Program Adverbs and Tlön Embeddings

YAO LI, University of Pennsylvania, USA
STEPHANIE WEIRICH, University of Pennsylvania, USA

Free monads (and their variants) have become a popular general-purpose tool for representing the semantics of effectful programs in proof assistants. These data structures support the compositional definition of semantics parameterized by uninterpreted events, while admitting a rich equational theory of equivalence. But monads are not the only way to structure effectful computation, why should we limit ourselves?

In this paper, inspired by applicative functors, selective functors, and other structures, we define a collection of data structures and theories, which we call *program adverbs*, that capture a variety of computational patterns. Program adverbs are themselves composable, allowing them to be used to specify the semantics of languages with multiple computation patterns. We use program adverbs as the basis for a new class of semantic embeddings called *Tlön embeddings*. Compared with embeddings based on free monads, Tlön embeddings allow more flexibility in computational modeling of effects, while retaining more information about the program's syntactic structure.

ACM Reference Format:
Yao Li and Stephanie Weirich. 2022. Program Adverbs and Tlön Embeddings. In . ACM, New York, NY, USA, 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Suppose that you want to formally verify a program written in your favorite language—be it Verilog, Haskell, or C—your first step would be to translate that program and a description of its semantics to a formal reasoning system, such as Coq [Coq development team 2021]. This step is known as *semantic embedding* [Boulton et al. 1992].

There are multiple approaches to semantic embeddings. The two most well-known were proposed by Boulton et al. [1992]: *shallow embeddings*, which represent terms of the embedded language using equivalent terms of the embedding language, and *deep embeddings*, which represent terms using abstract syntax trees (ASTs) and represent their semantics via some interpretation function.

Shallow embeddings are convenient because they are simple, but they have their limitations. It is impossible to use them to state and reason about properties related to syntax, because they do not retain the syntactic structure of the original program. Furthermore, shallow embeddings fix a single semantics, so they are less robust to changes in program interpretations. Such edits require changing the translation process, in addition to the semantic domain (*i.e.,* the type used for representing the semantics of the embedded language).

On the other hand, deep embeddings are more modular thanks to an extra layer (*i.e.,* the AST) that defines the syntax of the embedded language. When we need to change the semantics, we only need to change the *interpretation* that maps the AST into some semantic domain—the translation to the formal reasoning system remains unchanged. Furthermore, the AST makes it possible to state and prove properties related to the original program's syntactic structure. The downside is that interpreting and reasoning about properties based on this AST takes more effort than with shallow embeddings.

The pros and cons make it hard to choose between shallow and deep embeddings. Fortunately, we don't need to commit to a single option. We can use *mixed embeddings*, a style of embedding

ICFP'22, 2022, ...

that includes characteristics of each. In this style, parts of a language are embedded "shallowly" while other parts are embedded "deeply". However, in any mixed embedding, we must ask: where should we draw the line to separate the shallowly embedded part from the deeply embedded part?

Recent efforts have focused on mixed embeddings based on freer monads [Kiselyov and Ishii 2015] or their variants [Dylus et al. 2019; McBride 2015; Swamy et al. 2020; Xia et al. 2020]. The style has been shown useful for representing and reasoning about effectful computation in various applications [Chlipala 2021; Christiansen et al. 2019; Foster et al. 2021; Letan et al. 2021; Nigron and Dagand 2021; Zakowski et al. 2021; Zhang et al. 2021]. Beyond these applications, this style points out a useful guideline for answering the question above. That is: modeling the pure parts of the computation "shallowly" and the effectful parts "deeply".

Our work builds on this idea of separating pure and effectful parts in a mixed embedding, but inspects the following question: Why freer monads? We find that this is because freer monads model *one* general computation pattern that is common in many languages. However, the finding also implies that there are other computation patterns not captured by freer monads.

Following this observation, we propose a new class of mixed embeddings called *Tlön embeddings*.[1] Tlön embeddings model programs using structures called *program adverbs*, which are reifications of familiar type classes (*e.g.*, `Applicative`, `Selective`, `Monad`) paired with equational theories. Like freer monads, these free structures can be used to combine shallowly embedded pure computation with deeply embedded computational effects. However, program adverbs provide choices in the semantics through the selection of the structure and equational theory. For example, the "statically" adverb, based on applicative functors and their free theory, models computation where control flow and data flow in the semantics are fixed. Or, by modifying the equational theory of the free applicative structure to include commutativity, we can describe computation that is "statically and in parallel".

We make the following contributions:

- We compare the trade-offs of different styles of semantic embeddings in the context of formal reasoning and propose Tlön embeddings (Section 2).
- We define program adverbs and show how to define their syntactic parts and their semantic parts (Section 3).
- We refactor program adverbs to support composition and extension. We motivate why we want to compose program adverbs and define a composition algebra (Section 4).
- We implement composable program adverbs using the Coq proof assistant. A major challenge for implementing them in Coq is supporting extensible inductive data types [Wadler 1998]. We show one way of addressing this challenge by adapting the *Meta Theory à la Carte* (MTC) approach [Delaware et al. 2013] (Section 4).
- We identify five basic program adverbs from commonly used type classes in Haskell and we prove that these program adverbs are sound (Section 3). We also identify two add-on program adverbs that are used in combination with basic program adverbs (Section 4).
- We demonstrate the usefulness of program adverbs via three distinct language examples including a simple circuit language (Section 2), Haxl [Marlow et al. 2014], and a networked server adapted from Koh et al. [2019] (Section 5).

Additionally, we discuss other aspects of our work in Section 6 and the related work in Section 7.

[1]The name Tlön embedding is a reference to the novel *Tlön, Uqbar, Orbis Tertius* by Jorge Luis Borges. In the novel, Tlön is an imaginary world, where its parent language does not have any nouns, but only "impersonal verbs, modified by monosyllabic suffixes (or prefixes) with an adverbial value" [Borges 1940].

literals	b	$::=$ `true` $\mid$ `false`
terms	t, u	$::= x \mid b \mid \neg t \mid t \wedge u \mid t \vee u$

Fig. 1. The syntax of $\mathcal{B}$.

2 SEMANTIC EMBEDDINGS

In this section, we first demonstrate different forms of semantic embeddings using a simple circuit language called $\mathcal{B}$ and compare how each form of embedding can be used to reason about programs written in this language. To distinguish the embedded language and the embedding language, we use mathematical notation to describe $\mathcal{B}$ and use Coq code to describe its embeddings.

The syntax of $\mathcal{B}$ appears in Fig. 1. Semantically, we want the boolean operators to have their usual semantics. However, $\mathcal{B}$ can read from the variables that represent references to external devices and we don't want to fix those values in the semantics. Furthermore, we don't know if the values are immutable: they might change over time, or they might change after each read, *etc.*

The four embeddings that we consider in this section are defined in the right column of Fig. 2. We use $[\![\cdot]\!]_S$, $[\![\cdot]\!]_D$, $[\![\cdot]\!]_M$, and $[\![\cdot]\!]_A$ to represent the translation from a term of $\mathcal{B}$ to shallow, deep, and two mixed embeddings, respectively. These translations refer to the definitions in the left column as well as to the standard classes and notations for functors, monads, *etc.* shown in Fig. 3.

To compare embeddings, we will use each to consider the following questions regarding the semantics of $\mathcal{B}$:

(1) Is x equivalent to $x \wedge x$?
(2) Is x equivalent to $x \wedge \text{true}$?
(3) Is $t \wedge u$ equivalent to $u \wedge t$?
(4) Is the number of variable accesses at its runtime always less than or equal to 2 to the power of the circuit's depth?

Because we are modeling a circuit language that uses unknown external devices, we don't want to be able to prove or disprove property (1). This property may hold or not hold depending on the situation. If the external devices are immutable, this property will be true. Otherwise, we may be able to falsify it. In contrast, we would like our semantic embedding to give us tools to verify properties (2) and (3) because these properties should hold regardless of the properties of our external device. The former holds because on both sides of the equivalence relation we have only accessed the variable x once. The latter is a property of circuits in general: it says that the operands of $\wedge$ are computed in parallel. The last property (4) relates a dynamic property of the semantics (the number of variable accesses) to a syntactic property of the circuit (the size of the circuit itself).

2.1 A Shallow Embedding

To use a shallow embedding to represent the semantics of $\mathcal{B}$, we need a way to represent the effects of reading from external devices—the most common way of doing this is using *monads* (Fig. 3). But *which* one? A simple option is the reader monad. We show core definitions of a specialized reader monad at the top left of Fig. 2.[2] The translation from $\mathcal{B}$ to `Reader bool` is given in the same figure. Following the terminology used by Svenningsson and Axelsson [2012], we call `Reader bool` the *semantic domain* of our shallow embedding. Of course, the reader monad is just one possible semantic

[2]For simplicity, we specialize the monad so that its environment has type `var -> bool`. The commonly used reader monad is more general that the type of its environment is parameterized.

Shallow Embedding

```
Definition Reader (A : Type) : Type :=
  (var -> bool) -> A.
Definition ret {A} (a : A) : Reader A :=
  fun _ => a.
Definition bind {A B} (m : Reader A)
  (k : A -> Reader B) : Reader B :=
    fun v => k (m v) v.
Definition ask (k : var) : Reader bool :=
  fun m => m k.
```

$$
\begin{aligned}
[\![\cdot]\!]_S &: \texttt{Reader bool}\\
[\![\texttt{true}]\!]_S &= \texttt{ret true}\\
[\![\texttt{false}]\!]_S &= \texttt{ret false}\\
[\![x]\!]_S &= \texttt{ask x}\\
[\![\neg t]\!]_S &= \texttt{negb <\$>}\ [\![t]\!]_S\\
[\![t \wedge u]\!]_S &= \texttt{t'<-}\ [\![t]\!]_S\texttt{; u'<-}\ [\![u]\!]_S\texttt{;}\\
&\quad\ \texttt{ret (andb t' u')}\\
[\![t \vee u]\!]_S &= \texttt{t'<-}\ [\![t]\!]_S\texttt{; u'<-}\ [\![u]\!]_S\texttt{;}\\
&\quad\ \texttt{ret (orb t' u')}
\end{aligned}
$$

Deep Embedding

```
Inductive term :=
| Var (v : var)
| Lit (b : bool)
| Neg (t : term)
| And (t : term) (u : term)
| Or (t : term) (u : term).
```

$$
\begin{aligned}
[\![\cdot]\!]_D &: \texttt{term}\\
[\![\texttt{true}]\!]_D &= \texttt{Lit true}\\
[\![\texttt{false}]\!]_D &= \texttt{Lit false}\\
[\![x]\!]_D &= \texttt{Var x}\\
[\![\neg t]\!]_D &= \texttt{Neg}\ [\![t]\!]_D\\
[\![t \wedge u]\!]_D &= \texttt{And}\ [\![t]\!]_D\ [\![u]\!]_D\\
[\![t \vee u]\!]_D &= \texttt{Or}\ [\![t]\!]_D\ [\![u]\!]_D
\end{aligned}
$$

Freer Monad Embedding

```
Inductive FreerMonad (E : Type -> Type) R :=
| Ret (r : R)
| Bind {X} (m : E X)
    (k : X -> FreerMonad E R).
Fixpoint bind {E A B} (m : FreerMonad E A)
  (k : A -> FreerMonad E B) : FreerMonad E B :=
  match m with
  | Ret r => k r
  | Bind m' k' =>
     Bind m' (fun a => bind (k' a) k) end.
Variant DataEff : Type -> Type :=
| GetData (v : var) : DataEff bool.
```

$$
\begin{aligned}
[\![\cdot]\!]_M &: \texttt{FreerMonad DataEff bool}\\
[\![\texttt{true}]\!]_M &= \texttt{Ret true}\\
[\![\texttt{false}]\!]_M &= \texttt{Ret false}\\
[\![x]\!]_M &= \texttt{Bind (GetData x) Ret}\\
[\![\neg t]\!]_M &= \texttt{negb <\$>}\ [\![t]\!]_M\\
[\![t \wedge u]\!]_M &= \texttt{t'<-}\ [\![t]\!]_M\texttt{; u'<-}\ [\![u]\!]_M\texttt{;}\\
&\quad\ \texttt{Ret (andb t' u')}\\
[\![t \vee u]\!]_M &= \texttt{t'<-}\ [\![t]\!]_M\texttt{; u'<-}\ [\![u]\!]_M\texttt{;}\\
&\quad\ \texttt{Ret (orb t' u')}
\end{aligned}
$$

Reified Applicative Embedding

```
Inductive ReifiedApp (E : Type -> Type) R :=
  | EmbedA (e : E R)
  | Pure (r : R)
  | LiftA2 {X Y} (f : X -> Y -> R)
    (a : ReifiedApp E X) (b : ReifiedApp E Y).
```

$$
\begin{aligned}
[\![\cdot]\!]_A &: \texttt{ReifiedApp DataEff bool}\\
[\![\texttt{true}]\!]_A &= \texttt{Pure true}\\
[\![\texttt{false}]\!]_A &= \texttt{Pure false}\\
[\![x]\!]_A &= \texttt{EmbedA (GetData x)}\\
[\![\neg t]\!]_A &= \texttt{negb <\$>}\ [\![t]\!]_A\\
[\![t \wedge u]\!]_A &= \texttt{LiftA2 andb}\ [\![t]\!]_A\ [\![u]\!]_A\\
[\![t \vee u]\!]_A &= \texttt{LiftA2 orb}\ [\![t]\!]_A\ [\![u]\!]_A
\end{aligned}
$$

Fig. 2. Semantic embeddings of $\mathcal{B}$ in Coq. We use the infix operator <$> to represent a functor's fmap method and a notation similar to Haskell's do notation to represent monadic binds. The functions negb, andb, and orb are Coq's functions defined on the bool type.

```
Class Functor (F : Type -> Type) :=
  { fmap : forall {A B}, (A -> B) -> F A -> F B }.

Class Applicative (F : Type -> Type) `{Functor F} :=
  { pure   : forall {A}, A -> F A ;
    liftA2 : forall {A B C}, (A -> B -> C) -> F A -> F B -> F C }.

Class Selective (F : Type -> Type) `{Applicative F} :=
  { selectBy  : forall {A B C}, (A -> ((B -> C) + C)) -> F A -> F B -> F C }.

Class Monad (F : Type -> Type) `{Applicative F} :=
  { ret : forall {A}, A -> F A ;
    bind : forall {A B}, F A -> (A -> F B) -> F B }.
```

Default fmap definitions

```
Definition fmap_monad {m} `{Monad m} {a b} (f : a -> b) (x : m a) : m b :=
  x >>= (fun y => ret (f y)).
Definition fmap_ap {t}`{Applicative t}{a b} (f: a -> b) (x : t a) : t b :=
  liftA2 id (pure f) x.
```

Fig. 3. Coq type classes for functors, applicative functors [McBride and Paterson 2008], selective functors [Mokhov et al. 2019], and monads [Moggi 1991; Wadler 1992], as well as default definitions of fmap.

domain, other candidates include Dijkstra monads [Swamy et al. 2013], predicate transformer semantics [Swierstra and Baanen 2019], *etc.*

Using the reader monad, we can prove that property (1) is true, using ($\simeq_S$), a point-wise equivalence relation on our semantic domain of the reader monad. More specifically, we can prove the following Coq theorem:

```
forall x, ask x ≃S x1 <- ask x; x2 <- ask x; ret (andb x1 x2)
```

We "ask" twice on the right hand side of the equivalence to model accessing variable x twice during program runtime. However, `x1` equals to `x2` in our case since nothing has changed the global store. After proving that, the theorem can be proved via a case analysis on `x1`.

However, note that our proof relies on "nothing has changed the global store," but we don't know if this is true, as we don't know anything about the characteristics of the external device. Indeed, property (1) should *not* be true if we have a device where its values change over time: the value of x might change between two variable access. This is a problem with our choice of semantic domain. By choosing the reader monad, we introduce more assumptions over the semantics of $\mathcal{B}$, which results in proving a property that is not supposed to be true in the original language $\mathcal{B}$.

Although this is not a problem with the approach of shallow embedding—we can choose a different monad than the reader monad, the style does force us to choose a concrete semantic domain early. In practice, we sometimes need to change the semantic domain, either because we made a wrong assumption or because the language evolves. With shallow embeddings, we would need to change the entire translation process to change this domain.

Unlike property (1), property (2) is true even though we don't know anything about the external device. This is because on both sides of the equivalence relation we have only accessed the variable x once. Property (2) can be stated as follow with our shallow embedding:

```
forall x, ask x ≃S x1 <- ask x; ret (andb x1 true)
```

The proof follows from the theories of Coq's `bool` type and the `Reader` monad. However, even though this property should be true regardless of the external device, our mechanical proof still relies on the assumption that the external device is immutable—this is again because the property is stated in terms of the reader monad. If we change the shallow embedding to use a different semantic domain, we would need to prove this property again.

Property (3) is true and we can prove it to be true using our shallow embedding, but that is just a lucky hit. Even though we know nothing about the external device, there is a bisimulation between $t \wedge u$ and $u \wedge t$ because the two operands t and u run in parallel in a circuit. A proof based on our shallow embedding would, on the other hand, be based on the wrong assumption that the external device is immutable.

We cannot state property (4) with our shallow embedding. Our shallow embedding does not retain the syntactic structure of the original program so we cannot define a function that calculates the depth of the circuit.

2.2 A Deep Embedding

In a deep embedding, we first define an abstract syntax tree (AST) for $\mathcal{B}$. For example, we can use the `term` data type shown in Fig. 2. Our translation from $\mathcal{B}$ to the `term` is shown in the same figure. Note that the `term` data type does not encode *any* semantic meaning.

Without an interpretation, we cannot prove any of the first three properties. This is actually ideal for answering question (1) since we know nothing about the external device so we should not be able to prove it (nor should we be able to prove it wrong!). However, by leaving the entire syntax tree uninterpreted we are now unable to prove property (2) or (3), either.

A way out of this quandary is to define a coarser *equivalence relation* for ASTs and use that relation in the statement of properties (2) and (3). For example, we can interpret each `term` using the reader monad (as in the shallow embedding) and use the point-wise equivalence relation for that type. The proofs are essentially the same as the above.

One advantage of the deep embedding in this case is that, if we would like to change our definition of equivalence, we can do so by choosing a different *interpretation* without changing the translation process. In other words, deep embeddings achieve better modularity by introducing an intermediate layer. The price, however, is that it takes effort to build that extra intermediate layer. This extra effort seems small here, but can become tedious with some languages, *e.g.,* those with features like "let" that introduce variable bindings [Aydemir et al. 2005].

However, we still face a similar problem with the shallow embedding: If we would like to change the interpretation in our definition of equivalence, we need to prove our properties again. This suggests that another intermediate layer between deep and shallow embeddings might be helpful, as we will see in the next subsection.

The primary benefit we have by using the deep embedding is that we can now state and prove property (4). This is because the deep embedding gives us a representation of the program's original syntactic structure. This allows us to define the following function that counts the depth of a circuit:

```
Fixpoint depth (t : term) : nat :=
  match t with
   | Var _ => 0
   | Lit _ => 0
```

Left identity	:	`ret a >>= h = h a`
Right identity	:	`m >>= ret = m`
Associativity	:	`(m >>= g) >>= h = m >>= (fun x => g x >>= h)`

Fig. 4. The monad laws. The >>= symbol is the infix operator for bind. Proving these laws for FreerMonad in Coq relies on the axiom of functional extensionality.

```
| Neg t => depth t + 1
| And t u => max (depth t) (depth u) + 1
| Or t u => max (depth t) (depth u) + 1
end.
```

Since we assume a straightforward semantics for $\mathcal{B}$, the number of variable access at runtime equals to the number of variables appeared in a `term`, so we can directly prove property (4) by an induction over the `term` data type.

2.3 A Mixed Embedding Based on Freer Monads

A semantic embedding can be partially shallow and partially deep. We use the term *mixed embeddings* to describe embeddings with this property. One style of mixed embeddings that is popular today is based on *freer monads* [Chlipala 2021; Dylus et al. 2019; Letan et al. 2021; McBride 2015; Nigron and Dagand 2021; Swamy et al. 2020; Xia et al. 2020]. In this type of mixed embeddings, the pure parts of the program are embedded shallowly, while effects are embedded deeply (and abstractly) using algebraic data types "connected" by freer monads.

The core definitions of freer monads are in the left column of Fig. 2. The `FreerMonad` data type is parameterized by an abstract effect `E` of `Type -> Type` and a return type `R` of `Type`. Conceptually, it collects all the deeply embedded effects `E` in a right-associative monadic structure.

For any effect type, `FreerMonad E` is a monad as demonstrated by the `Ret` constructor and `bind` function. The `bind` function pattern matches its first argument `m` and, in the case of `Bind`, passes its second arguments `k` to the continuation of `m`. This "smart constructor" ensures that binds always associate to the right.

To embed $\mathcal{B}$, we model reading data from external devices using the effect type `DataEff`. This datatype includes only one (abstract) effect, called `GetData`. This constructor represents a data retrieval with the variable `v : var` that returns an unknown `bool`. Similar to how the `term` data type says nothing about the semantics of $\mathcal{B}$, the effect data type `DataEff` says nothing about the semantics of a data read. As a result, we say that the effects are embedded deeply in this style.

The embedding function appears on the right side of Fig. 2. The translation strategy is almost the same as embedding $\mathcal{B}$ using the reader monad. The only exception is in the variable case (the effectful part): here the `Bind` constructor marks the occurrence of the `GetData` effect.

In this mixed embedding, the pure parts of a $\mathcal{B}$ program have been translated to a shallow semantic domain, but the effectful parts remain abstract. It turns out that this separation is useful for both questions (1) and (2).

For question (1), we cannot answer it. This is desirable since we don't know if it's true without knowing more about the external device.

We can prove that property (2) is true even though the read effect is not interpreted—this is because the property follows from the monad laws (Fig. 4). However, we cannot prove property (3) because the commutativity law is not one of the monad laws.

Ideally, we would also like to state and prove property (4). However, the dynamic nature of freer monads forbids us from statically inspecting the syntactic structure of the program. Interpreting the embedding does not help us, either, since that would not preserve the original syntactic structure.

Our success with questions (1) and (2) suggests that we have found an useful intermediate layer between shallow and deep embeddings, but our failure in stating or proving properties (3) and (4) indicates that we haven't yet found the right representation.

2.4 Another Mixed Embedding Based on Reified Applicative Functors

The last embedding shown in the figure uses a type that reifies the interface of *applicative functors* (Fig. 3). As in freer monads, this datatype is parameterized by deeply embedded abstract effects. These effects, of type `E R`, are recorded by the `EmbedA` data constructor.

However, instead of constructors for `ret` and `bind`, this datatype includes constructors for `pure` and `liftA2`, the two operations that define applicative functors.[3] The `Pure` constructor shallowly "embeds" a pure computation into the domain, and `LiftA2` "connects" two computations that potentially contain effect invocations. These constructors provide a trivial implementation of the `Applicative` type class for this datatype.

The translation of $\mathcal{B}$ to this datatype uses a deep embedding of variable reads, using the `EmbedA` data constructor with the `DataEff` type from the previous embedding. Because, as in freer monads, this effect is modeled abstractly, we cannot prove or disprove (1).

The translation function uses the applicative interface in the datatype to translate the constants, unary and binary operators. These components are modeled shallowly (*i.e.,* as boolean constants and operators), but the program's syntactic structure is retained by the translation. However, because of the retainment, we need an additional equivalence relation to equate semantically equivalent terms that are not syntactically equal. We use an equivalence relation based on the applicative laws (appear shortly in the next section). These laws are sufficient to show that (2) holds.

On the other hand, we cannot prove (3) with the equivalence based solely on applicative laws. To model this sort of parallelism, we add a commutativity law to our equivalence relation which allows us to show (3). We defer the justification of adding this commutativity law to Section 3.4.

Finally, an advantage of this embedding is that it preserves enough of the syntax of the original program to prove (4). To do so, we must first calculate the depth of circuits and the number of variables under this encoding.

```
Fixpoint app_depth {E A} (t : ReifiedApp E A) : nat :=
  match t with
   | EmbedA _ => 0
   | Pure _ => 0
   | LiftA2 _ t u => 1 + max (app_depth t) (app_depth u)
   end.
```

We omit the function that counts the number of variables as it is similar to `app_depth`. Then we can formalize (4) in Coq as follow:

```
Theorem heightAndVar : forall (c : ReifiedApp DataEff bool),
    app_numVar c <= Nat.pow 2 (app_depth c).
```

The theorem is provable by an induction over `c`.

[3] Alternatively, `Applicative` can also be defined by `pure` and another operation `<*>` of type `F (A -> B) -> F A -> F B`, where `F` is an `Applicative` instance. These two definition are equivalent, as we can derive the definition of `<*>` from `liftA2` and vice versa.

2.5 Tlön embeddings

Just as the reader monad models *one* particular effect, freer monads model *one* particular computation pattern. Unfortunately, that particular computation pattern is not suitable for our $\mathcal{B}$ example, because it does not model parallel computation (*i.e.,* property (3)), nor does it capture the static data and control flows (*i.e.,* property (4)). Instead we saw that the mixed embedding in the previous subsection, based on reified applicative functors, is a better approach.

Can we generalize the key idea even further? If we go beyond $\mathcal{B}$, we might need to model other computation patterns. Are there other mixed embeddings that would be suitable for these tasks? How might we derive them?

To that end, we identify a novel set of mixed embeddings that we call *Tlön embeddings*. The goal of these embeddings is to provide flexibility in our models of effectful computation.[4] We define Tlön embeddings by identifying a set of *program adverbs* that specify the embedding type and equational theory used in the embedding. For example, the embedding in Section 2.4 is based on an adverb composed of the `ReifiedApp` type and an equational theory based on commutative applicative functors.

The flexibility that program adverbs provide can perhaps be understood by comparing them with effects: effects *do* certain actions, and program adverbs model *how* these actions are done—similar to the difference between verbs and adverbs. For example, the adverb we used in Section 2.4 is called "statically and in parallel", which states that there is a static dependency between different effect invocations and some of these effect invocations are executed in parallel.

In the next section, we define our set of program adverbs more precisely and discuss the reasoning principles that they provide for effectful computation.

3 PROGRAM ADVERBS

Program Adverbs are the building blocks of Tlön embeddings. Mathematically, they are composed of two parts: a syntactic part, called the adverb data type, and a semantic part, called the adverb theory. More formally, we define program adverbs as follow:

Definition 3.1 (Program Adverb). A program adverb is a product $(D, \cong)$. D is called the adverb data type and is parameterized by an effect type E and a return type R. The $\cong$ operation is called the adverb theory. It is a binary operation that defines a bisimulation relation on $D(E, R)$ for any E and R.

In Coq terms, an adverb data type has the type `(Type -> Type) -> Type -> Type`. The first parameter of `Type -> Type` is the effect E and it's parameterized by its own return type; the second parameter is the return type of R. The adverb theory $\cong$ is a typed binary relation:[5]

```
Definition Bisim {E : Type -> Type} {R : Type} : relation (D E R) := ...
Notation "a ≅ b" := (Bisim a b).
```

where `D` is the adverb data type and `relation` is defined as:

```
Definition relation (A : Type) := A -> A -> Prop.
```

[4]Here, we define effects as communications with external environment that are performed by some explicit operations. For example, *mutable states* are effects which can be explicitly incurred by operations such as `get` and `set`. For the same reason, we also consider I/O (with operations like `read`, `print`, *etc.*), exceptions (with operations like `throw`, *etc.*) as effects.

[5]In addition to bisimulation relations, we can also define refinement relations on program adverbs. We will show in Section 4.3 some adverbs with refinement relations, but bisimulation relations would suffice for most adverbs, so we only include them in the core definitions of adverb theories. Refinement relations can be added on demand.

```
(* Streamingly *)
Inductive ReifiedFunctor (E : Type -> Type) (R : Type) : Type :=
| EmbedF (e : E R)
| FMap {X : Type} (g : X -> R) (f : ReifiedFunctor E X).

(* Statically and StaticallyInParallel *)
Inductive ReifiedApp (E : Type -> Type) (R : Type) : Type :=
| EmbedA (e : E R)
| Pure (r : R)
| LiftA2 {X Y : Type} (f : X -> Y -> R)
        (a : ReifiedApp E X) (b : ReifiedApp E Y).

(* Conditionally *)
Inductive ReifiedSelective (E : Type -> Type) (R : Type) : Type :=
| EmbedS (e : E R)
| PureS (r : R)
| SelectBy {X Y : Type} (f : X -> ((Y -> R) + R))
          (a : ReifiedSelective E X) (b : ReifiedSelective E Y).

(* Dynamically *)
Inductive ReifiedMonad (E : Type -> Type) (R : Type) : Type :=
| EmbedM (e : E R)
| Ret (r : R)
| Bind {X : Type} (m : ReifiedMonad E X) (k : X -> ReifiedMonad E R).
```

Fig. 5. The adverb data types

This definition is overly general, so we focus our attention only on program adverbs that are *sound* according to the definition that we will develop below. Furthermore, in this paper we will only consider adverbs defined by reifying classes of functors.

3.1 Adverb Data Types and Theories

The four key adverb data types, shown in Fig. 5, are derived from the four type classes shown in Fig. 3. We have already seen one before in the applicative embedding in Fig. 2. Other definitions follow a similar pattern: the constructors of each data type include one for embedding effects (of type `E R`) and a constructor that reifies the interface of each method of the type class.

In addition to an adverb data type, every program adverb also comes with some theories, defined by a bisimulation relation $\cong$. The purpose of the $\cong$ relation is to equate all computations that are semantically equivalent regardless of what effects are present.

For example, an adverb called `Statically` is composed of the `ReifiedApp` datatype with an equational theory based on three sorts of rules: (1) a congruence rule with respect to `LiftA2`, (2) the laws of applicative functors,[6] and (3) the equivalence properties (*i.e.,* reflexivity, symmetry, transitivity). We show the concrete rules in Fig. 6.

Why do we call this adverb `Statically`? The data dependency in the `LiftA2` constructor of `ReifiedApp` shows that the data type imposes a "static" data flow and control flow on the computation: we will always need to run both parameters of type `ReifiedApp E A` and `ReifiedApp E B` to

[6] https://en.wikibooks.org/wiki/Haskell/Applicative_functors#Applicative_functor_laws

Congruence Rule

$$\textsc{Congruence} : \frac{\texttt{a1} \cong \texttt{a2} \qquad \texttt{b1} \cong \texttt{b2}}{\texttt{liftA2 f a1 b1} \cong \texttt{liftA2 f a2 b2}}$$

Applicative Functor Laws

$$\textsc{Left Identity} : \frac{\forall \texttt{y, (fun \_ x => x) a y = f a y}}{\texttt{liftA2 f (pure a) b} \cong \texttt{b}}$$

$$\textsc{Right Identity} : \frac{\forall \texttt{x, (fun x \_ => x) x b = f x b}}{\texttt{liftA2 f a (pure b)} \cong \texttt{a}}$$

$$\textsc{Associativity} : \frac{\forall \texttt{x y z, f x y z = g y z x}}{\texttt{liftA2 id (liftA2 f a b) c} \cong \texttt{liftA2 (flip id) a (liftA2 g b c)}}$$

$$\textsc{Naturality} : \frac{\forall \texttt{x y z, p (q x y) z = f x (g y z)}}{\texttt{liftA2 p (liftA2 q a b)} \cong \texttt{liftA2 f a . liftA2 g b}}$$

Equivalence Properties

$$\textsc{Reflexivity} : \frac{}{\texttt{a} \cong \texttt{a}} \qquad \textsc{Symmetry} : \frac{\texttt{a} \cong \texttt{b}}{\texttt{b} \cong \texttt{a}}$$

$$\textsc{Transitivity} : \frac{\texttt{a} \cong \texttt{b} \qquad \texttt{b} \cong \texttt{c}}{\texttt{a} \cong \texttt{c}}$$

Fig. 6. The equivalence relations for `ReifiedApp`.

get the result of type `ReifiedApp E C`, *i.e.,* we cannot skip either computation. In addition, neither of the two parameters depends on the result of the other, which allows us to statically inspect either of them without running the other.

Remark. The adverb data types and their associated theories form free structures similar to those in Capriotti and Kaposi [2014]; Kiselyov and Ishii [2015]; Mokhov [2019]; Mokhov et al. [2019]. However, one distinction is that we intentionally do not normalize the adverb data types to preserve syntactic structures. To distinguish un-normalized free structures and normalized free structures, we use the term *reified* structures to describe the former and the term free structures to exclusively describe the latter. We defer the detailed comparison and trade-offs between reified structures and free structures to Section 6.

3.2 Adverb Simulation

One important property of `ReifiedApp` is that it can be interpreted to any other instance of the `Applicative` class, as long as its embedded effects can be interpreted to that instance. We can show this via the abstract interpreter `interpA` shown in Fig. 7. The interpreter shows that given *any* effect `E` and *any* instance `I` of `Applicative`, as long as we can find an effect interpretation from `E A` to `I A` for any type `A`, we can interpret a `ReifiedApp E A` to an `I A` for any type `A`.

For example, we can interpret a `ReifiedApp DataEff` to the reader applicative functor (Fig. 2) [7] by supplying the following function to the parameter `interpE` of `interpA`:

[7] Every monad is also an applicative functor, so the reader monad is also a reader applicative functor.

```
Fixpoint interpA {E I : Type -> Type} `{Applicative I} {A : Type}
        (interpE : forall A, E A -> I A) (t : ReifiedApp E A) : I A :=
  match t with
  | EmbedA e => interpE _ e
  | Pure a => pure a
  | LiftA2 f a b => liftA2 f (interpA interpE a) (interpA interpE b)
  end.
```

Fig. 7. The interpretation from `ReifiedApp` to any instance of the `Applicative` type class.

```
Definition interpDataEff {A : Type} (e : DataEff A) : Reader A :=
  match e with GetData v => ask v end.
```

Similarly, we can interpret `ReifiedApp DataEff` to other semantic domains that are applicative functors.

Why do we care if `ReifiedApp` can be interpreted into any instance of `Applicative`? This is because different instances of `Applicative` model different effects—if we have a data structure that can be interpreted to all instances, we can develop a theory of it that can be used for reasoning about properties that are true regardless of what effects are present.

To make the relation between an adverb data type like `ReifiedApp` and a class of functor like `Applicative` more precise, we define the following *adverb simulation* relation:

Definition 3.2 (Adverb Simulation). Given an adverb data type D, a class of functor C, and a transformer T on all instances of C, we say that there is an adverb simulation from D to C under T, written $D \models_T C$, if we can define a function that, for any effect type E, instance F of type class C, and interpreter f from $E(A)$ to $F(A)$ for any type A, interprets a value of $D(E, A)$ to $T(F)(A)$ for any type A.

We add some flexibility to this definition by making it parameterize over a transformer T—we do not need this extra flexibility for now, but we will see why it is useful in Section 3.4.

We also define an *adverb interpretation* as follow:

Definition 3.3 (Adverb Interpretation). Given an adverb data type D, a class of functor C, and a transformer T on all instances of C, the interpreter I that shows $D \models_T C$ is called an adverb interpretation, and we write $I \in D \models_T C$.

Our `interpA` in Fig. 7 is an adverb interpretation. More specifically, we say that

$$\texttt{interpA} \in \texttt{ReifiedApp} \models_{\texttt{IdT}} \texttt{Applicative}.$$

where the `IdT` transformer is an identity `Applicative` transformer that "does nothing". In the rest of the paper, when we have $D \models_{\texttt{IdT}} C$ for any D and C, we abbreviate it as $D \models C$.

3.3 Sound Adverb Theories

To know that our adverb theory is *sound*, *i.e.*, it doesn't equate computations that are not semantically equivalent, we define the following soundness property of adverb theories:

Definition 3.4 (Soundness of Adverb Theories). Given a program adverb $(D, \cong)$ and an adverb interpretation $I \in D \models_T C$, we say that the adverb theory $\cong$ is sound with respect to I if there exist a lawful equivalence relation $\equiv$ such that for all $d_1, d_2 \in D$,

$$d_1 \cong d_2 \implies I(d_1) \equiv I(d_2).$$

Let us use `idT` for the transformer T for the moment. The equivalence relation $\equiv$ on C is lawful if they respect the congruence laws and the class laws of C. For `Applicative`, we use the common applicative functor laws regarding $\equiv$. Based on the soundness of adverb theories, we can define the following soundness property of program adverbs with respect to their adverb interpretations:

Definition 3.5 (Soundness of Program Adverbs). Given a program adverb $(D, \cong)$ and an adverb interpretation $I \in D \models_T C$, we say that the adverb is sound if the $\cong$ relation is sound with respect to I.

We can now prove that the `Statically` adverb is sound:

THEOREM 3.6. *The `Statically` adverb is sound with respect the adverb interpretation `interpA` $\in$ `ReifiedApp` $\models$ `Applicative`.*

PROOF. By induction over the $\cong$ relation. □

3.4 "Statically and in Parallel"

Two adverbs can use the same data type yet differ in their theories. Let's look at a variant of the `Statically` adverb called `StaticallyInParallel`. As its name suggests, it adds parallelization to a static computation pattern.

Recall that the two computations connected by `liftA2` do not depend on each other. This suggests that an implementation of `liftA2` can choose to run them in parallel. Indeed, that observation is one of the key ideas behind Haxl [Marlow et al. 2014].

Based on this idea, we also define the `StaticallyInParallel` adverb. The definitions of this adverb are mostly the same as the `Statically` adverb, except that it adds one additional rule to the adverb theory $\cong$:

```
liftA2 f a b ≅ liftA2 (flip f) b a
```

This rule, also known as the *commutativity* rule, states that the order that effects are invoked does not matter.

Note that compared with other rules, the commutativity rule is not satisfied by every applicative functor. This might suggest that we should not add it to the theory, as it might be a theory that only holds for certain effects. Nevertheless, we can prove the soundness of the adverb theory with respect to the following adverb simulation:

$$\texttt{ReifiedApp} \models_{\texttt{PowerSet}} \texttt{Applicative}$$

The `PowerSet` transformer is *a powerset applicative functor transformer* and its core definitions are shown in Fig. 8. The key of `PowerSet` is the `liftA2PowerSet` operation. When executed, it creates two nondeterministic branches (indicated by the disjunction `\/`): on one branch, it computes `a' : I A` before `b' : I B`, and vice versa on the other branch. Intuitively, this is to model the nondeterministic execution order in a parallel evaluation. Many of these operations depend on $\equiv$, which is the lawful $\equiv$ relation on `I`.

LEMMA 3.7. *If $\equiv$ is a lawful equivalence relation on `Applicative`, `EqPowerSet` is a lawful equivalence relation on `Applicative` that additionally satisfies the commutativity rule.*

PROOF. By definition. □

THEOREM 3.8. *The adverb is sound: `ReifiedApp` $\models_{\texttt{PowerSet}}$ `Applicative`.*

PROOF. We can construct an interpreter $P \in$ `ReifiedApp` $\models_{\texttt{PowerSet}}$ `Applicative` by modifying `interpA` (Fig. 7) so that it uses `embedPowerSet` on the `EmbedA` case, `purePowerSet` on the `Pure` case, and `liftA2PowerSet` on the `LiftA2` case. The rest follows from Lemma 3.7. □

```
Definition PowerSet (I : Type -> Type) (A : Type) := I A -> Prop.

Definition embedPowerSet {A : Type} (a : I A) : PowerSet I A := fun r => r ≡ a.

Definition purePowerSet {A : Type} (a : A) : PowerSet I A := fun r => r ≡ pure a.

Definition liftA2PowerSet {A B C} (f : A -> B -> C)
                     (a : PowerSet I A) (b : PowerSet I B) : PowerSet I C :=
  fun r => exists a', a a' /\ exists b', b b' /\
          (liftA2 f a' b' ≡ r \/ liftA2 (flip f) b' a' ≡ r).

Definition EqPowerSet {A} relation (PowerSet I A) :=
  fun p q => forall a, p a <-> q a.
```

Fig. 8. The core definitions of a powerset applicative functor transformer.

Intuitively, we can define `StaticallyInParallel` as an adverb because, even though with an effect running computations in different order might return different results, a language can be implemented in a parallel way such that the difference in evaluation orders is no longer observable.

3.5 Other Basic Adverbs

Besides `Statically` and `StaticallyInParallel`, we also identify three other basic adverbs, namely `Streamingly`, `Conditionally`, and `Dynamically`, defined using the adverb data types in Fig. 5.

`Streamingly`. This program adverb simulates `Functor` under `IdT`. The most simple form of stream processing computes the data directly as it is received. This is captured by the `fmap` interface (Fig. 3).

`Dynamically`. This adverb adverb simulates `Monad` (Fig. 3). A monad is the most expressive and dynamic among all four classes of functors thanks to its core operation `bind`. Any kind of computation can happen in the second operand and we can't know it without knowing a value of type `A`, which we can only get by running the first operand. This program adverb is commonly used in representing many programming language for its expressiveness, but it also allows for the least amount of static reasoning.

Unlike `Statically`, this variant does not have an `InParallel` variant. This might be surprising because there are many commutative monads. However, those monads are commutative because their specific effects are commutative. We cannot define a general powerset *monad transformer* that can make any monad satisfy the commutativity law.

`Conditionally`. We use this adverb to model conditional execution. The definition of its adverb data type is shown in Fig. 5. It reifies the `Selective` type class (Fig. 3). The signature operation of `Selective` is the `selectBy` operation. Loosely, "applying" a function of type `A -> ((B -> C) + C)` to a computation of type `F A` gets you either `F (B -> C)` or `F C`. In the first case, you will need to run the computation of type `F B`. You don't *need* to run the computation of type `F B` in the second case, but you can still choose to run it.

Because we can encode conditional execution with this adverb, it is more expressive than `Statically`. However, the extra expressiveness also makes static analysis less accurate. Since we cannot know statically if the computation `F B` in `selectBy` is executed, we can only get an under-approximation (assuming that `F B` is not executed) and an over-approximation (assuming that `F B` is executed) of the effects that would happen, but not an exact set.

Even though we derive this adverb by reifying `Selective`, we do not wish to model the adverb's theory using the laws of selective functors. This is because the laws of selective functors do not distinguish them from applicative functors. Indeed, every applicative functor is also a selective functor (by running the second argument even when not required) and vice versa, so adhering to the "default" laws do not allow us to prove more properties. Therefore, we add one simple rule to the selective functor laws:

```
select (inr <$> a) b ≅ a
```

This forces `select` to ignore the second argument when it does not need to be run. However, we can no longer show that `Conditionally` adverb simulates `Selective` by adding this laws, because ≅ is no longer an under-approximation of ≡. Instead, we show the following adverb simulation:

$$\texttt{ReifiedSelective} \models \texttt{Monad}$$

In this way, `Conditionally` serves as a compromise between `Statically` and `Dynamically`. Its adverb data type is more similar to `Statically` and allows for some static analysis, while its theories are more similar to `Dynamically`.

4 COMPOSABLE PROGRAM ADVERBS

From a monad instance, we can derive an applicative functor instance. From an applicative functor instance, we can derive a functor instance. We can derive a selective instance from an applicative functor and vice versa.[8] This subsumption hierarchy among classes of functors means that we can choose the most expressive abstract interface of a data type, and that choice automatically includes the less expressive interfaces.

However, although we can derive a "default" applicative functor from a monad, we don't always want to do that—*e.g.*, we may want to define a different behavior for `liftA2` than the one derived from `bind`. Indeed, Haxl is one such example, where `bind` is defined as a sequential operation and `liftA2` is parallel so that certain tasks with no data dependencies can be automatically parallelized [Marlow et al. 2014]. In the program adverbs terminology, the semantics of their language is composed of a "statically and in parallel" adverb and a "dynamically" adverb.

In addition, some languages may have a subset that corresponds to the "statically" adverb and some extensions that correspond to "dynamically". If we only use the "dynamically" adverb to reason about programs written in this language, we lose the ability to state properties for the "statically" subset.

We need a way to compose multiple program adverbs. Therefore, in this section, we refactor program adverbs to *composable program adverbs*. Composable program adverbs come with one operation ⊕, which joins adverbs as well as effects.

4.1 Uniform Treatment of Effects and Program Adverbs

Effects are commonly considered as secondary to monads. This treatment of effects carries over to the freer-monad based approaches and our previous implementation of program adverbs, where the effects are a parameter of adverb data types.

This approach works well when we use one fixed program adverb, but needs update when multiple adverbs are involved. This is because, in both scenarios we mentioned earlier, our intention is not to

[8] This is one special thing about selective functors: every selective functor is an applicative functor and the reverse is also true. However, separating these two classes is still useful because the automatically derived instances might not be what we want, as discussed in Mokhov et al. [2019].

```
(* Least fixpoint for program adverbs and effects. *)
Definition Alg1 (F : (Set -> Set) -> Set -> Set) (E : Set -> Set) : Type :=
  forall {A : Set}, F E A -> E A.
Definition Fix1 (F : (Set -> Set) -> Set -> Set) (A : Set) :=
  forall (E : Set -> Set), Alg1 F E -> E A.

(* Least fixpoint for equivalence relations of program adverbs and effects. *)
Definition AlgRel {F : Set -> Set}
           (R : (forall (A : Set), relation (F A)) -> forall (A : Set), relation (F A))
           (K : forall (A : Set), relation (F A)) : forall (A : Set), relation (F A) :=
  fun A (a b : F A) => R K _ a b -> K _ a b.

Definition FixRel {F : Set -> Set}
           (R : (forall (A : Set), relation (F A)) -> forall (A : Set), relation (F A))
  : forall (A : Set), relation (F A) :=
  fun A (a b : F A) => forall (K : forall (A : Set), relation (F A)),
      (forall (A : Set) (a b : F A), AlgRel R K _ a b) -> K _ a b.
```

Fig. 9. The algebra and the least fixpoint operators for effects and adverb data types (`Alg1`, `Fix1`), and for adverb theories (`AlgRel, FixRel`).

combine program adverbs that each contain their own set of effects—we would like the composed program adverbs to share the same set of effects. One solution is requiring that we can only join program adverbs when they share the same set of effects, but that would requires extra machinery.

In our work, we choose to give a uniform treatment to effects and program adverbs. On the type level, in our first definition, adverb data types have type `(Type -> Type) -> Type -> Type`, where the first parameter is an effect. For the composable version, we modify the first parameter so that it can be either an effect or an adverb. In our first definition, effects have type `Type -> Type`, we modify them to have the same type as adverbs. Both effects and program adverbs can be recursive, which means their first parameter can be a union that includes themselves (we will see how to implement this in Coq in Section 4.2). We also define the $\oplus$ operation so that we can apply either effects or program adverbs (or both effects and program adverbs joined by $\oplus$) on both sides of the operation.

4.2 The Coq Implementation

All the program adverbs we have seen are recursive. When we compose these program adverbs, we cannot simply put them into a sum type—we need to adapt each adverb so that it recurses on the new composed adverb rather than itself. In other words, we need *extensible inductive types*. However, extensible inductive types are not directly supported by most formal reasoning systems including Coq. In fact, how to support extensible inductive types is an open problem known as *the expression problem* [Wadler 1998].

In this paper, we address the problem and implement composable adverbs in Coq using a technique presented in *Meta Theory à la Carte* (MTC) [Delaware et al. 2013]. The key idea of MTC is using Church encodings of data types [d. S. Oliveira 2009; Wadler 1990] instead of Coq's native inductive types. We apply and extend this idea to define two least fixpoint operators `Fix1` and `FixRel` that work on adverb data types and adverb theories, respectively. We show the definitions of these operators in Fig. 9.

```
Variant ReifiedPure (K : Set -> Set) (R : Set) : Set :=
| Pure (r : R).

Variant ReifiedFunctor (K : Set -> Set) (R : Set) : Set :=
| FMap {X : Set} (g : X -> R) (f : K X).

Variant ReifiedApp (K : Set -> Set) (R : Set) : Set :=
| LiftA2 {X Y : Set} (f : X -> Y -> R)(g : K X) (a : K Y).

Variant ReifiedSelective (K : Set -> Set) (R : Set) : Set :=
| SelectBy {X Y : Set} (f : X -> ((Y -> R) + R)) (a : K X) (b : K Y).

Variant ReifiedMonad (K : Set -> Set) (R : Set) : Set :=
| Bind {X : Set} (m : K X) (g : X -> K R).
```

Fig. 10. The composable adverb data types.

Figure 10 shows the definitions of composable adverb data types. Compared with the adverb data types in Fig. 5, a composable adverb data type replaces the effect parameter (which is named as `E`) with a recursive parameter (which is named as `K`) so that it "recurses" on `K` instead of itself.

We also factor out the `Pure` constructor, a common part shared by multiple basic adverb data types, as a separate composable adverb data type called `ReifiedPure`. In this way, we avoid introducing multiple `Pure` constructors, *e.g.*, by combining `Statically` and `Conditionally`. Furthermore, we remove the `Embed` constructors in composable adverb data types. Thanks to the uniform treatment of effects and program adverbs, we can now embed effects simply by including them in `K`, so we have no need for those constructors.

As an example, we can define an inductive type `T : Set -> Set` that is composed of `ReifiedPure`, `ReifiedApp`, and some effect `E` as follow:

```
Definition T := Fix1 (ReifiedPure ⊕ ReifiedApp ⊕ E).
```

We define all composable adverb data types using `Set` rather than `Type` because we use the impredicative sets extension in Coq, following MTC. The consequence of this decision is that (1) certain types cannot inhabit in `Set`, and (2) the extension is inconsistent with certain axioms like classical axioms.[9] We also develop other mechanisms like the injection type classes, the induction principles following MTC. We omit more detail here due to the space constraint. The interested readers can find them in MTC [Delaware et al. 2013].

Besides MTC, there are other solutions [Forster and Stark 2020; Kravchuk-Kirilyuk et al. 2021] that address the expression problem in theorem provers like Coq. We discuss those alternative solutions in Section 6.

4.3 Add-on Adverbs

Another benefit of making program adverbs composable is that we can now define two add-on adverbs, namely `Repeatedly` and `Nondeterministically`, which are not suitable as standalone adverbs. These two adverbs reify two classes of functors, namely `AppKleenePlus` and `FunctorPlus`, that we define ourselves. We show these classes of functors and their reifications in Fig. 11. `AppKleenePlus` is

[9] https://github.com/coq/coq/wiki/Impredicative-Set

```
Class AppKleenePlus (F : Type -> Type) `{Applicative F} :=
  { kplus {A} : F A -> F A }.

Class FunctorPlus (F : Type -> Type) `{Functor F} :=
  { plus {A} : F A -> F A -> F A }.

(* The adverb data type for Repeatedly. *)
Variant ReifiedKleenePlus (K : Set -> Set) (R : Set) : Set :=
| KPlus : K R -> ReifiedKleenePlus K R.

(* The adverb data type for Nondeterministically. *)
Variant ReifiedPlus (K : Set -> Set) (R : Set) : Set :=
| Plus : K R -> K R -> ReifiedPlus K R.
```

Fig. 11. The adverb data types of `Nondeterministically` and `Repeatedly`.

$$
\begin{array}{rcl}
\textsc{Repeat} & : & \forall n,\ \texttt{repeat a n} \subseteq \texttt{kplus a} \\
\textsc{Kplus} & : & \dfrac{\texttt{a} \subseteq \texttt{kplus b}}{\texttt{kplus a} \subseteq \texttt{kplus b}} \\
\textsc{Commutativity} & : & \texttt{plus a b} \cong \texttt{plus b a} \\
\textsc{Associativity} & : & \texttt{plus a (plus b c)} \cong \texttt{plus (plus a b) c} \\
\textsc{Plus} & : & \dfrac{\texttt{a} \subseteq \texttt{c} \qquad \texttt{b} \subseteq \texttt{c}}{\texttt{plus a b} \subseteq \texttt{c}} \\
\textsc{Left Plus} & : & \texttt{a} \subseteq \texttt{plus a b} \\
\textsc{Right Plus} & : & \texttt{b} \subseteq \texttt{plus a b}
\end{array}
$$

Fig. 12. The adverb theories for `Repeatedly` and `Nondeterministically`. The function `repeat a n` repeats `a` for `n` times. Functions `kplus` and `plus` are smart constructors of `KPlus` and `Plus`, respectively.

a subclass of `Applicative` and represents the Kleene plus.[10] It is a Kleene plus rather than a Kleene star because no empty element is defined. `FunctorPlus` is similar to the commonly-used `Alternative` and `MonadPlus` type classes in Haskell, but contains no empty element and only requires itself to be a subclass of `Functor`. We define these type classes' reifications as add-on adverbs so that these adverbs can be composed with classes of functors at different expressive levels: *e.g.*, `Repeatedly` can be composed with `Statically` as well as `Dynamically`.

We show the the adverb theories of `Repeatedly` and `Nondeterministically` in Fig. 12. Both of these two add-on adverbs are somewhat nondeterministic, so one change we make to their adverb theories is adding refinement relations ($\subseteq$) in addition to bisimulation relations ($\cong$).

We show that these two adverbs are sound with respect to the following adverb simulations:

$$\texttt{ReifiedKleenePlus} \models_{\texttt{PowerSet}} \texttt{AppKleenePlus}$$
$$\texttt{ReifiedPlus} \models_{\texttt{PowerSet}} \texttt{FunctorPlus}$$

[10] https://en.wikipedia.org/wiki/Kleene_star#Kleene_plus

```
(* FunctorPlus transformer. *)
Definition fmapPowerSet {A B : Type} (f : A -> B) (a : PowerSet I A) : PowerSet I B :=
  fun r => exists a', a a' /\ fmap f a' ≡ r.

Definition plusPowerSet {A : Type} (a b : PowerSet I A) : PowerSet I A :=
  fun r => a r \/ b r.

(* AppKleenePlus transformer. *)
Definition liftA2PowerSet {A B C : Type} (f : A -> B -> C)
           (a : PowerSet I A) (b : PowerSet I B) : PowerSet I C :=
  fun r => exists a' b', a a' /\ b b' /\ (liftA2 f a' b' ≡ r).

Definition seqPowerSet {A B : Type}
           (a : PowerSet I A) (b : PowerSet I B) : PowerSet I B :=
  fun r => exists a' b', a a' /\ b b' /\ (a' *> b' ≡ r).

Fixpoint repeatPowerSet {A : Type} (a : PowerSet I A) (n : nat) : PowerSet I A :=
  match n with
  | 0 => a
  | S n => seqPowerSet a (repeatPowerSet a n)
  end.

Definition kplusPowerSet {A : Type} (a : PowerSet I A) : PowerSet I A :=
  fun r => exists n, repeatPowerSet a n r.
```

Fig. 13. The `FunctorPlus` transformer instance and the `AppKleenePlus` transformer instance of the `PowerSet` data type. ≡ is the lawful equivalence relation on original functor/applicative functor `I`. The infix operator `*>` is the sequencing operation on `Applicative` that discards the value of the first argument.

The definition of `PowerSet` data type is the same as that in Fig. 8, but we are using its `AppKleenePlus` transformer and `FunctorPlus` transformer instances here. The core definitions of these transformers are shown in Fig. 13.

5 EXAMPLES

In this section, we demonstrate using program adverbs and Tlön embeddings to formally reason about programs with effects via two examples. The examples and the properties we focus on in these examples are intentionally different to show the usefulness of program adverbs and Tlön embeddings in different scenarios. The first example is a Haskell library that automatically parallelizes operations. We show that we can use composable program adverbs to capture the two different computation patterns in the same library. The second example is a networked server adapted from Koh et al. [2019]. We show that Tlön embeddings are useful as intermediate layers in a layered verification approach.

5.1 Haxl

Haxl is a Haskell library developed and maintained by Meta (formerly known as Facebook) that automatically parallelizes certain operations to achieve better performance [Marlow et al. 2014]. As an example, suppose that we want to fetch data from a database and we have a `Fetch : Type -> Type` data type that encapsulates the fetching effect. The key insight of the Haxl library is to distinguish the operations of `Fetch`'s `Monad` instance and those of its `Applicative` instance. When we use >>= to bind two `Fetch`s, those data fetches are sequential; but when use `liftA2` to bind two them, those data fetches are batched and will be sent to the database together. Furthermore, when writing the program in Haskell using its `do` notation, the `applicative-do` language extension automatically infers the use of `Applicative` operations when possible [Marlow et al. 2016]. In this way, we can write a program imperatively using `do` notation and Haskell automatically batches some of those operations to reduce the number of database accesses, hence improving the overall performance.

This design of Haxl poses a challenge to mixed embeddings based on freer monads or any other variant of a single basic adverb, because we need to distinguish when `Applicative` operations are used and when `Monad` operations are used. Fortunately, we can handle this distinction with composable adverbs.

In this example, we assume that we already have a translation from Haxl's `Applicative` and `Monad` operations to those operations in Coq.[11] In our embedding, we use the following composition of adverbs and effects to model a data fetching program in Haxl (recall the definitions of these adverbs in Fig. 10.):

```
ReifiedPure ⊕ ReifiedApp ⊕ ReifiedMonad ⊕ DataEff
```

We use `ReifiedApp` to model the batched operations and the theory of `StaticallyInParallel` to model their parallel nature. We use `ReifiedMonad` to model the sequential operations. When using this composition, we need to aware that both `ReifiedApp` and `ReifiedMonad` are instances of `Applicative` so we need to use the right instance when calling `Applicative` operations. In Coq, we can ensure this by either explicitly provide the correct instance or assigning a priority to each instance.[12] In our Coq development, we take the second approach and assign a lower priority to `ReifiedMonad`'s instance.

We cannot know statically how many database accesses would happen in a Haxl program, because a program can choose to do different things depending on the result of some data fetch. Therefore, we need to pick an effect interpretation for `DataEff` to reason about this property. In this example, we are assuming that the database does not change, so we interpret our embeddings of a Haxl program to the `DB` monad shown in Fig. 14.

The `DB` monad is essentially a combination of a reader monad and a writer monad.[13] The "reader state" has type `var -> val` which represents an immutable key-value database we can read from. The "writer state" is a `nat`, which represents the accumulated number of database accesses. The `bind` operation propagates the key-value database and accumulates the cost. The `liftA2` operation, on the other hand, only records the maximum number of database accesses in one of its branches.

Note that we are fine with using the `DB` monad here because it satisfies the commutativity rule. We should apply the `PowerSet` applicative transformer if that is not the case.

5.2 A Networked Server

A common technique used in formal verification is dividing the verification into multiple layers and establishing a refinement relation between each two layers [Gu et al. 2015; Koh et al. 2019; Lorch et al. 2020; Zakowski et al. 2021]. This approach offers better abstraction and modularity, as at each layer, we only need to consider certain subsets of properties.

[11] Tools like `hs-to-coq` [Spector-Zabusky et al. 2018] can be adapted to implement the translation.
[12] https://coq.inria.fr/refman/addendum/type-classes.html
[13] The `DB` monad is *not* a state monad, because the combined reader and writer monads are instantiated with different types of states.

```
Definition DB A :=  ((var -> val) -> A * nat).

Definition ret {A} (a : A) : DB A := fun map => (a, 0).

Definition bind {A B} (m : DB A) (k : A -> DB B) : DB B :=
  fun map =>
    match m map with
    | (i, n) => match (k i map) with
                | (r, n') => (r, n + n')
                end
    end.

Definition liftA2 {A B C} (f : A -> B -> C) (a : DB A) (b : DB B) : DB C :=
  fun map =>
    match (a map, b map) with
    | ((a, n1), (b, n2)) => (f a b, max n1 n2)
    end.

Definition get (v : var) : DB val := fun map => (map v, 1).
```

Fig. 14. The DB monad.

In this example, we apply program adverbs and Tlön embeddings to a networked server adapted from that of Koh et al. [2019]. While an end-to-end proof of the server's correctness is beyond the scope of this work, we show that Tlön embeddings are useful as some of those intermediate layers.

The server communicates with multiple clients via a network interface. Whenever the server receives a request, it stores the number in the request and send back a number in its store—a client does not necessarily receives what they send, because the server can interleave multiple sessions.

The implementation. Similar to the server of Koh et al. [2019], the server is implemented using a single-process *event loop* [Pai et al. 1999]. Instead of processing a request and sending back a response immediately, the server divides the processing into multiple steps. In each iteration of the event loop, the server advances the processing of each request by one step, thus interleaves different sessions.

We show the main loop body of our adapted version of the networked server in Fig. 15a. For simplicity, we use a custom language called NetImp that is adapted from the Imp language [Pierce et al. 2021]. The NetImp language supports datatypes like booleans, natural numbers, and a special record type called `connection`. It has network operations like `accept`, `read`, and `write`. All these operations return natural numbers, with 0 indicating failures. The language does not have a while loop but it has a `FOR` loop that iterates over a list. The loop variable is implemented as a pointer that points to the elements in the list in iterations. We also use C-like notations (*i.e.,* `*` and `->`) for operations on pointers.

The loop body maintains a lists of connections called `conns`. Each connection in the `conns` list has a state in one of three values: `READING`, `WRITING`, or `CLOSED`. At the start of each loop, the server checks if there is a new connection waiting to be established by calling the non-blocking operation `accept`. If there is, the server adds it to `conns`. The server then goes over each connection in `conns`: if the connection is in the `READING` state, the server tries to read from the connection and updates

```
1  newconn ::<- accept ;;
2  IF (not (*newconn == 0)) THEN
3    newconn_rec ::= connection *newconn
4                                WRITING ;;
5    conns ::++ newconn_rec
6  END ;;
7  FOR y IN conns DO
8    IF (y->state == WRITING) THEN
9      r ::<- write y->id *s ;;
10     IF (*r == 0) THEN
11       y->state ::= CLOSED
12     END
13   END ;;
14   IF (y->state == READING) THEN
15     r ::<- read y->id ;;
16     IF (*r == 0) THEN
17       y->state ::= CLOSED
18     ELSE
19       s ::= *r ;;
20       y->state ::= WRITING
21     END
22   END
23 END.
```

(a) The implementation.

```
Some
  (Or (newconn ::<- accept ;;
       IF (not (*newconn == 0)) THEN
         newconn_rec ::= connection *newconn
                                     WRITING ;;
         conns ::++ newconn_rec
       END)
       (OneOf (conns) y
         (Or (IF (y->state == WRITING) THEN
                r ::<- write y->id *s ;;
                IF (*r == 0) THEN
                  y->state ::= CLOSED
                END
             END)
            (IF (y->state == READING) THEN
                r ::<- read y->id ;;
                IF (*r == 0) THEN
                  y->state ::= CLOSED
                ELSE
                  s ::= *r ;;
                  y->state ::= WRITING
                END
             END))))
```

(b) Our intermediate layer specification.

Fig. 15. The implementation and the intermediate layer specification of our networked server.

an internal state `s` with the recently read value; if the connection is in the `WRITING` state, the server sends the current value of its internal state `s` to the connection; once a connection enters the `CLOSED` state, it remains that state forever and the server will not do anything with it—we design the server in this way for simplicity; a more realistic server should remove the connection from the list.

The specification. In general, we would like our specifications to omit implementation details—but we can do this slowly, one step a time. For example, Koh et al. first show that their implementation refines an *implementation model*, which is a specification that still involves low-level language mechanisms like the network interface and the `connection` data type, but blurs the *control flow*. After that, they show that the implementation model refines a higher-level specification that describes observation over the network. Here, we show a refinement that is similar to the former. Furthermore, we show that we can model this refinement as a refinement *over adverbs*.

We show our specification in Fig. 15b. The specification is written in a language similar to NetImp but with a few additional commands: `Some` is an unary operation that models the Kleene plus; `Or` is a binary operation that models a nondeterministic choice; `OneOf` is also a nondeterministic choice, but it does so by choosing from a list—line 8 means that we nondeterministically assign the variable `y` with one element from the list in `conns`.

Compared with the implementation, the specification is at the same level in terms of language mechanisms but is more nondeterministic. At each iteration of the main event loop, the implementation always first tries to `accept` a connection. After that, it goes over the list of `conns` in a fixed order. The specification does not enforce order: an `accept` could happen immediately after another `accept`; we can access elements in `conns` in any order and some connection might get visited more often than others.

What is the point of this specification? Suppose that one day we decide to change to a different implementation that switches the program fragment in lines 8-13 with that in lines 14-22 (and add an `ELSE` after the new first `IF` statement), the new implementation should refine the same specification. In that case, we only need to re-establish the refinement between the new implementation with the same specification, while other reasoning that we have done on the specification remains intact.

Tlön embeddings and the refinement proof. To show that our implementation refines our specification, we embed both NetImp and the specification language in Coq using program adverbs. We use the following composition:

```
ReifiedKleenePlus ⊕ ReifiedPlus ⊕ ReifiedPure ⊕ ReifiedMonad ⊕
  NetworkEff ⊕ MemoryEff ⊕ FailEff
```

We have already seen the first four adverbs. `NetworkEff` models the effects incurred by network operations `accept`, `read`, and `write`. `MemoryEff` models the effects incurred by assigning values to variables and retrieving values from them. `FailEff` models when the program crashes. We omit our translation due to space constraints.

Both the implementation and specification result in large embedded expressions, which poses a challenge in proving the refinement. However, we can observe that these two programs share some common program fragments (*e.g.,* lines 1–6 of the implementation are the same as lines 2–7 of the specification). Indeed, there are three such common fragments.

Our proof works by further recognizing four intermediate layers between the implementation and the specification. At the first layer, we reorganize the implementation into a program composed of three abstract fragments. At the second layer, we replace the for loop with the nondeterministic choice `OneOf`. At the third layer, we replace the sequencing on line 13 of the implementation with the `Or` in line 9 of the specification. At the fourth layer, we replace the sequencing on line 6 the implementation with the `Or` on line 2 of the specification. Adverb theories suffice for proving that each layer refines a higher layer. We obtain a proof that shows our implementation refines the intermediate layer specification by connecting all these refinement proofs together.

6 DISCUSSION

The expression problem. The composable program adverbs require extensible inductive types. We implement this feature in Coq by using the Church encodings of data types, following the precedent work of MTC [Delaware et al. 2013]. There are several consequences of using Church encodings instead of Coq's original inductive data types.

First, we cannot make use of Coq's language mechanisms, libraries, and plugins that make use of Coq's inductive types (*e.g.,* Coq's builtin induction principle generator, the Equations library [Sozeau and Mangin 2019], the QuickChick plugin [Lampropoulos and Pierce 2021], *etc.*). Furthermore, the extra implementation overheads incurred by Church encodings (*e.g.,* proving an algebra is a functor, proving the induction principle using dependent types, *etc.*) can be huge. However, this situation can be helped by developing tools or plugins for supporting Church encodings.

The other consequence is that, following the practice of MTC, we use Coq's impredicative set extension. This causes (1) certain types cannot inhabit in `Set`, and (2) our Coq development to be inconsistent with certain axioms like classical axioms, as we have discussed in Section 4.2.

There are alternative methods for addressing the expression problem. One option is the meta-programming approach proposed by Forster and Stark [2020]. Using this approach, we can define each composable adverb separately in a meta language and use a language plugin to generate a combined definition in Coq. This approach does not fully address the expression problem as extending the combined definition requires recompilation—but the amount of code that needs to be recompiled is much smaller. Another option is adding *family polymorphism* to theorem provers, which has recently been explored by Kravchuk-Kirilyuk et al. [2021]. Even though these works are promising, they either lack mature tool support or is still in development at the moment, so we are not using these approaches in our current development.

Reified vs. free structures. Even though the reified structures used in adverb data types are free structures, they are different from those free structures present in Capriotti and Kaposi [2014]; Kiselyov and Ishii [2015]; Mokhov [2019]; Mokhov et al. [2019]. The biggest difference between reified structures and these free structures are the parameters they recurse on: all the reified structures recurse on both their computational parameters, while each free structure only recurses on one of them.[14] Therefore, a free structure does not just reify a class of functors, it also converts the reification to a left- or right-associative normal form.

One advantage of the normal forms in free structure definitions is that the type class laws can be automatically derived from definitional equality (with the help of the axiom of functional extensionality). However, this conversion would eliminate some differences in the syntax. Taking `ReifiedApp` as an example, normalizing it would result in a “list” rather than a “binary tree”, making analyzing the depth of the tree impossible. Preserving the original tree structure of `StaticallyInParallel` also plays a crucial role in our examples shown in Section 2.4 and 5.1.

7 RELATED WORK

Semantic embeddings. There are various works that study different semantic embeddings. Boulton et al. [1992] are the pioneers who coined terms such as semantic embeddings, shallow embeddings, and deep embeddings. It is known that there are many styles of embeddings between shallow and deep embeddings,[15] but the term mixed embeddings was not seen before Chlipala [2021], which proposes a mixed embedding based on freer monads.

Freer Monads and Variants. Freer monads [Kiselyov and Ishii 2015] and their variants are studied by many researchers in formal verification to reason about programs with effects. For example, Letan et al. [2021] use freer monads to develop a modular verification framework based on effects and effect handlers called FreeSpec. Christiansen et al. [2019] develop a framework based on free monads and containers for reasoning about Haskell programs with effects. Swierstra and Baanen [2019] interpret freer monads into a predicate transformer semantics that is similar to Dijkstra monads; Nigron and Dagand [2021] interprets freer monads using separation logic.

On the *coinductive* side, Xia et al. [2020] develop a coinductive variant of freer monads called *the interaction trees* that can be used to reason about general recursions and nonterminating programs. Koh et al. [2019] encode interaction trees in VST [Appel 2014] to reason about networked servers. Mansky et al. [2020] use interaction trees as a lingua franca to interface and compose higher-order separation logic in VST and a first-order verified operating system called CertiKOS [Gu et al. 2015]. Zakowski et al. [2020] propose a technique called generalized parameterized coinduction for developing equational theory for reasoning about interaction trees. Zakowski et al. [2021] use interaction trees to define a modular, compositional, and executable semantics for LLVM. Silver and Zdancewic [2021] connect interaction trees with Dijkstra monads [Maillard et al. 2019] for writing termination sensitive specifications based on uninterpreted effects.

[14] With the exception of reified/free functors, since each of them has only one computational parameters to be recursed on.
[15] https://twitter.com/AndrewDGordon/status/1430262928096350209

Among many variants of freer monads, one particular structure closely resembles program adverbs. That is the action trees defined in Swamy et al. [2020]. The action trees have four constructors, `Act`, `Ret`, `Par`, and `Bind`, whose types correspond to effects, `ReifiedPure`, `ReifiedApp`, and `ReifiedMonad` in composable program adverbs, respectively, another evidence that program adverbs are general models. In contrast to our work, compositionality and extensibility of "adverbs" are not the main issue action trees try to address, so action trees are not built in a composable way. On the other hand, action trees are embedded with separation logic assertions, which are not the focus of Tlön embeddings or program adverbs.

Other Free Structures. Other free structures are also explored by various works. Capriotti and Kaposi [2014] propose two variants of freer applicative functors, which correspond to the left- and right-associative variants, respectively. Xia [2019] explores defining freer applicative functors in Coq, and points out that the right associative variant is harder to define in Coq. Milewski [2018] discusses how to derive freer monoidal functors.[16] Mokhov [2019] defines the freer selective applicative functors.

Programming Abstractions. We are not the first to observe that monads are too dynamic for certain applications. For example, Swierstra and Duponcheel [1996] identify that a parser that has some static features cannot be defined as a monad. Inspired by their observation, Hughes [2000] proposes a new abstract interface called arrows. The relationship among arrows, applicative functors, monads are studied by Lindley et al. [2011]. Willis et al. [2020] observe that monads generate dynamic structures that are hard to optimize. They further show that, by using applicative and selective functors instead, it is possible to implement staged parser combinators that generate efficient parsers. Mokhov et al. [2020] observe that the data type of tasks in a build system (called `Task` in their paper) can be parameterized by a class constraint to describe various kinds of build tasks. For example, a `Task Applicative` describes tasks whose dependencies are determined *statically* without running the task; and a `Task Monad` describes tasks with *dynamic* dependencies.

8 CONCLUSION

In this paper, we compare different styles of semantic embeddings and how they impact formal reasoning about programs with effects. We find that, if used properly, mixed embeddings can combine benefits of both shallow and deep embeddings, and be effective in (1) preserving syntactic structures of original programs, (2) showing general properties that can be proved without assumptions over external environment, and (3) reasoning about properties in specialized semantic domains.

We propose *program adverbs* and *Tlön embeddings*, a class of structures and a style of mixed embeddings based on these structures, that enable us to reap these benefits. Like free monads, program adverbs embed pure computations shallowly and effects deeply (and abstractly, but can later be interpreted). However, various program adverbs correspond to alternative computation patterns, and can be composed to model programs with multiple characteristics.

Based on program adverbs, Tlön embeddings cover a wide range of programs and allow us to reason about syntactic properties, semantic properties, and general semantic properties with no assumption over external environment within the same embedding.

REFERENCES

Andrew W. Appel. 2014. *Program Logics - for Certified Compilers.* Cambridge University Press. http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB

[16]Monoidal functors are equivalent to applicative functors, so they also correspond to the `Statically` adverb.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*. 50–65. https://doi.org/10.1007/11541868_4

Jorge Luis Borges. 1940. Tlön, Uqbar, Orbis Tertius. In *Labyrinths*, Donald A. Yates and James E. Irby (Eds.). New Directions.

Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. 1992. Experience with Embedding Hardware Description Languages in HOL. In *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings (IFIP Transactions, Vol. A-10)*, Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute (Eds.). North-Holland, 129–156.

Paolo Capriotti and Ambrus Kaposi. 2014. Free Applicative Functors. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Levy and Neel Krishnaswami (Eds.). 2–30. https://doi.org/10.4204/EPTCS.153.2

Adam Chlipala. 2021. Skipping the binder bureaucracy with mixed embeddings in a semantics course (functional pearl). *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–28. https://doi.org/10.1145/3473599

Jan Christiansen, Sandra Dylus, and Niels Bunkenburg. 2019. Verifying effectful Haskell programs in Coq. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, Richard A. Eisenberg (Ed.). ACM, 125–138. https://doi.org/10.1145/3331545.3342592

Bruno C. d. S. Oliveira. 2009. Modular Visitor Components. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, 269–293. https://doi.org/10.1007/978-3-642-03013-0_13

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 207–218. https://doi.org/10.1145/2429069.2429094

Sandra Dylus, Jan Christiansen, and Finn Teegen. 2019. One Monad to Prove Them All. *Art Sci. Eng. Program.* 3, 3 (2019), 8. https://doi.org/10.22152/programming-journal.org/2019/3/8

Yannick Forster and Kathrin Stark. 2020. Coq à la carte: a practical approach to modular syntax with binders. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 186–200. https://doi.org/10.1145/3372885.3373817

Simon Foster, Chung-Kil Hur, and Jim Woodcock. 2021. Formally Verified Simulations of State-Rich Processes using Interaction Trees in Isabelle/HOL. *CoRR* abs/2105.05133 (2021). arXiv:2105.05133 https://arxiv.org/abs/2105.05133

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 595–608. https://doi.org/10.1145/2676726.2676975

John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.* 37, 1-3 (2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. 94–105. https://doi.org/10.1145/2804302.2804319

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 234–248. https://doi.org/10.1145/3293880.3294106

Anastasiya Kravchuk-Kirilyuk, Yizhou Zhang, and Nada Amin. 2021. Family Polymorphism for Proof Extensibility. In *Proceedings of the 27th International Conference on Types for Proofs and Programs, TYPES 2021, June 14–18, 2021*. https://types21.liacs.nl/download/family-polymorphism-for-proof-extensibility/

Leonidas Lampropoulos and Benjamin C. Pierce. 2021. *QuickChick: Property-Based Testing in Coq*. Electronic textbook. Version 1.2. https://softwarefoundations.cis.upenn.edu/qc-1.2/.

Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. 2021. Modular verification of programs with effects and effects handlers. *Formal Aspects Comput.* 33, 1 (2021), 127–150. https://doi.org/10.1007/s00165-020-00523-2

Sam Lindley, Philip Wadler, and Jeremy Yallop. 2011. Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous. *Electron. Notes Theor. Comput. Sci.* 229, 5 (2011), 97–117. https://doi.org/10.1016/j.entcs.2011.02.018

Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. 2020. Armada: low-effort verification of high-performance concurrent programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 197–210. https://doi.org/10.1145/3385412.3385971

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra monads for all. *Proc. ACM Program. Lang.* 3, ICFP (2019), 104:1–104:29. https://doi.org/10.1145/3341708

William Mansky, Wolf Honoré, and Andrew W. Appel. 2020. Connecting Higher-Order Separation Logic to a First-Order Outside World. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 428–455. https://doi.org/10.1007/978-3-030-44914-8_16

Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: an abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 325–337. https://doi.org/10.1145/2628136.2628144

Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, Geoffrey Mainland (Ed.). ACM, 92–104. https://doi.org/10.1145/2976002.2976007

Coq development team. 2021. *The Coq proof assistant.* http://coq.inria.fr Version 8.13.2.

Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings.* 257–275. https://doi.org/10.1007/978-3-319-19797-5_13

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

Bartosz Milewski. 2018. Free Monoidal Functors, Categorically! https://bartoszmilewski.com/2018/05/16/free-monoidal-functors-categorically/ Blog post.

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Andrey Mokhov. 2019. Implementation of selective applicative functors in Haskell. https://hackage.haskell.org/package/selective

Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jerémie Dimino. 2019. Selective applicative functors. *Proc. ACM Program. Lang.* 3, ICFP (2019), 90:1–90:29. https://doi.org/10.1145/3341694

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2020. Build systems à la carte: Theory and practice. *J. Funct. Program.* 30 (2020), e11. https://doi.org/10.1017/S0956796820000088

Pierre Nigron and Pierre-Évariste Dagand. 2021. Reaching for the Star: Tale of a Monad in Coq. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:19. https://doi.org/10.4230/LIPIcs.ITP.2021.29

Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. 1999. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Annual Technical Conference, June 6-11, 1999, Monterey, California, USA*. USENIX, 199–212. http://www.usenix.org/events/usenix99/full_papers/pai/pai.pdf

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2021. *Logical Foundations.* Electronic textbook. Version 6.1. http://www.cis.upenn.edu/~bcpierce/sf.

Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434307

Matthieu Sozeau and Cyprien Mangin. 2019. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.* 3, ICFP (2019), 86:1–86:29. https://doi.org/10.1145/3341690

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 14–27. https://doi.org/10.1145/3167092

Josef Svenningsson and Emil Axelsson. 2012. Combining Deep and Shallow Embedding for EDSL. In *Trends in Functional Programming - 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7829)*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer, 21–36. https://doi.org/10.1007/978-3-642-40447-4_2

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.* 4, ICFP (2020), 121:1–121:30. https://doi.org/10.1145/3409003

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 387–398. https:

//doi.org/10.1145/2491956.2491978

S. Doaitse Swierstra and Luc Duponcheel. 1996. Deterministic, Error-Correcting Combinator Parsers. In *Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text (Lecture Notes in Computer Science, Vol. 1129)*, John Launchbury, Erik Meijer, and Tim Sheard (Eds.). Springer, 184–207. https://doi.org/10.1007/3-540-61628-4_7

Wouter Swierstra and Tim Baanen. 2019. A predicate transformer semantics for effects (functional pearl). *Proc. ACM Program. Lang.* 3, ICFP (2019), 103:1–103:26. https://doi.org/10.1145/3341707

Philip Wadler. 1990. Recursive types for free! http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt Draft.

Philip Wadler. 1992. Comprehending Monads. *Math. Struct. Comput. Sci.* 2, 4 (1992), 461–493. https://doi.org/10.1017/S0960129500001560

Philip Wadler. 1998. The Expression Problem. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt Email correspondence.

Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged selective parser combinators. *Proc. ACM Program. Lang.* 4, ICFP (2020), 120:1–120:30. https://doi.org/10.1145/3409002

Li-yao Xia. 2019. Free applicative functors in Coq. https://blog.poisson.chat/posts/2019-07-14-free-applicative-functors.html Blog post.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473572

Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 71–84. https://doi.org/10.1145/3372885.3373813

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32