Program Adverbs

Structures for Embedding Effectful Programs

Yao Li and Stephanie Weirich

University of Pennsylvania
{liyao, sweirich}@cis.upenn.edu

Abstract. We propose a new class of structures called *program adverbs* designed to abstractly represent effectful programs in theorem provers like Coq. Program adverbs enable formal reasoning about general properties that hold for certain computation patterns regardless of the effects that may occur during computation. Program adverbs are composable, allowing them to be used to model languages containing different characteristics.

We define program adverbs and propose a criterion to guarantee the soundness of their theories. Following these definitions, we identify five basic program adverbs and two add-on adverbs. We demonstrate that program adverbs are useful general structures via three examples based on distinct programming languages.

1 Introduction

Suppose that you want to formally verify a program, written in a language such as Verilog [11], Haskell [14], or C [16]. The first step of verifying the program would be embedding it in a language that is amenable to formal reasoning, such as Coq [28]. This step is known as *semantic embedding* [3].

There are many different approaches to semantics embedding. The two most well-known were proposed by Boulton et al. [3]: (1) *Shallow embeddings*, which represent terms of the embedded language directly using terms of the embedding language; (2) *Deep embeddings*, which represent terms using abstract data types.

Shallow embeddings are usually easier to work with, both in terms of defining the embedding and reasoning about it, because we can reuse the semantics of the embedding language as well as the theories already available. However, there are several limitations to shallow embedding: (1) The embedded language's semantics is not explicitly formalized, so there might be a hidden discrepancy between the original program and the embedded program; (2) Certain features in the embedded language might not be directly available in the embedding language, *e.g.,* side effects; (3) Programs' syntactic structures are not available in the shallow embedding, so we cannot state theorems about classes of programs.

Deep embeddings address all three limitations of shallow embeddings by *reifying* the source language using data types—the semantics is defined separately as an "interpretation" of the data types. However, defining the semantics and developing the metatheory for the language can be a huge effort—even though

there are many techniques or tools [2, 41] to facilitate the process, it remains nontrivial.

Fortunately, we are not restricted to a binary choice between the two styles. There are also many *mixed embeddings* between them: a language might have some of its terms embedded "shallowly" while the others embedded "deeply". Some examples include the characteristic formulae [6], and the mixed embeddings based on the delay monad [4] or the resumption monad [39].

Recently, much attention has been given to mixed embeddings based on freer monads or their variants [10, 12, 17, 21, 29, 36, 45, 52]. These mixed embeddings are useful for representing many kinds of effectful programs: in these embeddings, the pure parts of programs are represented "shallowly" while the effectful parts are represented "deeply".

However, freer monads are not the only structures that we can use to embed effects and, depending on the language to be represented, they might even be unsuitable (more detail in Section 2).

The major contribution of this paper is the generalization of freer monads to a class of data structures called *program adverbs*. Program adverbs model general computation patterns commonly used in effectful programs. One example of a program adverb is "statically", which indicates that data flow and control flow in the semantics are fixed. Another example of program adverbs is "in parallel," which models parallel computation. Program adverbs are useful because they summarize properties that are true regardless of what effects happen in the program.

We make the following contributions:

- We discuss the limitations of shallow embeddings and mixed embeddings based on freer monads in formal reasoning (Section 2).
- We define program adverbs and propose a criterion for checking the *soundness* of their theories (Section 3).
- We refactor program adverbs to support composition. We motivate why we want to compose program adverbs and define a composition algebra (Section 4).
- We identify five basic program adverbs from commonly used type classes 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 implement composable program adverbs using the Coq proof assistant [28]. A major challenge for implementing them in Coq is supporting extensible inductive data types [50]. We address this challenge by adapting the *Meta Theory à la Carte* (MTC) approach [9].
- We demonstrate the usefulness of program adverbs via three distinct language examples including a simple circuit language, Haxl [26], and a networked server adapted from Koh et al. [18]. (Section 5).

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

$$
\begin{array}{rll}
\textit{literals} & b & ::= \texttt{true} \mid \texttt{false} \\
\textit{terms} & t, u & ::= x \mid b \mid \neg t \mid t \wedge u \mid t \vee u
\end{array}
$$

$$
\begin{array}{rlrl}
[\![\texttt{true}]\!] &= \texttt{ret true} & [\![\texttt{false}]\!] &= \texttt{ret false} \\
[\![x]\!]_R &= \texttt{(\$x) <\$> ask} & [\![x]\!]_F &= \texttt{embed (GetData x)} \\
[\![\neg t]\!] &= \texttt{negb <\$>} [\![t]\!] & & \\
[\![t \wedge u]\!] &= \texttt{t'<-} [\![t]\!]\texttt{; u'<-} [\![u]\!]\texttt{; ret (andb t' u')} & & \\
[\![t \vee u]\!] &= \texttt{t'<-} [\![t]\!]\texttt{; u'<-} [\![u]\!]\texttt{; ret (orb t' u')} & &
\end{array}
$$

Fig. 1: The syntax of language $\mathcal{B}$ and a translation for embedding language $\mathcal{B}$ in Coq with two different monads. We use $[\![t]\!]_R$ to represent t's embedding in a reader monad. We use $[\![t]\!]_F$ to represent t's embedding in freer monads. When the translation is the same for both monads, we write $[\![t]\!]$. Additionally, we use the infix operator `<$>` to represent a functor's `fmap` method. We use `($x)` to represent `(fun f => f x)`. We also use 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.

2 Representing Circuits

In this section, we demonstrate shallow embeddings and freer-monad-based mixed embeddings by embedding a simple circuit language $\mathcal{B}$ in Coq [28]. To distinguish the embedded language and the embedding language, we use mathematical notation to describe the former and use Coq code to describe its embedding.

The syntax of $\mathcal{B}$ appears in the top part of Fig. 1. Compared to a simple boolean algebra, $\mathcal{B}$ can read from the variables that represent references to external devices.

To use a shallow embedding to represent language $\mathcal{B}$, we need a way of representing the effects of reading from external devices—the most common way of doing this is using *monads* [32, 49]. But *which* one?

A shallow embedding using the reader monad. For a first attempt, we can use the reader monad, whose core definitions are shown in Fig. 2. Suppose that we have a Coq type `var` that represents variables, we use the type `var -> bool` to represent a map from `var` to `bool`. Then we can embed $\mathcal{B}$ using `Reader (var -> bool)`. The translation is given in the bottom part of Fig. 1.

By embedding $\mathcal{B}$ using the reader monad, we can now reason about properties of circuits just like reasoning about Coq programs. For example, we can show that x is equivalent to $x \wedge \texttt{true}$ by proving the following property in Coq:

```
Theorem and_true : forall x,
  ask x = t <- ($x) <$> ask ; u <- ret true ; ret (andb t u)
```

```
Definition Reader (V A : Type) : Type := V -> A.

Definition ret {V A} (a : A) : Reader V A := fun _ => a.

Definition bind {V A B}
           (m : Reader V A) (k : A -> Reader V B) : Reader V B :=
  fun v => k (m v) v.

Definition ask {V} : Reader V V := fun v => v.
```

Fig. 2: The core definitions of the reader monad (simplified).

The proof follows from the theories of Coq's `bool` type and the `Reader` monad.[1]

The shallow embedding is appealing. The semantic of the embedded language is easy to define—all we need is a translation strategy like Fig. 1—and the subsequent formal reasoning is also straightforward as we can reuse Coq's existing theories and proof tools. However, there are several limitations:

1. Is the reader monad really the right choice for representing our circuits? The answer is no. For example, the reader monad assumes that the value in the external device does not change—this may not be true. A consequence of choosing a wrong representation is that we could potentially prove wrong theorems. Consider the property $[\![x]\!] \cong [\![x \wedge x]\!]$. We can prove that this is true with our shallow embedding, but it shouldn't be! The value on the device might change and we can observe that if there is a delay between the circuit's two reads.
2. If we revisit the `and_true` theorem, we can note that the theorem does not rely on the wrong assumption we have discussed in the first point—the circuit only reads from `x` once in either side of the equation. This observation indicates that the `and_true` theorem is more general than the particular interpretation we have chosen. This is fine if we are committed to one interpretation, but if we ever change our mind (*e.g.,* because we chose the wrong interpretation), we need to prove the property again under the new embedding.
3. This approach shares the common limitations of shallow embeddings. For example, we cannot state properties about $\mathcal{B}$ programs that relate to the structure of the circuit, *e.g.,* the number of variables cannot be larger than 2 to the power of the circuit's depth.

A mixed embedding using freer monads. To avoid committing to one particular effect interpretation, we can make the embedding deeper by using a mixed embedding based on *freer monads* [17], an approach that has been explored in

[1] We also need to use the axiom of functional extensionality or replace the definitional equality with the pointwise equality, since the data type underlying a reader monad is a function.

```
Inductive FreerMonad (E : Type -> Type) (R : Type) :=
| Ret (r : R)
| Bind {X} (m : E X) (k : X -> FreerMonad E R).

Definition ret {E A} : A -> FreerMonad E A := Ret.

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.

Definition embed {E A} (e : E A) : FreerMonad E A := Bind e Ret.
```

Fig. 3: The core definitions of freer monads (simplified).

various works [12, 21, 29, 36, 52]. In this type of mixed embedding, the pure parts of the program are embedded shallowly, while the effects are embedded deeply as abstract, uninterpreted data types "connected" by freer monads.

We show the core definitions of freer monads in Fig. 3. 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 uninterpreted effects `E` in a right-associative monadic structure. `FreerMonad` is a monad as demonstrated by the `ret` and `bind` functions. The `ret` function is simply a thin wrapper of the `Ret` data constructor. The `bind` function does a pattern match on its first argument `m`, and puts the second arguments `k` to the continuation of `m`. In addition to `ret` and `bind`, we define an `embed` function that "embeds" an effect.

As an example of an abstract effect `E`, we model reading data from external devices in $\mathcal{B}$ as follows:

```
Variant DataEff : Type -> Type :=
| GetData (v : var) : DataEff bool.
```

`DataEff` is of `Type -> Type` and its parameter represents the return type of the effect. The `GetData` constructor represents a data retrieval with the variable `v : var` that returns an unknown `bool`. We say that `DataEff` is an *abstract* effect representation because it says nothing about how data read actually happens in $\mathcal{B}$.

We show how to embed language $\mathcal{B}$ using freer monads in Fig. 1. 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), which uses the `embed` function (Fig. 3).

Compared with using the reader monad, we lose the ability to reason about particular effects *directly*.[2] This addresses the first limitation of shallow embeddings: because we do not know how effects are implemented, we can no longer prove the over-specialized property that $[\![x]\!] \cong [\![x \wedge x]\!]$.

The ignorance of effect implementation also *partially* addresses the second limitation of shallow embeddings. Consider the `and_true` theorem again: The theorem is still provable with this mixed embedding because it is a direct consequence of the monad laws, which freer monads satisfy. Furthermore, since we do not know how effects are implemented, we have proved a general theorem that is true regardless of the effect interpretation.

We used the word "partially" above. To see why, consider the property $[\![t \wedge u]\!] \cong [\![u \wedge t]\!]$, expressed in Coq as:

```
Theorem and_comm: forall x y,
  t <- embed (GetData x) ; u <- embed (GetData y) ; ret (andb t u) =
  t <- embed (GetData y) ; u <- embed (GetData x) ; ret (andb t u)
```

We cannot prove this property because monads are not commutative in general. For example, we might change the value at `y` by accessing `x`.

However, recall that we are reasoning about circuits: an and gate in a circuit does not impose a fixed order in reading from its inputs—it does so in parallel. In this case, the commutativity rule should be true, but we cannot prove it with a mixed embedding based on freer monads.

The reason freer monads fail here is similar to the reason the shallow embedding based on the reader monad fails: we choose the wrong data type. However, we say that we incorrectly interpret the *effects* by choosing the reader monad, what part are we incorrectly interpreting by choosing freer monads?

The program adverbs. Just like the reader monad models one particular effect, freer monads model *one* particular computation pattern. Therefore, in this paper, we generalize the idea behind freer monads and define a class of structures that capture different computational characteristics.

We call these structures *program adverbs*. Program adverbs might be better 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, we consider mutable states, I/O, exceptions, *etc.* as effects, but we consider *parallel computation* as a program adverb—referring to it as "in parallel", in its adverb form.

The rest of this paper shows what program adverbs are and demonstrates why they are useful.

[2] We can still reason about particular effects *indirectly* by interpreting freer monads to a particular monad, like the reader monad—the interpretation essentially returns a shallow embedding.

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

Fig. 4: The adverb data type of the `Statically` adverb.

3 Program Adverbs

A program adverb is composed of two parts: its "form" and its "content". More specifically, we define *program adverbs* as follow:

Definition 1 (Program Adverb). *A program adverb is a tuple* $(D, \cong, \subseteq)$. *D is called the adverb data type and is parameterized by an effect type E and a return type R. The $\cong$ and $\subseteq$ operations are called the adverb theories. They are binary operations that define, respectively, the bisimulation and refinement relations on $D(E, R)$ for all E and R.*

The adverb data type D is the "form" of the program adverb. In Coq terms, it 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 theories $\cong$ and $\subseteq$ are the "content". Let us demonstrate program adverbs in more detail by closely inspecting one adverb in particular, the `Statically` adverb.

3.1 Adverb Data Types

Figure 4 shows the adverb data type of `Statically`. It consists of three constructors: `EmbedA`, which "embeds" an *effect* (of type `E R`) into the adverb; `Pure`, on the other hand, "embeds" a pure computation into the adverb; Finally, `LiftA2` "connects" two computations that use the `Statically` adverb.

Readers who are familiar with Haskell might recognize that the `Pure` and `LiftA2` constructors are reminiscent of the key operations of an abstract interface called applicative functors [30]. Indeed, the `Statically` adverb is a *reification* of the abstract interface of applicative functors.

We show the definition of applicative functors as a Coq type class in Fig. 5. The applicative functor class, *i.e.,* `Applicative`, is a subclass of `Functor`. It has two key operations: `pure` and `liftA2`.[3] The `pure` operation embeds a pure value in the applicative functor and the `liftA2` operation connects two applicative functors. We define `StaticallyAdv` simply by converting these operations to data

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

```
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 }.
```

Fig. 5: The abstract interfaces of functors and applicative functors shown as Coq type classes. The definitions have been simplified for readability. In our actual Coq development, we use the type classes translated from Haskell using `hs-to-coq` [44].

constructors, substituting `F` with `StaticallyAdv E`, and adding one additional `EmbedS` constructor.

A `StaticallyAdv` combined with any effect `E` is an instance of `Functor` and `Applicative`. The `pure` and `liftA2` operations are thin wrappers of `Statically`'s data constructors. The `fmap` operation can be implemented using `pure` and `liftA2`:

```
Definition fmap (f : A -> B) := liftA2 id (pure f).
```

The data dependency in the `liftA2` method shows that the `Applicative` interface imposes a "static" data flow and control flow on the computation: we will always need to run both parameters of type `F A` and `F B` to get the result of type `F C`, *i.e.,* we cannot skip either computation. In addition, there is no dependency between the two parameters, which allows us to statically inspect either of them without running the other.

The correspondence between the `Statically` adverb and the applicative functors is not a coincidence. In fact, all program adverbs correspond to classes of functors. We make this correspondence more precise in the next section.

3.2 Adverb Simulation

One important property of `StaticallyAdv` is that it can "simulate" any other instance of the `Applicative` class. We can show this via the abstract interpreter `interpA` shown in Fig. 6. 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 `StaticallyAdv E A` to an `I A` for any type `A`.

Why is it important that `StaticallyAdv` 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 correspond 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 in the program.

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

```
Fixpoint interpA {E I : Type -> Type} `{Applicative I} {A : Type}
         (interpE : forall A, E A -> I A) (t : StaticallyAdv 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. 6: The interpretation from `StaticallyAdv` to any instance of the `Applicative` type class.

Definition 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 the `Statically` adverb, but we will see why it is useful in Section 3.4.

We also define an *adverb interpretation* as follow:

Definition 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. 6 is an adverb interpretation. More specifically, we say that

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

where the `IdT` transformer is an identity 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 Adverb Theories

In addition to an adverb data type, every program adverb also comes with some theories, defined by a bisimulation relation $\cong$ and a refinement relation $\subseteq$. To prove theorems about adverbs, we want these relations to relate as many objects as possible, but also avoid relating computationally non-related objects.

To relate as many objects as possible, we directly encode all the properties we want in the theories. For the bisimulation relation $\cong$ of the `Statically` adverb, we define it using three sorts of equivalence relations: (1) the congruence rule with respect to `liftA2`, (2) the laws of applicative functors, and (3) the equivalence properties (*i.e.,* reflexivity, symmetry, transitivity). For the refinement relation

$\subseteq$, we define it using three sorts of properties: (1) the congruence rule with respect to `liftA2`, (2) the pre-order properties (*i.e.,* reflexivity and transitivity), and (3) the fact that $\cong$ implies $\subseteq$.

To avoid relating computationally non-related objects, we also show that the adverb theories $\cong$ and $\subseteq$ satisfy the following *soundness* property:

Definition 4 (Soundness of Adverb Theories). *Given a program adverb* $(D, \cong, \subseteq)$ *and an adverb interpretation* $I \in D \models_T C$*, we say that the adverb theories* $\cong$ *and* $\subseteq$ *are sound with respect to* I *if there exist a lawful equivalence relation* $\equiv$ *and a lawful preorder relation* $\leq$ *on* $T(C)$*, such that for all* $d_1, d_2 \in D$*,*

1. $d_1 \cong d_2 \implies I(d_1) \equiv I(d_2)$*, and*
2. $d_1 \subseteq d_2 \implies I(d_1) \leq I(d_2)$*.*

Let us use `idT` for the transformer T for the moment. The equivalence relation $\equiv$ and pre-order relation $\leq$ on C are lawful if they respect the congruence laws and the class laws of C. In addition, $a \equiv b$ should imply $a \leq b$ for any a and b. For `Applicative`, we use the common applicative functor laws regarding $\equiv$.[4] There is no class law regarding $\leq$.

Definition 5 (Soundness of Program Adverbs). *Given a program adverb* $(D, \cong, \subseteq)$ *and an adverb interpretation* $I \in D \models_T C$*, we say that the adverb is sound if the* $\cong$ *and* $\subseteq$ *relations are sound with respect to* I*.*

Theorem 1 (Soundness of the `Statically` Adverb). *The `Statically` adverb is sound with respect the adverb interpretation `interpA` ∈ `StaticallyAdv` ⊨ `Applicative`.*

Proof. By induction over the $\cong$ and $\subseteq$ relations, respectively. □

3.4 "Statically and in Parallel"

Before getting to other adverbs, let's look at a variant of the `Statically` adverb: `StaticallyInParallel`. As its name suggests, it adds parallelization to a static computation.

Recall that the two parameters of type `F A` and `F B` in `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 [26].

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 $\cong$ relation:

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

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

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

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

Definition purePowerSet {A : Type} (a : A) : PowerSet I A :=
  fun r => EqI 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' /\
           (EqI (liftA2 f a' b') r \/ EqI (liftA2 (flip f) b' a') r).

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

Fig. 7: The core definitions of a powerset applicative functor transformer (simplified).

This rule, also known as the *commutativity* rule, states that the order that effects happen 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 adverb theories, as it might be a theory that only holds for certain effects. Nevertheless, we can prove the soundness of th adverb theories with respect to the following adverb simulation:

$$\texttt{StaticallyInParallelAdv} \models_{\texttt{PowerSet}} \texttt{Applicative}.$$

The `PowerSet` transformer is *a powerset applicative functor transformer* and its core definitions are shown in Fig. 7. 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 the `EqI` relation, which is the lawful $\equiv$ relation on `I`. We omit the pre-order relation here.

Lemma 1. *If `EqI` is a lawful equivalence relation on `Applicative`, `EqPowerSet` is a lawful equivalence relation on `Applicative` that additionally satisfies the commutativity rule.*

Proof. This is true by definition. □

Theorem 2 (Soundness of the `StaticallyInParallel` adverb). *The adverb is sound with respect to* $\texttt{StaticallyInParallelAdv} \models_{\texttt{PowerSet}} \texttt{Applicative}$.

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

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

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

Fig. 8: The adverb data types for `Streamingly`, `Conditionally`, and `Dynamically`.

Proof. We can construct an interpreter $P \in$ `StaticallyInParallelAdv` $\models_{\texttt{PowerSet Applicative}}$ by modifying `interpA` (Fig. 6) 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 1. □

Intuitively, we can define `StaticallyInParallel` as 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

`Streamingly`. This program adverb 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 : (A -> B) -> F A -> F B` interface.

`Statically` and `StaticallyInParallel`. We have already seen these two adverb in previous sections. They are both useful for embedding a language whose data flow and control flow are static. In addition, their structures allow for static analysis without interpreting them [5].

`Conditionally`. We use this adverb to model conditional execution. This adverb is similar to `Statically` and `Nondeterministically`. The major difference is: `Nondeterministically` runs one of the two computations it "connects," `Statically` runs both of them, and `Conditionally` might run one computation or both of them depending on the result of the first computation.

The definition of its adverb data type is shown in Fig. 8. It reifies the `Selective` type class [34]. The signature operation of `Selective` is the `select` operation, which has the type `F (A + B) -> F (A -> B) -> F B`. The first parameter

encapsulates a computation that could either return an `A` or a `B`. In the first case, we must run the second computation of `F (A -> B)` to get the result. However, we can still run the second computation even if we don't need to.

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 the exact number.

Even though we derive this adverb by reifying `Selective`, we do not wish to model the adverb's theories using the laws of selective functors [34]. 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 if they don't need to) 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:

$$\texttt{select (inr <\$> a) b} \cong \texttt{a}$$

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

$$\texttt{ConditionallyAdv} \models \texttt{Monad}$$

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

`Dynamically`. This adverb adverb simulates `Monad`. A monad is the most expressive and dynamic among all four classes of functors thanks to its core operation `bind : F A -> (A -> F B) -> F B`. Any kind of computation of `F` 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 analysis.

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

3.6 Remark

The definitions of the basic program adverbs are similar to those of free or freer structures [5, 17, 33, 34]. Indeed, we have considered using free or freer structures

to implement program adverbs, but we eventually choose the version presented in the paper. We defer the detailed comparison between program adverbs and free structures to Section 6.2.

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. And we can derive a selective instance from an applicative functor and vice versa.[5] 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 [26]. In the program adverbs terminology, the semantics of their language is composed of a "statically 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 $\oplus$, which joins two adverbs together.

But before getting to the algebra of program adverbs and $\oplus$, we need to make some changes to the relation between effects and program adverbs.

4.1 Uniform Treatment of Effects and Program Adverbs

In many semantic frameworks for programming languages with effects, effects are considered as secondary to monads [8, 22, 32]. 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 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

[5] 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. [34].

$$\textit{effects and adverbs} \quad A, B, C \quad ::= \texttt{Effect}\ e \mid \texttt{BasicAdverb}\ a \mid A \oplus B$$

$$\begin{aligned} \textsc{Commutativity} &: & A \oplus B &\equiv B \oplus A \\ \textsc{Associativity} &: & A \oplus B \oplus C &\equiv A \oplus (B \oplus C) \end{aligned}$$

Fig. 9: The algebra for effects and program adverbs.

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 4.3). 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 An Algebra for Effects and Program Adverbs

Figure 9 shows our algebra for effects and program adverbs. As we have discussed, we treat effects and adverbs uniformly so that they can be composed with each other via the $\oplus$ operation. The rules are simple, thanks to the uniform treatment of effects and program adverbs,

In addition, we define an equivalence relation $\equiv$ on effects and program adverbs, which states that there is a bijection between them. We can show that the $\oplus$ operation is commutative and associative with respect to $\equiv$.

4.3 The Coq Implementation

A challenge of implementing the algebra in Coq is that we need to encode extensible inductive types, which are not directly supported by most theorem provers. Fortunately, *Meta Theory à la Carte* (MTC) [9] provides a technique for implementing this in Coq using Church encodings of data types [40, 48]. 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 also need to change the definitions of adverb data types and adverb theories to make them "recurse" on a recursive parameter instead (we will see concrete definitions shortly in the next section).

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

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

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

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

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

Fig. 10: The composable adverb data types.

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 [9].

4.4 Deconstructing Program Adverbs

Figure 10 shows the definitions of composable adverb data types. Compared with the adverb data types in Fig. 4 and 8, a composable adverb data type replaces the effect parameter (which is named as `E`) with a recursive parameter (which is named as `K`) and "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 `PurelyAdv`. 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.

4.5 Add-on Adverbs

Another benefit of making program adverbs composable is that we can now define two add-on adverbs, adverbs that are not suitable as standalone adverbs. We show the definitions of these add-on adverbs in Fig. 11.

`Repeatedly`. We define this adverb to model that a computation can be executed nondeterministically many times, but at least once. The functionality is similar to that of the Kleene plus.[6] We establish the following adverb simulation relation

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

```
Variant RepeatedlyAdv (K : Set -> Set) (R : Set) : Set :=
| Repeat : K R -> RepeatedlyAdv K R.

Variant NondeterministicallyAdv (K : Set -> Set) (R : Set) : Set :=
| Choice : K R -> K R -> NondeterministicallyAdv K R.
```

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

```
Definition bindPowerSet {A B : Type} (m : PowerSet I A)
           (k : A -> PowerSet I B) : PowerSet I B :=
  fun r => exists m' k', m m' /\ (forall a, k a (k' a)) /\
         (EqI (m' >>= k') r).

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

Definition powerSet {A : Type} (a : PowerSet I A) : PowerSet I A :=
  fun r => exists n, seq a n r.

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

Fig. 12: The `powerSet` and `choicePowerSet` functions defined on the `PowerSet` transformer.

between it and `Monad`:

$$\texttt{RepeatedlyAdv} \models_{\texttt{PowerSet}} \texttt{Monad}$$

We interpret the `Repeat` constructor using the `powerSet` function shown in Fig. 12.

There is only one adverb theory for this adverb (shown in Fig. 13). The `seq` function is defined in the same way as that in Fig. 12, but works on different types: it repeats the computation in its first parameter by $n + 1$ times. The adverb theory says that repeating `a` by *any* positive number of times refines `repeat a`.

Note that `Repeatedly` models a Kleene plus rather than a Kleene star because no empty element is defined. It is possible, however, to compose `RepeatedlyAdv` and `PurelyAdv` to obtain an adverb representing a Kleene star.

`Nondeterministically`. We define this adverb to model nondeterministic choices. The `Choice` constructor models the situation when we can nondeterministically *choose* one of two computations to run. We show the following adverb simulation

$$\textsc{Seq} \quad : \quad \forall n, \texttt{seq a n} \subseteq \texttt{repeat a}$$

$$\begin{aligned}
\textsc{Commutativity} &: \texttt{choice a b} \cong \texttt{choice b a} \\
\textsc{Associativity} &: \texttt{choice a (choice b c)} \cong \texttt{choice (choice a b) c} \\
\textsc{Left Choice} &: \texttt{a} \subseteq \texttt{choice a b} \\
\textsc{Right Choice} &: \texttt{b} \subseteq \texttt{choice a b}
\end{aligned}$$

Fig. 13: The adverb theories for `Repeatedly` and `Nondeterministically`.

relation for this adverb:

$$\texttt{NondeterministicallyAdv} \models_{\texttt{PowerSet}} \texttt{Functor}$$

The theories of this adverb is shown in Fig. 13.

The `Nondeterministically` adverb is similar to the `Alternative` and `MonadPlus` type classes found in Haskell. Indeed, we can make it an instance of those type classes by composing it with `PurelyAdv` $\oplus$ `StaticallyAdv` or `PurelyAdv` $\oplus$ `DynamicallyAdv`, respectively.

5 Examples

5.1 Circuits

In our first example, we re-embed the circuit language $\mathcal{B}$ using program adverbs (recall the syntax of $\mathcal{B}$ in Fig. 1). For $\mathcal{B}$, we can just use the `StaticallyInParallel` adverb presented in Section 3.4 and we do not need composable adverbs. Our new translation is largely the same as the translation using freer monads, except for the following two terms:

$$\begin{aligned}
[\![t \wedge u]\!] &= \texttt{liftA2 andb } [\![t]\!] \ [\![u]\!] \\
[\![t \vee u]\!] &= \texttt{liftA2 orb } [\![t]\!] \ [\![u]\!]
\end{aligned}$$

The `liftA2` operations call the `LiftA2` constructors of `StaticallyInParallelAdv` under the hood to create a tree-like structure for representing the structure of the circuits.

With this embedding, we can now describe properties about the structures of circuits in Coq. For example, we can define functions that computes the depth of a circuit or the number of variables a circuit accesses without relying on any effect interpretations. We can also prove the following property in Coq:

```
Theorem heightAndVar : forall (c : StaticallyInParallelAdv DataEff bool),
    numVar c <= Nat.pow 2 (depth c).
```

The theorem states that the number of variables in a circuit cannot be larger than 2 to the power of the circuit's depth. The theorem is provable by an induction over `c`.

In addition, by choosing the `StaticallyInParallel` adverb, we get the commutativity property for free.

5.2 Haxl

Haxl is a Haskell library developed and maintained by Facebook that automatically parallelizes certain operations to achieve better performance [26]. 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 `Applicative` instance and those of its `Monad` 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 [27]. 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.[7] 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.):

```
PurelyAdv ⊕ StaticallyInParallelAdv ⊕ DynamicallyAdv ⊕ DataEff
```

We use `StaticallyInParallelAdv` to model the batched operations and their "parallel" nature and we use `DynamicallyAdv` to model the sequential operations. When using this composition, we need to aware that both `StaticallyInParallelAdv` and `DynamicallyAdv` 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.[8] In our Coq development, we take the second approach and assign a lower priority to `DynamicallyAdv`'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

[7] Tools like `hs-to-coq` [44] can be adapted to implement the translation.

[8] `https://coq.inria.fr/refman/addendum/type-classes.html`

```
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 (simplified).

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.[9] 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.

If we are more careful, we should interpret our embedding using the `PowerSet` transformer (Fig. 7), because we have used the `StaticallyInParallel` adverb. However, we are fine with using the `DB` monad here since it also satisfies the commutativity rule.

5.3 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 [13,

[9] The `DB` monad is *not* a state monad, because the combined reader and writer monads are instantiated with different types of states.

```
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.

18, 24, 53]. This approach offers better abstraction and modularity, as at each layer, we only need to consider certain subsets of properties.

In this example, we show that our mixed embeddings based on program adverbs are useful as some of those intermediate layers. We illustrate this via a networked echo server adapted from that of Koh et al. [18]. 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 it 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. [18], the server is implemented using a single-process *event loop* [37]. 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 [38]. 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 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 many 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 Koh et al.'s first refinement. 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.

The embedding 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:

```
RepeatedlyAdv ⊕ NondeterministicallyAdv ⊕ PurelyAdv ⊕ DynamicallyAdv ⊕
  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.

Unfortunately, we are not able to complete the end-to-end proof by the time of the submission—we are at trouble in proving the refinement between the second and the third layer. We conjecture that this is because we are missing some theory of the `Repeatedly` adverb. However, the rest of the refinements have all been proved simply by using existing adverb theories, which demonstrates the usefulness of program adverbs.

6 Discussion

6.1 Church Encodings

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 [9]. 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 [43], the QuickChick plugin [19], *etc.*). However, this situation can be helped by developing tools or plugins for Church encodings.

The other consequence is that, following the practice of MTC, we use Coq's impredicative set extension, an extension that is known to be incompatible with classical axioms.[10]

6.2 Free/Freer Structures

Free and freer structures [5, 17, 33, 34] are another way to reify classes of functors.[11] The biggest difference between adverb data types and these free/freer structures are the parameters they recurse on: all the adverb data types recurse on both their computational parameters, while each free/freer structure only recurses on one of them. Therefore, a free/freer 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 (plus the axiom of functional extensionality). This makes them appealing candidates for implementing program adverbs. However, we find that the conversions to normal forms are not always desirable in our work. Taking `Statically` or `StaticallyInParallel` 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 5.1 and Section 5.2.

7 Related work

Freer Monads and Variants Freer monads [17] are studied by many researchers in formal verification to reason about programs with effects. For example, Letan et al. [21] develop a modular verification framework based on effects and effect handlers called FreeSpec. Xia et al. [52] 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. [18] encode freer monads in VST [1] to reason about networked servers. Christiansen et al. [7] develop a framework based on free monads and containers for reasoning about Haskell programs with effects.

Among many variants of freer monads, one particular structure closely resembles program adverbs. That is the action trees defined in Swamy et al. [45]. The action trees have four constructors, `Act`, `Ret`, `Par`, and `Bind`, whose types correspond to effects, `PurelyAdv`, `StaticallyAdv`, and `DynamicallyAdv` in composable program adverbs, respectively, another evidence that program adverbs are general models. Compared with the action trees, program adverbs can be composed with effects and other program adverbs as needed. On the other hand, action

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

[11] However, the commonly used definition of free monads cannot be encoded in Coq because it is not strictly positive [10]. The common ways to work around this problem are: (1) using containers [10], or (2) using freer monads [17, 20, 29, 52].

trees are embedded with separation logic assertions, which program adverbs do not have.

Formal Reasoning with Freer Monads There are many works that explore formal reasoning with freer monads. Zakowski et al. [54] propose a technique called generalized parameterized coinduction for developing equational theory for reasoning about the interaction trees. Zakowski et al. [53] use the interaction trees to define a modular, compositional, and executable semantics for LLVM. Silver and Zdancewic [42] connect interaction trees with Dijkstra monads [25] for writing termination sensitive specifications based on uninterpreted effects.

On the other hand, some work take a different view on freer monads by treating them as a "syntactic" structures, and they focus their reasoning on the interpreted monads. For example, Letan and Régis-Gianas [20] reason about freer monads using effect handlers; Swierstra and Baanen [47] interpret freer monads into a predicate transformer semantics that is similar to Dijkstra monads; Nigron and Dagand [36] interprets freer monads using separation logic.

Other Free/Freer Structures Other free structures are also explored by various works. Capriotti and Kaposi [5] propose two variants of freer applicative functors, which correspond to the left- and right-associative variants, respectively. Xia [51] explores defining freer applicative functors in Coq, and points out that the right associative variant is harder to define in Coq. Milewski [31] discusses how to derive freer monoidal functors.[12] Mokhov [33] defines the freer selective applicative functors.

In the context of formal reasoning, one of the main inspirations of our work is Capriotti and Kaposi [5]. They observe that the structures of freer monads are not amenable to static analysis and propose freer applicative functors. Our work takes the observation further and identifies a class of composable program adverbs.

Programming Abstractions We are not the first to observe that monads are too dynamic for certain tasks. For example, Swierstra and Duponcheel [46] identify that a parser that has some static features cannot be defined as a monad. Inspired by their observation, Hughes [15] proposes a new abstract interface called arrows. The relationship among arrows, applicative functors, monads are studied by Lindley et al. [23].

In Mokhov et al. [35], the authors observe that the data type of the tasks of 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. We consider the concept of program adverbs an extension of this observation to denotational semantics.

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

8 Acknowledgment

We thank Conal Elliott, Andrey Mokhov, Benjamin Pierce, Antal Spector-Zabusky, Nikhil Swamy, Val Tannen, and Steve Zdancewic for their feedback during discussion. We also thank people in Penn PLClub and on CoqClub Zulipchat including Paolo Giarrusso, Li-yao Xia, and Irene Yoon for their technical help.

Bibliography

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

[2] Aydemir, B.E., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 3–15. ACM (2008), https://doi.org/10.1145/1328438.1328443

[3] Boulton, R.J., Gordon, A.D., Gordon, M.J.C., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In: Stavridou, V., Melham, T.F., Boute, R.T. (eds.) 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, pp. 129–156. North-Holland (1992)

[4] Capretta, V.: General recursion via coinductive types. Log. Methods Comput. Sci. 1(2) (2005), https://doi.org/10.2168/LMCS-1(2:1)2005

[5] Capriotti, P., Kaposi, A.: Free applicative functors. In: Levy, P., Krishnaswami, N. (eds.) Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014. EPTCS, vol. 153, pp. 2–30 (2014), https://doi.org/10.4204/EPTCS.153.2

[6] Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 418–430. ACM (2011), https://doi.org/10.1145/2034773.2034828

[7] Christiansen, J., Dylus, S., Bunkenburg, N.: Verifying effectful Haskell programs in Coq. In: Eisenberg, R.A. (ed.) Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019. pp. 125–138. ACM (2019), https://doi.org/10.1145/3331545.3342592

[8] Darais, D., Labich, N., Nguyen, P.C., Horn, D.V.: Abstracting definitional interpreters (functional pearl). Proc. ACM Program. Lang. 1(ICFP), 12:1–12:25 (2017), https://doi.org/10.1145/3110256

[9] Delaware, B., d. S. Oliveira, B.C., Schrijvers, T.: Meta-theory à la carte. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 207–218. ACM (2013), https://doi.org/10.1145/2429069.2429094

[10] Dylus, S., Christiansen, J., Teegen, F.: One monad to prove them all. Art Sci. Eng. Program. 3(3), 8 (2019), https://doi.org/10.22152/programming-journal.org/2019/3/8

[11] Flake, P., Moorby, P., Golson, S., Salz, A., Davidmann, S.J.: Verilog HDL and its ancestors and descendants. Proc. ACM Program. Lang. 4(HOPL), 87:1–87:90 (2020), https://doi.org/10.1145/3386337

[12] Foster, S., Hur, C., Woodcock, J.: Formally verified simulations of state-rich processes using interaction trees in isabelle/hol. CoRR abs/2105.05133 (2021), https://arxiv.org/abs/2105.05133

[13] Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X.N., Weng, S., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. pp. 595–608. ACM (2015), https://doi.org/10.1145/2676726.2676975

[14] Hudak, P., Hughes, J., Jones, S.L.P., Wadler, P.: A history of Haskell: being lazy with class. In: Ryder, B.G., Hailpern, B. (eds.) Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007. pp. 1–55. ACM (2007), https://doi.org/10.1145/1238844.1238856

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

[16] Kernighan, B., Ritchie, D.: The C Programming Language. Prentice Hall (1978)

[17] Kiselyov, O., Ishii, H.: Freer monads, more extensible effects. In: Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015. pp. 94–105 (2015), http://doi.acm.org/10.1145/2804302.2804319

[18] Koh, N., Li, Y., Li, Y., Xia, L., Beringer, L., Honoré, W., Mansky, W., Pierce, B.C., Zdancewic, S.: From C to interaction trees: specifying, verifying, and testing a networked server. In: Mahboubi, A., Myreen, M.O. (eds.) Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019. pp. 234–248. ACM (2019), https://doi.org/10.1145/3293880.3294106

[19] Lampropoulos, L., Pierce, B.C.: QuickChick: Property-Based Testing in Coq. Software Foundations series, volume 4, Electronic textbook (May 2021), version 1.2. https://softwarefoundations.cis.upenn.edu/qc-1.2/

[20] Letan, T., Régis-Gianas, Y.: Freespec: specifying, verifying, and executing impure computations in coq. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020. pp. 32–46. ACM (2020), https://doi.org/10.1145/3372885.3373812

[21] Letan, T., Régis-Gianas, Y., Chifflier, P., Hiet, G.: Modular verification of programs with effects and effects handlers. Formal Aspects Comput. 33(1), 127–150 (2021), https://doi.org/10.1007/s00165-020-00523-2

[22] Liang, S., Hudak, P., Jones, M.P.: Monad transformers and modular interpreters. In: Cytron, R.K., Lee, P. (eds.) Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995. pp. 333–343. ACM Press (1995), https://doi.org/10.1145/199448.199528

[23] Lindley, S., Wadler, P., Yallop, J.: Idioms are oblivious, arrows are meticulous, monads are promiscuous. Electron. Notes Theor. Comput. Sci. 229(5), 97–117 (2011), https://doi.org/10.1016/j.entcs.2011.02.018

[24] Lorch, J.R., Chen, Y., Kapritsos, M., Parno, B., Qadeer, S., Sharma, U., Wilcox, J.R., Zhao, X.: Armada: low-effort verification of high-performance concurrent programs. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 197–210. ACM (2020), https://doi.org/10.1145/3385412.3385971

[25] Maillard, K., Ahman, D., Atkey, R., Martínez, G., Hritcu, C., Rivas, E., Tanter, É.: Dijkstra monads for all. Proc. ACM Program. Lang. 3(ICFP), 104:1–104:29 (2019), https://doi.org/10.1145/3341708

[26] Marlow, S., Brandy, L., Coens, J., Purdy, J.: There is no fork: an abstraction for efficient, concurrent, and concise data access. In: Jeuring, J., Chakravarty, M.M.T. (eds.) Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014. pp. 325–337. ACM (2014), https://doi.org/10.1145/2628136.2628144

[27] Marlow, S., Jones, S.P., Kmett, E., Mokhov, A.: Desugaring haskell's donotation into applicative operations. In: Mainland, G. (ed.) Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016. pp. 92–104. ACM (2016), https://doi.org/10.1145/2976002.2976007

[28] Coq development team: The Coq proof assistant (2021), http://coq.inria.fr, version 8.13.2

[29] McBride, C.: Turing-completeness totally free. In: Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings. pp. 257–275 (2015), https://doi.org/10.1007/978-3-319-19797-5_13

[30] McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. 18(1), 1–13 (2008), https://doi.org/10.1017/S0956796807006326

[31] Milewski, B.: Free monoidal functors, categorically! (May 2018), https://bartoszmilewski.com/2018/05/16/free-monoidal-functors-categorically/, blog post

[32] Moggi, E.: Notions of computation and monads. Inf. Comput. 93(1), 55–92 (1991), https://doi.org/10.1016/0890-5401(91)90052-4

[33] Mokhov, A.: Implementation of selective applicative functors in haskell (2019), https://hackage.haskell.org/package/selective

[34] Mokhov, A., Lukyanov, G., Marlow, S., Dimino, J.: Selective applicative functors. Proc. ACM Program. Lang. 3(ICFP), 90:1–90:29 (2019), https://doi.org/10.1145/3341694

[35] Mokhov, A., Mitchell, N., Jones, S.P.: Build systems à la carte: Theory and practice. J. Funct. Program. 30, e11 (2020), https://doi.org/10.1017/S0956796820000088

[36] Nigron, P., Dagand, P.: Reaching for the star: Tale of a monad in coq. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference). LIPIcs, vol. 193, pp. 29:1–29:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021), https://doi.org/10.4230/LIPIcs.ITP.2021.29

[37] Pai, V.S., Druschel, P., Zwaenepoel, W.: Flash: An efficient and portable web server. In: Proceedings of the 1999 USENIX Annual Technical Conference, June 6-11, 1999, Monterey, California, USA. pp. 199–212. USENIX (1999), http://www.usenix.org/events/usenix99/full_papers/pai/pai.pdf

[38] Pierce, B.C., de Amorim, A.A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., Yorgey, B.: Logical Foundations. Software Foundations series, volume 1, Electronic textbook (Aug 2021), version 6.1. http://www.cis.upenn.edu/~bcpierce/sf

[39] Piróg, M., Gibbons, J.: The coinductive resumption monad. In: Jacobs, B., Silva, A., Staton, S. (eds.) Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA, June 12-15, 2014. Electronic Notes in Theoretical Computer Science, vol. 308, pp. 273–288. Elsevier (2014), https://doi.org/10.1016/j.entcs.2014.10.015

[40] d. S. Oliveira, B.C.: Modular visitor components. In: Drossopoulou, S. (ed.) ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5653, pp. 269–293. Springer (2009), https://doi.org/10.1007/978-3-642-03013-0_13

[41] Schäfer, S., Tebbi, T., Smolka, G.: Autosubst: Reasoning with de bruijn terms and parallel substitutions. In: Urban, C., Zhang, X. (eds.) Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9236, pp. 359–374. Springer (2015), https://doi.org/10.1007/978-3-319-22102-1_24

[42] Silver, L., Zdancewic, S.: Dijkstra monads forever: termination-sensitive specifications for interaction trees. Proc. ACM Program. Lang. 5(POPL), 1–28 (2021), https://doi.org/10.1145/3434307

[43] Sozeau, M., Mangin, C.: Equations reloaded: high-level dependently-typed functional programming and proving in coq. Proc. ACM Program. Lang. 3(ICFP), 86:1–86:29 (2019), https://doi.org/10.1145/3341690

[44] Spector-Zabusky, A., Breitner, J., Rizkallah, C., Weirich, S.: Total Haskell is reasonable Coq. In: Andronick, J., Felty, A.P. (eds.) Proceedings of the

7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018. pp. 14–27. ACM (2018), https://doi.org/10.1145/3167092

[45] Swamy, N., Rastogi, A., Fromherz, A., Merigoux, D., Ahman, D., Martínez, G.: Steelcore: an extensible concurrent separation logic for effectful dependently typed programs. Proc. ACM Program. Lang. 4(ICFP), 121:1–121:30 (2020), https://doi.org/10.1145/3409003

[46] Swierstra, S.D., Duponcheel, L.: Deterministic, error-correcting combinator parsers. In: Launchbury, J., Meijer, E., Sheard, T. (eds.) Advanced Functional Programming, Second International School, Olympia, WA, USA, August 26-30, 1996, Tutorial Text. Lecture Notes in Computer Science, vol. 1129, pp. 184–207. Springer (1996), https://doi.org/10.1007/3-540-61628-4_7

[47] Swierstra, W., Baanen, T.: A predicate transformer semantics for effects (functional pearl). Proc. ACM Program. Lang. 3(ICFP), 103:1–103:26 (2019), https://doi.org/10.1145/3341707

[48] Wadler, P.: Recursive types for free! (July 1990), http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt, draft

[49] Wadler, P.: Comprehending monads. Math. Struct. Comput. Sci. 2(4), 461–493 (1992), https://doi.org/10.1017/S0960129500001560

[50] Wadler, P.: The expression problem (November 1998), http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, email correspondence

[51] Xia, L.: Free applicative functors in Coq (July 2019), https://blog.poisson.chat/posts/2019-07-14-free-applicative-functors.html, blog post

[52] Xia, L., Zakowski, Y., He, P., Hur, C., Malecha, G., Pierce, B.C., Zdancewic, S.: Interaction trees: representing recursive and impure programs in coq. Proc. ACM Program. Lang. 4(POPL), 51:1–51:32 (2020), https://doi.org/10.1145/3371119

[53] Zakowski, Y., Beck, C., Yoon, I., Zaichuk, I., Zaliva, V., Zdancewic, S.: Modular, compositional, and executable formal semantics for LLVM IR. Proc. ACM Program. Lang. 5(ICFP), 1–30 (2021), https://doi.org/10.1145/3473572

[54] Zakowski, Y., He, P., Hur, C., Zdancewic, S.: An equational theory for weak bisimulation via generalized parameterized coinduction. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020. pp. 71–84. ACM (2020), https://doi.org/10.1145/3372885.3373813