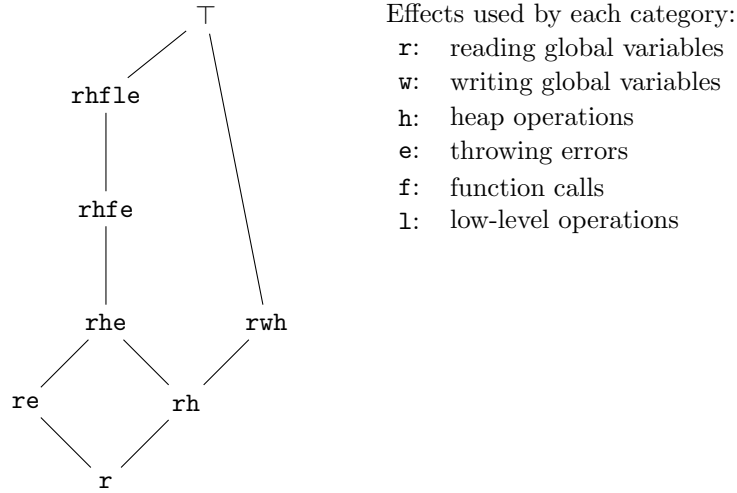


Figure 12: Hasse diagram for the different effects within a R monadic interpreter [1].

An interesting part of this diagram are the categories that are not shown: for instance, we did not find any interpreter functions that could raise an error (effect e) without reading the interpreter’s global variables (effect r), thus the re category is shown in the diagram, but not e . Almost all interpreter functions read these global variables, but only a small number write them: these are the functions initialising the interpreter at the very beginning of the execution, never to be called back again. In figure 12, these initialising functions fall into the rwh category. As a consequence, for most of the interpreter execution, these variables are constant by construction: no interpreter function syntactically call the write operations. The invariant that these global variables are constant after the initialisation phase thus ought to be simple to prove, but it turned out to be cumbersome in practice. We expect the \top category to be almost empty, with the sole exception of the main function that sets up the initialisation and calls the read-eval-print loop.

We believe that the ongoing work we present in this paper may help ease the points observed in this large scale project: the categories of Figure 12 fit well into the directed set of Section 3.3. We could thus imagine rewriting the interpreter within this formalism. As the approach presented in this work is based on a `bind` operation whose usage is very close to the usual monadic `bind`, we do not expect this conversion to require a significant amount of work. During type inference, Coq will compute where each interpreter function falls within the directed set. This enables a first phase of verification for the programmer: for instance, if a function is inferred to be of type \top (which in the context of R is expected to be very rare), something may be wrong and is worth inspecting. Furthermore, each of these categories provides some invariants: the sketches of methodology presented in Section 5 should be improved upon to cleanly state, prove, and leverage these invariants. In the case of this R interpreter, the size of the interpreter is large compared to the number of monads involved: we expect to get similar results to the ones presented in a cumbersome way in [1], but with less work.

References

- [1] Martin Bodin, Tomás Diaz, and Éric Tanter. A trustworthy mechanized formalization of R. In *Dynamic Languages Symposium, DLS*, 2018.
- [2] Venanzio Capretta. General recursion via coinductive types. *Log. Methods Comput. Sci.*, 1(2), 2005.
- [3] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. Choice trees. *POPL*, 2023.
- [4] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 71–84. ACM Press, 1993.
- [5] Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. C4: Verified transactional objects. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022.
- [6] Thomas Letan and Yann Régis-Gianas. Freespec: specifying, verifying, and executing impure computations in Coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, pages 32–46. ACM, 2020.
- [7] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effect handlers in Coq. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik P. de Vink, editors, *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018*, volume 10951 of *Lecture Notes in Computer Science*, pages 338–354. Springer, 2018.
- [8] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effect handlers in Coq. In *Formal Methods - 22nd International Symposium, FM 2018*, pages 338–354, 2018.
- [9] Kenji Maillard, Cătălin Hrițcu, Exequiel Rivas, and Antoine Van Muylder. The next 700 relational program logics. *POPL*, 4, 2020.
- [10] Conor McBride. Turing-completeness totally free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, pages 257–275, 2015.
- [11] Eugenio Moggi. *Computational Lambda-Calculus and Monads*. IEEE Computer Society Press, 1989.
- [12] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.
- [13] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.
- [14] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees. *POPL*, 4, 2020.
- [15] Irene Yoon, Yannick Zakowski, and Steve Zdancewic. Formal reasoning about layered monadic interpreters. *Proc. ACM Program. Lang.*, 6(ICFP):254–282, 2022.
- [16] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *ICFP*, 5, aug 2021.