Formal Reasoning About Layered Monadic Interpreters

IRENE YOON, YANNICK ZAKOWSKI, and STEVE ZDANCEWIC

Monadic computations built by interpreting, or “handling,” operations of a free monad, are a compelling formalism for modeling language semantics and defining the behaviors of effectful systems, especially in dependent type theories such as Coq’s. The resulting layered semantics offer the promise of modular reasoning principles based on the equational theory of the underlying monads. However, there are a number of obstacles to using such layered interpreters in practice. With more layers comes more boilerplate and glue code needed to define the monads and interpreters involved. That overhead is compounded by the need to define (and justify) the relational reasoning principles that characterize the equivalences at each layer.

This paper addresses these problems by significantly extending the capabilities of the Coq interaction trees (ITrees) library, which supports layered monadic interpreters. We characterize a rich class of “interpretable monads”—obtained by applying monad transformers to ITrees—and show how to generically lift interpreters through them. We also introduce a corresponding framework for relational reasoning about “equivalence of monads up to a relation $R$.” This collection of typeclasses, instances, new reasoning principles, and tactics greatly generalizes the existing theory of the ITrees library, eliminating large amounts of unwieldy boilerplate code and dramatically simplifying proofs.

ACM Reference Format:

1 INTRODUCTION

Since their inception, monads and monadic interpreters have been recognized as appealing and mathematically elegant ways to define programs and their semantics, especially in the presence of I/O, state, nondeterminism, failure, or other effects [24, 25, 30, 41]. The monad laws, suitably extended with domain-specific equations that capture the semantics of effects, enable reasoning about the equivalence of monadic programs, and, more generally, yield powerful relational program logics (such as Dijkstra monads [2, 26, 27]) that can be used to prove properties ranging from the correctness of program optimizations to information-flow noninterference [8].

It is no surprise, then, that when it comes to formalizing the behavior of complex language semantics or the behavior of interactive systems, monads play a crucial rôle. They are particularly well suited for defining the semantics of effects when the metalanguage is pure and total, which is the case when embedding a language semantics into dependent type theory, such as that found in Coq. Moreover, by working with free monads [43] and monadic interpreters, one can obtain a flexible, general-purpose reasoning framework for effectful computations. Variations of this idea have appeared throughout the literature, for instance as the program monad in the FreeSpec project [23], as I/O-trees [14], and as McBride’s general monad [28].

In this paper, we focus on interaction trees [46] (ITrees) a recent realization of this approach as a Coq library. ITrees are defined as a coinductive variant of the freer monad [20] and are also closely

Authors’ address: Irene Yoon; Yannick Zakowski; Steve Zdancewic.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery.
XXXX-XXXX/2022/5-ART $15.00
https://doi.org/10.1145/nnnnnn.nnnnnnn

, Vol. 1, No. 1, Article . Publication date: May 2022.
related to *resumption monads* [35]. The core data structure, \( \text{itree} \ E \), represents a computation as a possibly-infinite tree with nodes labeled by events \( e \) drawn from an event signature \( E \). An event \( e \) can be thought of as a point at which the computation interacts with its environment, allowing the environment to supply a response \( r \). The node corresponding to \( e \) in the ITree also has the continuation of the computation, given as a function of \( r \).

Several other features of ITrees make them well-suited to defining semantics. First, event signatures can be combined through a disjoint-union operation: an ITree of type \( \text{itree} \ (E + F) \) can have events drawn from either \( E \) or \( F \), and the algebra of event signatures induces structure on the monad computations. Second, it is possible to write interpreters for the events in an ITree. We write \( h : E \leadsto M \) to mean that \( h \) is an *event handler* that maps events in the signature \( E \) into computations in the monad \( M \). Then \( \text{interp} \ h : \forall A, \text{itree} \ E A \rightarrow M A \) maps an ITree computation into a monadic computation by “handling” the events in \( E \), which gives them a semantic definition according to the operations of \( M \). For instance, one might interpret “read” and “write” events into a state monad. Taken together, the algebra of events and event handlers allows for the decomposition of effects into stages, modularizing the semantics in a style akin to *algebraic effects* [36]. Third, by implementing the \( \text{itree} \ E \) monad coinductively, divergence is given native support, allowing for generic fixed-point combinators to be defined without compromising the (lazy) computability of the resulting definitional interpreters. Indeed the type \( \text{itree} \ \text{void}E \) (where \( \text{void}E \) is the empty event signature, which is also the unit of \( + \)') is isomorphic to Capretta’s *delay monad* [10], which was designed specifically to represent nonterminating computations in type theory. Finally—and perhaps most importantly for the purposes of formal verification—ITrees come equipped with a rich relational theory: \( t \approx_R u \) means that ITrees \( t \) and \( u \) are *weakly bisimilar* and produce answers related by \( R \). This relational theory generalizes the notion of program equivalence (just take \( R \) to be \( = \)), and, importantly, it can be lifted to the other monads obtained by interpretation of ITrees, yielding a way to reason formally about monadic semantics.

Past experience suggests that ITrees provide a very comfortable semantic toolbox, whether to reason about the functional correctness of programs or the correctness—in terms of equivalence or refinement—of program transformations. The original paper [46] illustrated the approach by proving the correctness of a toy compiler from \texttt{imp}, an imperative language, to \texttt{asm}, a simplified assembly language. The story is pleasantly clean: a termination-sensitive result is established without the need for any explicit coinduction; the compiler is proved in a compositional fashion (open pieces of programs can be related to their compilation in isolation); and the proof method is heavily equational, relying on \( \approx_R \) to allow tedious but easy-to-produce proofs by rewriting.

The ITrees approach has also been shown to scale. The most ambitious project in this realm, to date, is Vellvm [47]. In this project, the semantics of the sequential fragment of LLVM IR has been formalized using ITrees, leading to a remarkably simplified semantics when contrasted with the previous iteration of the project [49], which was based on a more traditional operational semantics. In particular, the program counter, and the heavy invariants it entails, have disappeared. When it comes to reasoning, the benefits of the ITrees semantics have been evaluated in a couple of dimensions. With respect to LLVM IR transformations, simple peep-hole optimizations admit local, simple proofs, and proving correct a block-fusion transformation can be done at a high level of abstraction—verifying a transformation that impacts only control-flow does not depend on the implementation of the state. With respect to front-ends targeting LLVM IR, the HELIX [48] project formalizes parts of a system called SPIRAL, and it compiles down to Vellvm.

While the results above are a promising testament to the viability of ITrees-based monadic interpreters for formal verification, things do not remain as smooth at the scale of Vellvm and HELIX as they were for \texttt{imp} and \texttt{asm} (and even at the small scale, the situation can be improved). In
this paper, we identify—and provide solutions to!—the following pain-points that are impediments to using ITrees at scale:

(1) As mentioned above, the primary feature of the free monad is its extensibility: there is a natural inclusion of type itree $E$ into itree $(E + F)$ with an event signature enriched via a coproduct. That inclusion necessitates a renaming (to add the left injection), and such renamings can be discharged with a simple typeclass (provided by the ITrees library). However, when it comes to using interp, which handles all of the events of an ITree, the existing typeclasses are inadequate: a handler $h : E \sim M$ cannot readily be lifted to a handler $h' : (E + F) \sim M$ (and such a lifting may not always be possible), which means that the user has to hardcode the syntactic structure of the event signatures for handlers and interpreters, breaking modularity. Existing techniques, for instance the automatic injections used in Swierstra’s data types à la carte [43] (and re-implemented in the ITrees library), do not provide an adequate solution.

(2) When modularly structuring a semantics as complex as Vellvm’s, interpretation takes place in layers: several interpreters are successively composed, each handling different events. However, while free monads, and ITrees in particular, give the freedom to interpret into any monad, we are now left to ponder how to interpret from other structures. For example, given a handler $h : E \sim \text{stateT} \ (\text{itree} F)$ that interprets $E$ events into the monad \text{stateT} \ (\text{itree} F), where \text{stateT} is the state monad transformer, the type of interp $h$ is itree $E \sim \text{stateT} \ (\text{itree} F)$. If we have another handler $f : F \sim M$, we would like to compose the interpreters: $(\text{interp} \ f) \circ (\text{interp} \ h)$ should have type itree $E \sim \text{stateT} \ M$, but the left-hand use of interp works over a state-transformed ITree, not an ITree, so this code does not typecheck—we require a much more general notion of “interpreter” to build such interpretation stacks. With the existing ITrees library, such compositions must be constructed painfully and repetitiously by hand.

(3) Beyond these issues that arise when building complex interpreter stacks, we also need to be able to reason about the resulting computations. Consequently, we need versions of the relational theories (monad laws, etc.) for every monad in sight! In practice, this means that we need mechanisms to lift the equational theory of one monad through a monad transformer to obtain a transformed theory. It is not enough to be able to construct proofs of equivalences or refinements, we also need inversion principles to extract information from such proofs. It is not at all obvious how to build such a relational reasoning framework generically.

Contributions. In this paper, we propose solutions to the problems described above. After giving the necessary background about the ITrees library and its current pain-points in Section 2, we tackle the problem of building layered monadic interpreters.

Section 3 introduces several novel typeclasses: Trigger, Subevent, and Interp, along with a generic operation called over, that collectively address the issues with modularity of event signatures and construction of layered interpreters. Along the way, we see how to define instances of our new typeclasses for a variety of frequently-used monads: itree $E$, state, error, Prop.

Section 4 develops new tools for relational reasoning, centered on a typeclass $\text{eqmR}$ (for “equivalence of monads up to $R$”)—a generalization of ITrees week bisimulation, $\approx_R$. We identify a suitable axiomatization of its properties and define operations that lift it through monad transformers. A key, and, we believe, novel idea here is the concept of the image of a monadic computation, which precisely characterizes its possible results. The image is defined purely in terms of $\text{eqmR}$, and it is a key ingredient needed to define the equational theory.

Section 5 uses $\text{eqmR}$ to define the properties of the new typeclasses introduced in Section 3, providing a rich framework for reasoning about layered monadic computations.

Section 6 describes how this framework pans out in practice, where a key contribution is a collection of tactics that provide type instantiations that help disambiguate typeclass resolution.
The datatype takes as its first parameter a signature—described as a family of types $E : \text{Type} \to \text{Type}$—that specifies the set of interactions the computation may have with the environment. The $\text{Vis}$ constructor builds a node in the tree representing such an interaction, followed by a continuation indexed by the return type of the event. The second parameter, $R$, is

$$E \rightsquigarrow F := (\forall X, E X \to F X).$$

(* Embedding of pure computations *)

**Definition** ret ($E : \text{Type} \to \text{Type}$) ($R : \text{Type}$) ($v : R$) : itree $E$ $R$.

(* Sequencing computations *)

**Definition** bind ($E : \text{Type} \to \text{Type}$) ($T U : \text{Type}$) ($u : \text{itree} E T$) ($k : T \to \text{itree} E U$) : itree $E$ $U$.

(* Atomic itrees triggering a single event. *)

**Definition** trigger ($E : \text{Type} \to \text{Type}$) : $E \rightsquigarrow \text{itree} E$.

(* Fixed-point combinator *)

**Definition** iter ($E : \text{Type} \to \text{Type}$) ($R I : \text{Type}$) ($\text{body} : I \to \text{itree} E (I + R)$) : $I \to \text{itree} E R$.

Fig. 1. Interaction trees: definition and type signature of its main combinators

We evaluate the effectiveness of this new infrastructure by porting the IMP-to-ASM proofs from the original ITrees development—we find that the resulting proofs are substantially less ad hoc, more compositional, and considerably simpler.

Section 7 situates our contributions with respect to related work, and Section 8 concludes with a discussion about further techniques and future work.

**Implementation.** The ideas in this paper are packaged as a Coq development (which will eventually be merged with the ITrees library) and all of the properties presented here have been proved in Coq. Although some of our contributions are Coq-specific (e.g., the need to deal with Proper instances for Coq’s setoid rewriting and the details of our tactics), we believe that most of the typeclasses and constructs proposed here could be profitably implemented in other settings as well.
the result type, the type of values that the computation may return if it halts. The constructor `Ret` builds such a pure computation, represented as a leaf. Finally, the `Tau` constructor models an internal, non-observable step of computation, allowing the representation of silently diverging computations; `Tau` is also used for guarding corecursive definitions.

ITrees are equipped with four main primitive combinators. As expected, `itree E` forms a monad at any signature `E`: pure computations can be embedded with `ret`, and computations can be sequenced with `bind`. The `trigger e` combinator builds the minimal computation performing the event `e`, which immediately returns the answer from the environment. Finally, ITrees support fixed-point combinators such as `iter` which encodes terminal recursion.

To illustrate how to model computations with ITrees, consider a signature describing printing on one hand, and interaction with a single memory cell storing a natural number on the other. The cell can be read or updated, and values can be sent to the external printer: notice how each event specifies the nature of the answer it expects from the environment in the index type.

```
Variant printE :=
    | Print : nat → printE unit.

Variant cellE :=
    | Get : cellE nat
    | Put : nat → cellE unit.
```

A computation that writes the value 3 to the cell, reads the content of the cell, and prints it to `stdout` can be represented as follows:

```
_ ← trigger (Put 3); x ← trigger Get;; trigger (Print x)
```

This computation has type `itree (printE + stateE) unit`, where `+` is the disjoint sum operator.

ITrees are an implementation of the freer monad with a coinductive model of divergence. The events contained in a tree are uninterpreted; they assume no predetermined semantics. For instance, the traditional algebraic law ensuring that the `Get` operation in the previous example should return 3 is not accounted for at this stage. Such semantics of the effects manipulated is given in a separate step that enriches the structure by interpreting events into appropriate monads.

The notion of equivalence of computations over interaction trees (before interpretation) is a weak bisimulation observing the uninterpreted events and the returned values. This relation is referred to as equivalence up-to taus, or `eutt` for short, and ensures co-termination and trace equivalence. Congruence, monadic, and iterative laws are proved with respect to `eutt`.

The iterative laws used in ITrees can be also generalized for any arbitrary monads, since it corresponds to Kleisli arrows (i.e. functions of type `A → M B` given a monad `M`) forming an iterative category. We call any such monad an iterative monad.

### 2.2 Monadic Implementation of Effects

The modularity of ITree-based semantics is embodied by the `interp` function. Through `interp`, a handler, which maps events into an iterative monad, can be freely lifted to a whole tree, essentially folding over the tree to produce a monadic computation.

To illustrate this idea, consider a handler that implements the memory cell events via a state monad, while leaving print events as uninterpreted.

```
Definition handle_cell : printE + stateT nat (itree printE) :=
  fun _ e n ⇒ match e with
    | inl1 (Print x) ⇒ trigger Print x;; Ret (m, tt)
    | inr1 Get ⇒ Ret (n, n)
    | inr1 (Put m) ⇒ Ret (m, tt)
```

3In Section 3, we introduce a more general version of `trigger`, and the overloading is handled by module namespaces (i.e. this ITree-specific trigger will be referred as `ITree.trigger`)

4We write `(x ← t;; k)` and `t ≫= k` as notations for `(bind t (fun x ⇒ k))` and `(bind t k).`
This handler gives a semantics to `printE` and `cellE` events through pattern matching on the sum type. Such handlers can then be lifted by `interp`.

Definition `interp_cell : itree (printE +' cellE) ~ stateT nat (itree printE) := interp handle_cell.`

### 2.3 Scaling Up: The Shortcomings of Layered Monadic Interpreters

This promise of modularity is deceitful when used at scale: layering monadic interpreters becomes unwieldy when scaling. Let’s look at an example of a realistic semantics to see what can go wrong—for instance, the Vellvm project [47] uses ITrees to formalize a large sequential subset of LLVM IR, an industrial-strength intermediate representation.

When dealing with large languages, the naïve interpretation scheme sketched above, which interprets all of its events at once, is undesirable. For one thing, some effects may be implemented in terms of others: memory operations, for instance, may introduce undefined behavior events. For another, decoupling the interpretation of different categories of events modularizes the monadic structure they introduce, improving the modularity of the semantics, and hence the robustness of the formalization. Such decoupling also leads to proof techniques allowing for some of the effects to remain uninterpreted during a proof—Zakowski, et al. [47] illustrate this idea by proving a simple block fusion optimization correct independently of the implementation of the memory model.

Thus, it is desirable that complex monadic interpreters be organized as layers of interpretation. Figure 2 reproduces the structure of Vellvm’s interpreter: a piece of LLVM IR syntax is first represented as an ITree over a rich signature before its effects are successively implemented, leading to a richer monadic structure at each layer. Intuitively, this sequence of interpreters implement: (1) calls to pure intrinsics interpreted as pure Coq functions, (2) the global state interpretation, (3) the local state interpretation, (4) the memory model interpretation, (5) the non-deterministic concretization of `undef` values, and (6) the non-deterministic refinement of undefined behaviors.

However, there are two glaring issues for both defining and working with layered interpreters, which are presented as follows.

#### 2.3.1 Problem 1: Lifting Partial Handlers for Whole Signatures

The first difficulty is defining a handler for partial interpretations, i.e., interpreting a particular effect out of a sum of events while leaving the others uninterpreted. The toy example that interprets away the `CellE` signature while preserving `printE` should be factored into a part that handles `CellE` in isolation and a generic part that injects the remaining events.

The ITree library provides some applicable tools. Figure 3 reproduces the standard interpreter for memory events, `stateE`, where `S` is the notion of state considered and `stateT S` is the state monad transformer defined as `fun M R ⇒ S → M (S * R)`. To be reusable, the `stateE` handler corresponding to our example `cellE` handler is defined in `h_state`, which is parametric in the leftover ambient
Definition \( h_{\text{state}} \) : 
\[ \text{state}E \leadsto \text{state}T \text{ S \ (itree } E) : \]
\[ \text{fun \ } _e s \Rightarrow \text{match } e \text{ with } \]
\[ | \text{Get} \Rightarrow \text{Ret } (s, s) \]
\[ | \text{Put } s' \Rightarrow \text{Ret } (s', \text{tt}) \]
\[ \text{end}. \]

Definition \( \text{pure}_{\text{state}} \) : 
\[ \text{E \leadsto \text{state}T \text{ S \ (itree } E) : \]
\[ \text{fun \ } _e s \Rightarrow \text{Vis } e \ (\text{fun } x \Rightarrow \text{Ret } (s, x)). \]

Definition \( \text{interp}_{\text{state}} \) : 
\[ \text{itree } (\text{state}E +' \text{ } E) \leadsto \text{state}T \text{ S \ (itree } E) : \]
\[ \text{interp } (\text{case}_h\text{state } \text{pure}_{\text{state}}). \]

**Fig. 3. State interpreter from the ITree library**

Definition \( \text{handle}_{\text{local}} \) : 
\[ \text{E } \leadsto \text{state}T \text{ map \ (itree } E) : \]
\[ \text{fun \ } _e \text{ env } \Rightarrow \]
\[ \text{match } e \text{ with } \]
\[ | \text{LocalWrite } k \text{ v } \Rightarrow \text{ret } (\text{add } k \text{ v } \text{env, tt}) \]
\[ | \text{LocalRead } k \Rightarrow \]
\[ \text{match } \text{lookup } k \text{ env } \text{ with } \]
\[ | \text{Some } v \Rightarrow \text{Ret } (\text{env, v}) \]
\[ | \text{None } \Rightarrow \text{fail} \]
\[ \text{end}. \]

Variable \((E \ F \ G \ H: \text{Type} \rightarrow \text{Type})\).
Context `(\text{Failure}_E < G)\).
Notation \( \text{Effin} := (E +' F +' \text{LocalE} +' G)\).
Notation \( \text{Effout} := (E +' F +' G)\).

**Fig. 4. Interpreting Vellvm’s register map**

The branch implementing \( \text{printE} \) is captured generically in \( \text{pure}_{\text{state}} \), since the implementation does not depend on the effect that remains uninterpreted. Finally, since \(+'\) forms a coproduct for an indexed function, a generic \text{case} combinator builds the handler used to interpret an arbitrary computation containing \( \text{state}E \) events, as shown in \( \text{interp}_{\text{state}} \).

So surely we should be able to happily simplify the definition of \( \text{handle}_{\text{cell}} \) by using the standard library’s \( \text{state}E \) events instead of specialized \( \text{cellE} \), and directly defining \( \text{interp}_{\text{cell}} \) as \( \text{interp}_{\text{state}} \)? Unfortunately, we cannot! A slight mismatch creeps in: our previous computations have been defined over the \( \text{printE} +' \text{ cellE} \) signature, while \( \text{interp}_{\text{state}} \) forces \( \text{state}E \) to be in the head position: \( \text{state}E +' \text{ printE} \). We are hence forced to either change our definitions to line up the signatures, or to duplicate the definitions.

These structural constraints add up to create bureaucratic clutter in large-scale developments. Figure 4 reproduces the Vellvm interpreter layer implementing the register map—this interpreter defines the translation from the Global env level to the Local env level of Figure 2. With this set up, events in the register map, defined in the LocalE signature, are structurally in the third position of the signature (see Effin) at the site where the interpreter of this handler is used, and three auxiliary definitions for triggering \( E, F \), and \( G \) events along with fiddly uses of \text{case} are needed to define \( \text{interp}_{\text{local}}\).

Also, notice the constraint \text{Failure} -< G in the context, which morally represents "any event \( G \) that supports failure". This corresponds to the constraint mechanism introduced by Sweistra’s Data Types à la Carte [43] to automate the renaming of triggered events. We discuss its limitations and introduce a more expressive substitute in Section 3.3.
Simply reorganizing the shape of the signature is impossible due to constraints that do not compose in various places of the semantics, forcing developers to painstakingly handcraft special-case interpreters. Naturally, this definition is very fragile to the introduction of additional effects, in blatant contradiction with the modularity otherwise achieved.

2.3.2 Problem 2: Building Layered Interpreters. The second issue in building layered interpreters is the complete lack of a theory for interpreting computations from monads other than ITree. The existing interp implementation is parametric in its target monad, but it is not in its source: interp (h : E ~ M) : itree E ~ M is defined for any iterative monad M.

Sticking to the same implementation of the register map, we see on Figure 2 that it occurs after the implementation of the global state. As a consequence, the domain of computation manipulated at this stage is already not a plain ITree, but rather stateT EnvG (itree E1) for some signature E1. This problem reoccurs at each subsequent level.

The previous approach to this issue is defining ad hoc solutions for each situation. Indeed, one can “interpret” a computation in stateT EnvG (itree E) by exposing the definition of the stateT transformer as a pure function of the initial state, and therefore interpreting the computation point-wise. While it certainly works, the status of these constructions becomes at best murky, leading to the necessary breaking of the abstractions introduced by this layered structure.

In Section 3, we introduce the appropriate abstract structures necessary for the principled construction of layered interpreters, providing a general and clean solution to both problems from the programmatic perspective. Of course, defining monadic computations in this modular, layered way is only half of the problem. For formalization, we also need to develop the corresponding metatheory for reasoning equationally about these constructions. That is the subject of Section 4.

3 BUILDING LAYERED MONADIC INTERPRETERS

In this section, we introduce (1) a novel over combinator for lifting partial handlers for whole signatures, (2) a general characterization of the trigger combinator, which interplays with a novel interp combinator for building layered interpreters. In addition, we propose a new kind of event constraint which characterizes isomorphisms between sums of events, where we use typeclass resolution to infer the correct type-injections for the over and trigger combinators.

The interpreter from Figure 3 is unsatisfactory because of the need for manual injection. To simplify this interpreter, we define a generic function that injects h_state into an arbitrary signature containing stateE. There are two main challenges in defining this automatic injection.

First, injecting the h_state handler induces auxiliary handlers, such as pure_state, that perform no action on the events: these auxiliary handlers must be inferred and applied to the uninterpreted remainder of the sum. Second, when the injected handler h_state acts on a signature containing stateE (where the signature may contain other events), it needs to return the remainder of the signature. This is trivially achieved when hard-coding the shape of the signature as stateE +′ E, but cannot be captured by the current inclusion constraint stateE -< F.

These challenges motivate respectively the definition of triggerable monads (Section 3.1) and the generalization of the inclusion of signature into a decomposition of signatures (Section 3.2). These two building blocks are sufficient to define the automatic injection of handlers addressing Problem 1 (Section 3.3). Problem 2 finds its resolution by the additional introduction of Interpretable monads (Section 3.4). While motivated by concrete problems, over, interp, and trigger also form a cohesive equational theory: Section 4 and 5 describe the equational properties of these combinators.
3.1 Triggerable Monads

Recall the trigger combinator from Section 2. It is defined as \( \text{trigger } e := \text{Vis } e \ (\text{fun } x \Rightarrow x) \), capturing the idea of a “minimal” impure computation performing an uninterpreted event—in this case, specialized to the ITrees monad. The \( \text{pure_state} \) function from Figure 3 mirrors this intent, but in the monad \( \text{stateT } S \ (\text{itree } E) \); it additionally makes explicit that this minimal computation does not affect the state.

We hence capture this notion into a Trigger typeclass, corresponding to an action(event) having a specific monad \( M \) as its domain of action.

\[
\text{Class Trigger } (\text{E: Type } \rightarrow \text{Type}) \ (\text{M: Type } \rightarrow \text{Type}) := \text{trigger: E } \sim M.
\]

On the implementation side, it is worth mentioning that this trigger operator does not explicitly mention the monadic structure of \( M \). This is inspired by the “unbundled” approach of Spitters and van der Weegen [40], that proves beneficial for mathematical formalizations in type theory. All typeclasses in our framework use this unbundled style.

Naturally, \( \text{ITree.trigger} \) is an instance for \( \text{itree E} \), and \( \text{pure_state} \) for \( \text{stateT } S \ (\text{itree } E) \). However, it is possible to capture a broader class of instances at once, as we will see shortly.

Since we also want to reason about the structures we introduce, this new definition raises the question of the axiomatization of the trigger operation. In the case of ITrees, there are two characteristic equations supported by trigger. First, when seen as a handler, its interpretation is the identity, i.e., \( \forall t, \text{interp trigger } t \approx t \). Second, when seen as an interpreted computation, it coincides with the effect of the handler on the event, i.e. \( \forall h e, \text{interp } h \ (\text{trigger } e) \approx h \ e \). At this point, we lack the tools to generalize these equations; in particular, we would need to be able to interpret a computation in the monad of interest, rather than an itree specifically. Therefore, we delay the question of axiomatizing these properties until Section 5.

3.2 Automatic Injection and Decomposition of Signatures

Let us temporarily set aside triggerable monads to turn our attention to the second issue: how to remove the hard-coded shape of the source event signature, yet reconcile that with a requirement that the target signature removes the handled event.

We achieve this by first enriching the typeclass responsible for expressing that a signature is a super-set of another. The current constraint, \( E \prec F \), simply requires an embedding of \( E \) into \( F \). Instead, we introduce a typeclass that explicitly computes the complement to \( E \) in \( F \).

The resulting relation is therefore three-placed: the constraint \( \text{Subevent } E \ F \ G \), written \( E +? G \prec F \), provides an isomorphism between the types \( E +' G \) and \( F \):

\[
\text{Class Subevent } \{E \ F \ G : \text{Type } \rightarrow \text{Type}\} : \text{Type} := \{ \text{split : } F \sim E +' G ; \\
\text{merge : } E +' G \sim F \}.
\]

From a resolution standpoint, one should think of it as taking as input the ambient signature \( F \)—from the return type of the computation being built—and the sub-signature \( E \)—from the concrete object manipulated, typically a triggered event or a handler—and inferring from this information the complement \( G \).

\[
\text{Instance Subevent_Base } (A \ B) : A +? B \prec A +' B. \\
\text{Instance Subevent_refl } (A) : A +? \text{void1} \prec A. \\
\text{Instance Subevent_Sum_In } (A \ B \ C \ D) \\
\ (A +? D \prec B) \ : (C +' A) +? D \prec C +' B. \\
\text{Instance Subevent_Sum_Out } (A \ B \ C \ D) \\
\ (A +? D \prec B) \ : A +? C +' D \prec C +' B. \\
\text{Instance Subevent_commute } (A \ B \ C) \\
\ (\text{Sub: } A +? B \prec C) \ : B +? A \prec C.
\]

Instance inference gets more involved that with Swierstra’s injection, especially when combining several constraints. We characterize to this end the algebraic properties of this abstract sum operation \( +? \) as instances. Essentially, we state that it extends \( +' \), admits \text{void1} as a unit, allows

\[\text{Vol. 1, No. 1, Article . Publication date: May 2022.}\]
Instance Trigger_ITree_base {E} : Trigger E (itree E) := fun _ e ⇒ trigger e.
Instance Trigger_ITree {E F G} `(E +? F < G): Trigger E (itree G) :=
  fun _ e ⇒ trigger (merge (inl1 e)).
Instance Trigger_MonadT {E F G} `(T : (Type → Type) → Type → Type) (T_MonadT: MonadT T) : Trigger E (T (itree G)) :=
  fun _ e ⇒ lift (trigger e).

Fig. 5. The trigger typeclass

Definition interp_state {E F} `(stateE +? E < F): itree F; stateT S (itree E) :=
  interp (over h_state).

Definition interp_local {E1 E2 F} `(LocalE +? E1 < F) `(FailureE +? E2 < E1):
  itree F; stateT map (itree E1) := interp (over handle_local).

Fig. 6. State interpreter and Vellvm’s register map interpreter using over

for injections of ‘+’ on either side of the decomposition, and commutes. We constraint the use of these instances — commutation in particular — to prevent the inference mechanism to diverge: we refer the interested reader to our formal development. Each of these instances come with a proof of isomorphism, thus guaranteeing soundness of the inference.

With this definition of Subevent in place, we can use it to generically define, once and for all, Trigger instances as shown in Figure 5. Event of type E can be triggered into ITrees either at the same signature, or an extension of it. Moreover, rather than painfully (and manually) introducing ad hoc instances such as pure_state, we can transport Trigger instances through arbitrary monad transformers. The lift operator is defined on all monad transformers T : (Type → Type) → Type → Type, where lift transforms a monad M : Type → Type into a monad T M.

3.3 Automatic Injection for Handlers

We now have all the necessary tools to properly solve Problem 1 by injecting handlers of a restricted signature into a larger ambient one.

Consider a handler h : E ↞ M implementing a set of effects described by E into an arbitrary monad M. The function over h transports h into an implementation of a larger signature G into the same monad:

Definition over {E F G M : Type → Type} {S : E +? G < F} {T : Trigger G M} (h : E ↞ M) : F ↞ M :=
  fun t f ⇒ match split f with
  | inl1 e ⇒ h e
  | inr1 g ⇒ trigger g
  end.

The function relies on the constraint E +? G < F to know how to case analyze on a F event whether it corresponds to the embedding of an E event or not. In the former case, it simply calls its implementation h. In the latter case, it relies on the Trigger G M constraint to know how to embed this event into M.

Figure 6 illustrates the cleaned-up definitions for the ITree’s state interpreter and Vellvm’s register map implementation. The interp_state definition is straightforwardly simplified: no explicit extension of the handler is needed, and the return type uses the complement specified in the typeclass constraint. In the second case, notice that we can easily enforce that the source signature contains both LocalE and FailureE, while the target signature is only stripped of the former. As
intended, these interpreters can consequently be used regardless of the structural position of the interpreted signature in the ambient context. The equational theory of over is discussed in Section 5.

3.4 Interpretable Monads

We now focus on the second obstacle to the compositional definition of layered monadic interpreters: interpreting monadic structures other than pure ITrees. **Interpretable monads** must encompass monads built from ITree through layers of interpretation: they should essentially be iterative monads that may trigger events. Furthermore, interpretable monads must interpret into interpretable monads, as we aim to stack layers of interpretation. The typeclass we need to provide generic constructions and axiomatizations over the computational structure suitable to be part of a monadic interpreter more specifically generalizes the ITree.diag function.

```
Class Interp (IM T : (Type → Type) → Type → Type) (M : Type → Type) :=
interp : ∀(E : Type → Type) (h : E → M), T (IM E) → T M.
```

Intuitively, an instance of this class expresses that a structure akin to ITrees can lift handlers into a structure M. The shape of the source structure is, however, further specified. At its base, it should contain a family of monads IM indexed by signatures—itree is one example. Intuitively, this minimal structure is the one upon which the implementation of the effects will be lifted. To compose cleanly, a monad transformer T is assumed on the source, and preserved into the target: previously introduced effects are left untouched.\(^5\)

The interp function provided by the ITree library is a particular instance of this typeclass, where IM is itree, T is the identity transformer, and M is an arbitrary iterative monad.

With this definition, very generic instances can be provided to build layered interpreters compositionally. Following [18], we make explicit the higher-order functorial structure of monad transformers: they directly transport indexed functions via hfmap, as well as functions through any functor, per the operations shown below.

```
Class HFunctor (f : (Type → Type) → Type → Type) :=
{ ffmap : ∀A B g, Functor g → (A → B) → f g A → f g B;
  hfmap : ∀g h, (g ∘ h) → (f g ∘ f h) }.
```

This functorial structure is sufficient to ensure that monad transformers preserve the fact that a structure is a valid source of interpretation, as is captured by the following instance:

```
Instance stack_interp {T IM : (Type → Type) → Type → Type} {M : Type → Type}
{HFunctor T} {IterativeMonad M} {Interp IM Id M} : Interp IM T M :=
fun E h R t ⇒ hfmap (interp h) t.
```

Requiring in the Interp assumption the transformer parameter to be Id forces the stack to be built by adding new effects at the bottom, eliminating ambiguity when inferring types — alternate usages can be manually recovered. In practice, we work with a more specialized instance, fixing IM to be itree, to lighten up the unification problems arising when using these highly overlapping and ambiguous typeclasses. Combined with interp as a base instance, we can build interpreters from any structure built by applying monad transformers atop of the ITree monad, resolving Problem 2.

Through this section, we have introduced principled tools that clean up the definitions of layered stacks of interpreters. To do so, we have identified general structures whose particular instances were used in the ITrees library: triggerable monads, decomposition of signatures, and interpretable indexed monads. What remains to be defined is the infrastructure needed to reason about these definitions. In the following section, we start introducing their equational theory.

\(^{5}\)Because of the ‘unbundled’ approach, the structural constraints on this multi-parameter typeclass are not apparent.
4 A COMPOSABLE EQUATIONAL THEORY FOR MONADS

Monadic interpreters come with a seducing promise: equivalences or refinements of computations can be established equationally. Part of this reasoning naturally relies on the domain-specific algebraic laws that a given monad satisfies. But a significant structural equational theory—relevant to essentially all of the monads in our layered interpretations—is equally necessary. These theories can be painfully (re-)discovered and manually implemented for each monad, but that approach does not scale in practice.

To alleviate this problem, we provide through this Section a rich equational axiomatization of the monadic structures that arise from the construction of layered monadic interpreters. This theory both refines the one provided by ITrees, and generalizes it greatly, notably by axiomatizing the new constructions introduced in Section 3. We present in Section 4.4 and 5 how these theories can be cleanly transported by monad transformers and interpretation, lightening the burden put on a user when building their own layered interpreter.

More specifically, Section 4.1 introduces eqmR, the family of relations over monadic computations we consider—one can think of it as a generalization of eutt. Section 4.2 defines the image of a monadic computation, allowing for the definition of the enriched set of monadic laws we axiomatize and prove to hold for standard monads in Section 4.3. Finally, we describe in Section 4.4 the transport of these structure through monad transformers to ease once and for all the construction of the structures used in layered interpreters, before discussing cross-monad relations.

4.1 Equivalence and Relations Between Monadic Computations

A monad only deserves its name if it satisfies the well-known three monad laws. The ret operation should be a unit to the left of the bind (x ← ret a ;; k x = k a) and to its right (x ← ma ;; ret x = ma). The bind operation should additionally be associative (b ← (a ← ma ;; f a) ;; g b = a ← ma ;; (b ← f a ;; g b)).

This statement is however too naive, hiding a major difficulty: these equations have no hope to hold up-to equality. In the case of a coinductive structure such as ITrees, for instance, even eta-laws do not hold with respect to eq, Coq’s equality. One therefore needs to switch to a different notion of equivalence, namely, (strong) bisimulation, which is defined the the ITrees library as eq_itree.

While eq_itree can be used to prove the monad laws, it is still too strong for some iterative laws and interpretation laws; for these, we need weak bisimilarity, i.e., eutt. The conclusion is not surprising: monads should come equipped with their own notion of equivalence of computations.

But monadic interpreters are used to prove more than program equivalence. For instance, the original ITree paper [46] establishes the correctness of a compiler. This is achieved by parameterizing eutt by a relation on computed values: the monad specifies that weak bisimilarity is the prime notion to compare computations, but lifts an arbitrary relation on the computed values in the process. This extension not only allows for relating heterogeneous computations, but provides the foundations for establishing bisimilarity results following a relational program logic style [9].

We therefore work with monads M equipped with an “equality of monads up to R”, a family of relations: eqmR A B (R : A → B → Prop) : M A → M B → Prop. We write ≈R in lieu of eqmR R.

Figure 7 axiomatizes the required behavior for eqmR. Equivalences should be lifted into equivalences, ensuring, in particular, that ≈eq — written ≈ in the following — behaves as a suitable tightest notion of equality of computations. We break this transport the slightly stronger request that partial equivalence relations (PERs) and reflexivity be preserved independently: PERs are used to define the notion of an image as introduced in Section 4.2. The RelComp and Transpose rules express the standard heterogeneous extensions of transitivity and symmetry (we write †R for the transposition of R and ⊆ for inclusion of relations).

, Vol. 1, No. 1, Article . Publication date: May 2022.
We prove that this definition of $eqmR$ while sequencing threads the states. The indexed relation should therefore be monotone, providing a weakening rule, and ensuring $S \rightarrow \text{Error}.$

Example 4.3. Propagating erroneous states, as depicted to the left in the following:

errorM relation of state considered here is equality, but could be additionally relaxed to other equivalences. equivalence to functional extensionality, and further lifts the relation overt the returned values. The strong ($\text{right column in Figure 7}$ specifies the necessary introduction and elimination rules.

Finally, anticipating Section 4.4, we wish to transport our equational theories via monad transformers. Since examples such as the state and error transformers expand the return type of the rule, but the Conj rule must be additionally required.

We conclude this section by illustrating valid instances of $eqmR$ over concrete monads.

Example 4.1. ITree.

At any signature $E,$ $\text{itree } E$ is known to be a monad. But more specifically, we prove that the strong ($\text{eq.itree})$ and weak bisimulation ($\text{eutt})$ are instances of $eqmR$ and satisfy its laws.

Example 4.2. State.

Computations in the state monad are state-passing functions over a domain of states $S : (\text{stateM} \rightarrow S \rightarrow S * X).$ We consider the standard operations: pure computations leave the state untouched while sequencing threads the states.

\[
\text{ret}^S v = \lambda s \Rightarrow (s, v) \quad \text{bind}^S m k = \lambda s \Rightarrow \text{let } (s', \ a') := m \ s \text{ in } f \ a' \ s'
\]

Since computations in the state monad are functions, the family of relations we consider relaxes equivalence to functional extensionality, and further lifts the relation overt the returned values. The relation of state considered here is equality, but could be additionally relaxed to other equivalences.

\[
m_a \approx_R m_b \triangleq \forall s, R \ (\text{snd} \ (m_a s)) \ (\text{snd} \ (m_b s)) \ \wedge \ (\text{fst} \ (m_a s)) = (\text{fst} \ (m_b s))
\]

We prove that this definition of $eqmR$ satisfies the required laws.

Example 4.3. Error.

Potentially failing computations over a type of errors $E$ can be implemented as a sum type $\text{errorM} E \triangleq E + R$ and equipped with the usual sequencing passing by valid computed value and propagating erroneous states, as depicted to the left in the following:
We highlight the situation of \( \text{propM} \) where the signature \( \text{ChoiceE} \) as well as for specifying a computation.

The proposition \( 14 \) Irene Yoon, Yannick Zakowski, and Steve Zdancewic

\[
\begin{align*}
\text{ret}^{\text{Err}} \ v & \triangleq \text{inr} \ v \\
\text{bind}^{\text{Err}} \ mk & \triangleq \text{match} \ m \ with \\
& | \text{inl} \ e \Rightarrow \text{inl} \ e \\
& | \text{inr} \ v \Rightarrow k \ v \\
\text{x} \approx_{\text{R}} \ y & \triangleq \text{match} \ x, y \ with \\
& | \text{inl} \_ \text{inl} \_ \Rightarrow \top \\
& | \text{inr} \ v_1, \text{inr} \ v_2 \Rightarrow \text{R} \ v_1 \ v_2 \\
& | \_ \_ \_ \Rightarrow \bot
\end{align*}
\]

To the right is an \( \text{eqmR} \) satisfying the required laws: it lifts the relation over valid related results, and accepts co-failure disregarding the value of the error.

**Example 4.4.** Non-determinism.

Non-deterministic computations can be represented as sets of outcomes using \( \text{Prop} : \text{propM} \ R \triangleq R \rightarrow \text{Prop} \). This propositional account of nondeterminism gives up its computational content, but is in exchange flexible to manipulate, allowing for modelling non-determinism over infinite sets, as well as for specifying a computation.

The pure computation is a deterministic one, it hence builds the singleton set: \( \text{ret}^{\text{ND}} \triangleq \lambda a' \Rightarrow a = a' \). The bind should flatten into a single set all possible outcomes for each non-deterministically reachable branch of the computation, i.e., \( \text{bind}^{\text{ND}} \ mk \triangleq \exists a, m a \land k a b \). A notion of bijection up-to relation defines an \( \text{eqmR} \) satisfying most of the laws:

\[
ma \approx_{\text{R}} mb \triangleq (\forall a, ma a \rightarrow \exists b, mb b \land \text{R} a b) \land (\forall b, mb b \rightarrow \exists a, ma a \land \text{R} a b)
\]

We highlight the situation of \( \text{propM} \) due to a use of a related structure in Vellvm, as depicted at the bottom of Figure 2. However, it is instructive to notice that it does not quite satisfy the interface: the \( \text{CONJ} \) rule and the rules related to the product of relations are invalid.

### 4.2 Image of Monadic Computations

The proposition \( ma \approx_{\text{R}} mb \) intuitively asserts that \( ma \) and \( mb \) are compatible computations—weakly bisimilar in the case of ITrees, for instance—and that the relation \( R \) is a valid relational post-condition over returned values. The axioms from Figure 7 provide the basis to justify this interpretation of \( R \) as a post-condition. To conduct relational reasoning, one needs additional structural rules that compositionally relate computations built from combinators. For instance, the ITree library provides an equation relating sequences of computations:

\[
ma \approx_{\text{S}} mb \Rightarrow (\forall x y, S x y \rightarrow \text{ka} x \approx_{\text{R}} \text{kb} y) \rightarrow ma \equiv \text{ka} \approx_{\text{R}} mb \equiv \text{kb}.
\]

This rule mirrors the familiar Hoare-style rule for sequence by quantifying existentially over \( S \), the intermediate post-condition.

Monadic computations expressed as sequences have a more subtle structure, though. A typical monadic computation might return only a strict subset of the values of its return type, \( R \), while its continuation is always defined over all of \( R \). By way of illustration, consider the monad \( \text{intree} \text{ChoiceE} \) where the signature \( \text{ChoiceE} \) provides an event, \( \text{choice} : \text{ChoiceE} \text{bool} \), encoding a binary branching indexed by a boolean. The computation \( c \triangleq b \leftarrow \text{trigger choice}; \text{if} \ b \ \text{then} \ \text{ret} \ 1 \ \text{else} \ \text{ret} \ 0 \) triggers this external choice event, and converts the returned boolean into a natural number. Because \( c \) has type \( \text{intree} \text{ChoiceE} \text{nat} \), any continuation \( k \) bound to \( c \) will be indexed over all natural numbers, but the only relevant branches should be \( (k \ 0) \) and \( (k \ 1) \). The proof rule above may lead to spurious proof obligations. While \( S \) can naturally always be taken sufficiently tight to rule out these spurious cases, there should be a systematic way to strengthen it.

Following this intuition, we introduce the notion of the **image** of a monadic computation: interpreting the diagonal of \( \text{eqmR} \) as a unary logic, we want to define uniformly a notion akin to a strongest post-condition of a computation. For an ITree, the **image** is, intuitively, the set of values that appear at its leaves, so the image of \( c \) will be just the set \( \{0, 1\} \).

#### 4.2.1 Image: a Semantic Characterization

We wish to capture abstractly the set of values possibly returned by a monadic computation. We hence seek to associate to each computation \( m : M A \) a predicate \( \text{image} \ m : A \rightarrow \text{Prop} \) over its return type. Concretely, for an ITree \( t \), \( \text{image} \ t \) should be
the set of leaves of \( t \); for a non-deterministic monad, the image should be the set of values it may return. The challenge is defining the image without referring to the particular structure of the monad definition, which is necessary for incorporating it into the general structural laws.

A first intuition on this path is to consider the diagonal of eqmR: suppose \( m : M \Rightarrow R \) is a computation in some monad \( M \) and \( R \) is a relation at which \( m \) self-relates: \( m \approx R m \). Since \( R \) is a relational post-condition, all pairs of returned values should belong to \( R \), and hence the square of the image \( m \), i.e., \( \{ (a, a) \mid \text{image } m \ a \} \) should be included in \( R \).

Following this idea, one could be tempted to carve out the extra junk in \( R \) by defining the image as the diagonal of the intersection of all relations at which \( m \) self-relates:

\[
\text{image } m \ u \equiv \forall R, m \approx R m \rightarrow R u \ u
\]

Interestingly enough, this first attempt turns out to be too naïve when considering non-determinism; it leads, in general, to a strict subset of the image we seek to capture. To see why, we focus our attention to the case of PropM, with eqmR defined as in Example 4.4.

Let \( (m : \text{PropM bool}) \) be the computation that nondeterministically returns a boolean, that is, \( m = \text{fun} x \Rightarrow x = \text{true} \lor x = \text{false} \). We naturally expect the image of \( m \) to contain both booleans as well. However, by taking for relation \( R = \{ (\text{true, false}); (\text{false, true}) \} \), we easily see that \( m \approx R m \) holds despite \( R \)'s diagonal being empty!

Intuitively, considering all relations is inadequate: since we are interested in the diagonal of the resulting relation, we should only consider those whose diagonal contains the elements it relates. To look on the side of equivalences would however be too drastic: all reflexive relations have their diagonal coincide with the whole return type. In particular, the ITtree representing a silently diverging computation self-relates at all post-conditions, we would hence fail to ascribe an empty image to it.

The right intuition echoes the idea of modeling the codomain of partial functions as Partial Equivalence Relations (PERs). PERs at which self-relation is possible always contain in their diagonal a superset of the image we seek to define, without being forced to contain the whole type. The image is precisely the diagonal of the smallest PER at which the computation self-relates:

\[
\text{imageH } m \ a_1 \ a_2 \triangleq \forall R, \text{PER } R \rightarrow m \approx_R m \rightarrow R \ a_1 \ a_2
\]

We write \( a \in m \) as a short-hand for image \( m \ a \).

### 4.2.2 Image for ITtree: a Concrete Characterization.

The semantic characterization of the image of a monadic computation only relies on eqmR, and can hence be leveraged to axiomatize the desired theory. This definition however gives few reasoning principles. We hence prove that in the case of the ITtree monad, it coincides with the concrete original intuition: a predicate collecting the reachable leaves. This predicate is defined inductively over the structure of the tree, existentially collecting all branches:

\[
\text{Inductive Leaves } \{ E \} \{ A : \text{Type} \} \ (a : A) : \text{ITtree } E \ A \rightarrow \text{Prop} :=
| \text{LeavesRet: } \forall t, t \approx \text{Ret } a \rightarrow \text{Leaves } a \ t
| \text{LeavesTau: } \forall u, t \approx \text{Tau } u \rightarrow \text{Leaves } a \ u \rightarrow \text{Leaves } a \ t
| \text{LeavesVis: } \forall \{ X \} \ (e : E \ X) \ (x : X) \ t \ k, t \approx \text{Vis } e \ k \rightarrow \text{Leaves } a \ (k \ x) \rightarrow \text{Leaves } a \ t.
\]

The structural and semantic definitions are proved equivalent, justifying the abstract definition, and providing inductive reasoning to establish membership to the image.

**Lemma 4.1.** \( a \in m a \iff \text{Leaves } a \ ma \)
4.2.3 The Case of stateM. We prove that the image of a stateM computation captures exactly the set of values that can be returned for some initial state, regardless of the final state.

**Lemma 4.2.** \( v \in m \iff \exists s_i s_f, (m s_i) = (s_f, v) \)

Notice the existential quantification on both the initial and final state.

Anticipating the axiomatization introduced in the following section, we consider how the image predicate and the bind construct interact. Following the analogy of a strongest post-condition, it could be hoped that the following rule universally hold:

\[ u \in m \quad v \in (k u) \quad \Rightarrow \quad v \in (m ;; k) \]

However, this would be misunderstanding what the image expresses: while it universally captures the tightest post-condition over the set of returned value, it cannot do so w.r.t. the effects the computation perform. Since the reachability of branches of the post-condition may depend on the history of effects, it is therefore expected that it will not behave as uniformly w.r.t. bind.

To illustrate more concretely this intuition, we build over this state monad instance a counter example to IMAGEBIND. Fixing the state to bool, consider the following computation that returns the initial state as value, but always sets the final state to true.

\[ m \triangleq \text{fun } b \Rightarrow \begin{cases} (\text{true}, \text{true}) & \text{if } b \\ (\text{true}, \text{false}) & \text{else} \end{cases} \]

The image of \( m \) contains both booleans since either can be found as returned value for a certain initial state. In particular, \( \text{false} \in m \). Now consider the continuation \( k \triangleq \text{fun } v b \Rightarrow (v, b) \) updating the state with its argument and returning the previous state. One can think of this computation as a balanced tree with four leaves, where each subtree admits both booleans in its image: in particular, \( \text{false} \in k \text{ false} \). The branches carrying \( \text{false} \) are, however, reachable only from a state set at false: since \( m \) does not return such state, they are hence unreachable branches. To sum up, \( \text{false} \in m, \text{false} \notin (m \gg k) \), contradicting IMAGEBIND.

Intuitively, the image characterizes what values a monadic computation might possibly return, but does so in a way that is parametric in the monad definition itself. As illustrated by this example, it does not account for information internalized by the monad, such as unreachable states. Nevertheless, the refined (if still approximate) reasoning enabled by the image is an essential ingredient for defining precise reasoning principles in the equational theory.

4.2.4 Image for errorM. As can be expected, the image over the error monad is much simpler to capture: it is either empty if the computation fails, or the singleton of the computed value otherwise.

**Lemma 4.3.** \( a \in ma \iff ma = \text{inr } a \).

4.2.5 Image for propM. For nondeterminism, the computation itself coincides with the image:

**Lemma 4.4.** \( a \in ma \iff ma a \).

4.3 Beyond Monadic Laws

We have all the tools required to define a first minimal axiomatization of the monads we accept to consider. This interface is described on Figure 8 and contains three kinds of properties. As expected, the three traditional monad laws are still required, but are expressed up-to \( \approx \).

Next are three rules constraining properties of the image of a monadic computation. Rules IMAGE\(_L\) and IMAGE\(_R\) systematically link the images of two computations that can be proved to be related by eqmR: any point in one of the images can be related to the other via the post-condition. Rule IMAGE\(_SELF\) ensures part of the intuition we started from: the image should capture all possibly returned values, it should therefore itself be a valid post-condition when self-relating. As illustrated in Section 4.2.3, the IMAGE\(_BIND\) rule should not be an axiom.
Finally come the two proof rules enriching our relational logic. They directly mirror the ones used in the ITree standard library, but generalized to work over arbitrary monads: the RET rule describes how to relate pure computation; the BIND rule how to relate two sequences. We have mentioned at the beginning of this Section that the BIND rule puts all the stress of constraining the image of the computations being related on the choice of \( R_A \). Using the IMAGESELF and RELCOMP rules, we can now prove abstractly the following principle, systematically enriching \( R_A \) by intersecting it with both images.

We illustrate the adequacy of this axiomatization by providing instances.

**Lemma 4.5.** The ITree, stateM\(_5\), errorM\(_5\) and propM monads satisfy the eqmR laws.

Furthermore, we capture the fact that the sequencing operation over ITrees depends exclusively on the computed value:

**Lemma 4.6.** The ITree monad additionally satisfies the IMAGEBIND rule.

### 4.3.1 Inversion Laws.

Figure 8 provides the core forward reasoning rules associated to eqmR, useful for establishing such relations. Reciprocally, backward reasoning is useful to derive information from an established relation between two computations. We capture in Figure 9 the inversion laws that hold true for all our structures of interest.

The two first rules derive information from an hypothesis that two computations of a certain shape are related. The RETINV rule ensures that related pure values are in the post-condition while the FMAPINV rule expresses that when \( fmap \) computations are related, the mapped functions can be pushed down the post-condition.

Interestingly, while we have seen that forward compatibility of the image with \( \text{bind} \) is invalid in monads as common as stateM\(_5\), backward compatibility is always valid. If we know that a value is in the image of a \( \text{bind} \), BINDINV decomposes this hypothesis by exhibiting a branch of the continuation whose image contains this same value.

And indeed, we show:

**Lemma 4.7.** The ITree, stateM\(_5\), errorM\(_5\) and propM monads satisfy the eqmR inversion laws.

Furthermore, in any monad satisfying both eqmR interfaces, the image of a pure computation is exactly the singleton:

**Lemma 4.8.** Over any eqmR monad satisfying all well-formedness laws, \( a \in \text{ret} b \leftrightarrow a = b \).
We have carefully defined the semantic definition of \textit{eqmR} as the smallest PER at which a computation self relates, as opposed to an arbitrary relation. This subtlety has been motivated by the case of the \textit{propM} monad: over this instance, one can self relate at a relation whose diagonal does not contain the image. We show that this restriction can be dropped for the three other example considered:

\textbf{Lemma 4.9.} \textit{Over the itree, stateM\textsubscript{S} and errorM\textsubscript{E} monads, the diagonal of any self-relating post-condition contains the image:}

\[
\begin{align*}
\text{ma} & \approx_{\text{R}} a \\
\text{ma} & \approx_{\text{R}} a \vdash \text{R} \text{ma} \\
a & \in \text{ma}
\end{align*}
\]

\textit{BINDIMAGEINV}

We have not considered any backward reasoning principle for related bind computations. Indeed, no such rule hold in general, but the notion of image allows us to provide one in specific cases:

\textbf{Lemma 4.10.} \textit{Over the itree monad, the following rule, inverting relations between binds sharing a common prefix, holds true:}

\[
\begin{align*}
t & \gg k_1 \approx_{\text{R}} t \gg k_2 \\
r & \in t \\
k_1 & r \approx_{\text{R}} k_2 r
\end{align*}
\]

\textit{BINDINV}

Such an inversion principle is in particular crucial when assuming an invariant of a computation expressed using the diagonal of \textit{eqmR}.

\section{Transporting eqmR via Monad Transformers}

We have specified a minimal equational theory that any target domain for a monadic interpreter must satisfy, and illustrated that it holds on specific monads. These domains are, however, typically built by stacking successive monad transformers atop of a base triggerable monad—ITrees typically—as echoed by the structure of Vellvm’s stack on Figure 2. In the absence of a clean abstract interface as the one we contribute here, users had no choice but to re-establish manually such a theory for each structure they consider. To alleviate this painful work, we hence provide the tools to systematically transport the \textit{eqmR} structure and its theory via appropriate monad transformers. We spell out all the requirement through an additional series of typeclasses.

On the operational side first, a monad transformer \(MT : (\text{Type} \to \text{Type})\to \text{Type} \to \text{Type} \) must provide the traditional \textit{lift} function embedding computations of an arbitrary monad into the transformed structures:

\[
\text{lift} : \forall (M : \text{Type} \to \text{Type})(A : \text{Type}), M A \to MT M A
\]

For any monad \(M\), \textit{lift} \(M\) must define a monad morphism, \textit{i.e.}, an indexed function commuting with the \textit{ret} and \textit{bind} operations. While standard, these two laws, depicted on the upper part of Figure 10\textsuperscript{6}, must be state with respect to \textit{eqmR} in our setup. Furthermore, \textit{lift} must itself respect

\textsuperscript{6}We omit the implicit argument \(M\) in this Figure, writing \textit{lift} in lieu of \textit{lift} \(M\).
eqmR, which we capture in the LiftEqmR rule. Monad transformers must additionally construct valid monads: lifted monads should satisfy the monad laws. Since we request a richer minimal equational theory of our monads, we spell out their preservation (Figure 11): the well-formedness of eqmR, its forward rules and its backward rules should all be preserved.

Example 4.5. stateT. The state monad can be generalized into an appropriate transformer stateT_S M A ≜ S → M (S × A). The return and bind definitions are standard, mirroring the ones from stateMs, but leveraging the underlying monadic operations: ret v ≜ λ s ⇒ ret (v, s) and bind c k ≜ λ s ⇒ c s ;; k v s. The lift operator relies on the underlying fmap to reinject the unchanged state: lift c ≜ λ s ⇒ fmap (λ x ⇒ (s, x)) c.

We parameterize the definition of eqmR at any monad transformed by the state transform by a relation on states R_{st}.

\[
ma ≈ mb ≜ \forall s_1 s_2 . (s_1, s_2) ∈ R_{st} → ma s_1 ≈_{R_{st} ⊗ R} mb s_2
\]

where \( R ⊗ R' \) is defined as \( λ x y ⇒ R x y ∧ R' x y \). Assuming related input states, eqmR unfolds the definition of the state monad to see it explicitly as a computation in the underlying monad over the product type, and enforces both the state relation on output state and the relation on values by taking as a post-condition the product of both relations.

The image admits a similar characterisation as in the case of stateMs, but expressed in terms of the underlying notion of image:

**Lemma 4.11.** \( a ∈ ma \iff (∃ s s' . (ma a) ∈ (s', a)) \)

We prove that stateT_S transports all interfaces, providing our equational theory for free for structures such as the ones introduced in the first three layers of Figure 2 or in both languages of the compiler described in the original ITree paper [46].

Example 4.6. errorT. Similarly, we support the standard generalization of errorM to a monad transformer: errorT_E M A ≜ M(E + A). The return and bind operators, omitted here, are standard. The lift simply injects the result of the underlying computation into a successful one: lift ≜ λ ma ⇒ fmap inr ma. Finally, eqmR is parameterized by an arbitrary relation over errors and feeds the coproduct of both relations to the underlying eqmR: ma ≈ mb ≜ ma ≈_{R_{exn} ⊗ R} mb.

We prove that errorT_E preserves all interfaces. The characterization of the image still holds: they are the successful elements of the underlying image.

**Lemma 4.12.** \( a ∈ ma \iff (inr a ∈ ma) \)

4.5 Relating Computations across Distinct Monads

For clarity of exposition, we have presented eqmR as a heterogeneous relation over the return type, but assuming the same monad on each side, which is the most direct analog to the ITrees library notion of eutt. However, in many situations—e.g., when expressing the correctness of a pass of compilation, such as the IMP-to-ASM compiler (see Section 6—we need to work across languages and hence relate computations in distinct monads. We therefore also provide in our formal development a more general notion of family of relations, heqmR, parameterized by two monads \( M \) and \( N \), and lifting relations at return types (\( M A → N B → \text{Prop} \)) to relations at computation-level across monads (\( M A → N B → \text{Prop} \)). Such relations are typically specific to the proof at hand: each monad still comes with its own eqmR that the cross-monad relation must be proved to respect.
5 LAYERING EQMR WITH INTERPRETERS

In the previous section, we have discussed how to build equational theories for monads and monad transformers, where we exposed several semantic characterizations such as element inclusion (using image) and equational laws that certain monads satisfy.

How does this relate to building layered interpreters? The monadic equational framework is the basis for expressing structural properties for interpretation. When we layer the interp combinator (from Section 3), we will also like certain structural properties to hold at each layer of interpretation. For instance, interp should respect monadic operators such as ret and bind, and interact well with iteration. Now, given an interpretable monad (see Section 3.4), equipped with an appropriate instance of eqmR, we can state what laws the trigger, over, and interp functions should obey. These laws are shown in Figure 12.

The InterpRet and InterpBind laws say that interp h is a monad homomorphism.

The InterpTrigger rule applies when the event being triggered belongs to the signature handled by the handler—it simply says that the interpretation (morally) is the result of the handler applied to the event. The lift on the right-hand-side is the one from the functor associated with the interpretable monad, and it coerces the handler’s output to the right form. (See the type of interp in Section 3.4, and the definition of lift in Section 4.4). Note that the handler might itself contain over, which means that this rule can still apply for a function that injects a handler for a program with a larger signature.

IgnoreTrigger covers the case when the handler cannot act on the triggered event. For example, for interp handle_mem (trigger Print), where handle_mem is a handler for memory operations, and Print is the event for I/O. Print cannot be handled by handle_mem, and thus is propagated by re-triggering the event in the target monad, in this case, as lift (trigger Print).

The last rule, InterpIter formalizes the interaction between iteration and interpretation. It says that if the interpreter respects the loop body for every iteration j, then it commutes with iter. This provides a kind of lock-step simulation principle that is useful in practice when reasoning about equivalence of two possibly diverging computations.

5.1 Higher-order Functors Lift Structural Properties : Interp Laws for any Stack

The key contribution in our framework is that these interpretation laws about monads need to be proven only once and for all, regardless of the stack of interpretation. Specifically, we have proved that the laws in Figure 12, which are specified via a typeclass in Coq, hold for ITrees as a base instance, with its standard definitions of interp, iter, etc.. To account for layered interpretations, we then define a set of well-formedness conditions on the monads and monad transformers, which let us derive further instances of the interpretation laws by applying monad transformers.

Recall the definition of stacking interpretation from Section 3.4—it derives instances of Interp.

Instance stack_interp {T IM : (Type → Type) → Type → Type} {M : Type → Type} {HFunctor T} {IterativeMonad M} {Interp IM Id M} : Interp IM T M :=
  fun E h R t ⇒ hfmap (interp h) t.

Fig. 12. Interpretation Laws (we write h_E for a handler of type E ~ M and e_E for an event of E)
One constraint imposed by this definition is that \( M \) is an \texttt{IterativeMonad}, a property that must be lifted to \( T \ M \) for a correct interpretation to exist. The other key well-formedness conditions express the functorial properties of the higher-order functor, \texttt{hfmap}, used in this definition. Such structural properties of higher-order functors, and their use in nesting monadic types, is known in the literature (see [18]); and we adapt those definitions for use in our setting. Select higher-order functor laws are presented in Figure 13, which shows that \texttt{hfmap} transports identity, commutes with composition, and preserves monad morphisms. These are unsurprising properties for higher-order functors, but they are useful in our setting when lifting \texttt{ret}, \texttt{bind}, \texttt{iter}, and \texttt{lift} combinators. The remaining requirements follow from general facts about compositionality: the function composition of monad transformers, iterative monad transformers, and higher-order functors preserve the operations and well-formedness properties, as shown in Figure 14, where \( >\gg \) is function composition. The structural properties \texttt{MonadT}, \texttt{IterativeMonadT}, and so on, correspond to the typeclasses that capture the structures and well-formedness laws. The complete set of \texttt{hfmap}-related laws and the details of these typeclasses is included in the Coq development.

Although there is a fair amount of effort to instantiate these well-formedness requirements for a given monad transformer, that work needs to be done only once, after which we can build interpretable monads by applying the transformer. We have shown that all of the necessary well-formedness properties hold for the identity monad transformer, state monad transformer, and error monad transformer. These form a useful basis for building language semantics; we expect that additional monad transformers could also be verified to satisfy these requirements, but leave that to future work.

6 EQMR IN PRACTICE: IMPLEMENTATION AND CASE STUDY

This section surveys our Coq formalization, briefly describing the typeclasses necessary for packaging the equational theory (Section 6.1), the custom tactics necessary for implicit type resolution (Section 6.1.1), and a case study of an \texttt{imp} to \texttt{asm} compiler to see how \texttt{eqmR} can be used in practice.

6.1 Typeclasses for EQM and Interp laws

Figure 15 summarizes the collection of typeclasses supported by our interpretation framework. The red nodes are the category theory-relevant typeclasses from the Interaction Trees library

![Diagram showing typeclass dependencies in EqmR framework](image)
(most notably the theory of iterative monads and equivalence on Kleisli arrows), the yellow nodes are standard functional programming typeclasses (functor, monad, monad transformers, higher-order functors), and the green nodes represent the structural properties that we have formalized in this framework. A dotted line connects an “operational” typeclass to its corresponding laws, and solid arrows represent dependencies.

6.1.1 Custom Tactics for Extending Coq Typeclass Inference. Without explicit support, the Coq typeclass inference algorithm often fails to infer the implicit arguments for interp, over, and trigger when using the interp laws. We solve this issue by providing custom tactics for each of the relevant lemmas for our interp laws: these are dubbed i-tactics for “interp law tactics”. For each of the interp laws, there is a corresponding i-tactic, and we also have a specialized setoid-rewriting tactic irewrite for inferring the necessary setoid rewriting we need for the laws.

This is necessary because (1) there are multiple overlapping instances of equational laws (for instance, both the strong and weak bisimulation in the ITree library satisfy eqmR rules), (2) the decomposition of stack of monad transformers is not unique (for instance, the identity monad transformer \( \text{fun} \ x \Rightarrow x \) is a trivial instance of a monad transformer), and (3) the type annotation for the sums of events of the over combinator in the IgnoreTrigger law (recall Figure 12) presents a difficult typeclass-resolution problem, and we would like the users not to use so many explicit type annotations while using these laws. We have developed a small custom tactic library for inferring the typeclass instances. Here we sketch (a simplified version of) the implementation of ibind, which corresponds to the application of the INTERPBIND law.

```coq
Ltac ibind_body TR h t k :=
  match type of h with
  | \forall T : Type, ?E T \rightarrow T \Rightarrow
    let Hbind := fresh "Hbind" in
    pose proof
    (interp_bind (T := TR) (E := E)
      h k t as Hbind;
    try (irewrite Hbind)
  end.

Ltac ibind_rec TR h x :=
  match x with
  | interp (T := ?TR) ?h ?x_ \Rightarrow
    ibind_rec TR h x_
  | bind ?t ?k \Rightarrow
    ibind_body TR h t k
  end.

Ltac ibind :=
  match goal with
  | \exists T : Type, interp (T := ?TR) ?h ?x \Rightarrow
    ibind_rec TR h x
  end.
```

The base case is ibind_body, which instantiates the INTERPBIND law with explicit type arguments for the monad transformer TR, triggered event type E, handler h, continuation k and prefix t of the bind operator. In ibind_body, we must explicitly match the type of the handler, h, because, given a program interp h (interp h' (x ← ma ;; k x), the typeclass inference in Coq cannot determine whether it should attempt applying the commutation lemma to the inner or the outer interp function. The irewrite tactic infers the specific setoid instance needed in order to perform the rewriting. The ibind_rec recursively matches against the argument of interp to specify the innermost monad in the interp function, and the top-level tactic ibind, simply calls into ibind_rec to start the process.
6.2 Case Study: IMP to ASM Compiler

We have re-verified the correctness of a simple imperative language (IMP) to a simple assembly language (ASM) compiler, the case study presented in the original Interaction Trees paper [46], in this equational framework to illustrate how the equational laws are lifted in the stack of interpretation. In the following subsection we discuss the benefits to using the framework that we have observed.

6.2.1 Sleek staged interpretations: ASM Example. Recall from Section 3.3 how automatic injection of handlers eliminates the need for extra manual annotation. The benefit for our layered interpretation is larger when it comes to larger stacks of interpretations as presented in 2, but we present here a simplified version for expository purposes. We illustrate this point again in ASM, and illustrate another benefit: writing layered interpreters is more straightforward.

The ASM language has two event signatures: Reg, for registers, and Memory, for the heap, and has two handlers respectively, h_reg and h_memory. Let’s look at the original definition of interp_asm, which uses bimap in order to manually inject the handlers for each of the events.

\[
\text{Definition} \quad \text{interp}_\text{asm} \ (E \ A) \ (t : \text{itree} \ (\text{Reg} + \text{Memory} + E) \ A) := \\
\quad \text{let} \ h := \text{bimap} \ h_\text{reg} \ (\text{bimap} \ h_\text{memory} \ (\text{id} \ _ \ _)) \ \text{in} \\
\quad \text{let} \ t' := \text{interp} \ h \ t \ \text{in} \ \text{fun} \ \text{mem} \ \text{regs} \Rightarrow \text{interp}_{\text{map}} \ (\text{interp}_{\text{map}} \ t' \ \text{regs}) \ \text{mem}.
\]

The \text{bimap} \ h_\text{reg} \ (\text{bimap} \ h_\text{memory} \ (\text{id} \ _ \ _)) manually injects the handlers for Reg and Memory for the signature of the program \text{itree} \ (\text{Reg} + \text{Memory} + E) \ A. The application of \text{interp}_{\text{map}} \ and \text{interp} does not accurately reflect the intuition that we are staging interpretation: in fact, the interpreters are being applied simultaneously. In addition, the event signature is very concrete, in the sense that the signature must provide the ordering of \text{Reg} + \text{Memory} + E for this \text{interp}_\text{asm} to apply.

The code below is the sleek, staged interpretation using \text{over} and the new \text{interp} combinators.

\[
\text{Definition} \quad \text{interp}_\text{asm} \ (D \ E \ F \ A) \ '(\text{Reg} +? D \ < \ E) \ '(\text{Memory} +? E \ < \ F) \ (t : \text{itree} \ F \ A) := \\
\quad (\text{interp} \ (T := \text{stateT} \ \text{memory}) \ (\text{over} \ \text{handle_reg}) \ (\text{interp} \ (T := \text{fun} \ x \Rightarrow x) \ (\text{over} \ \text{handle_mem}) \ t)).
\]

Observe the benefits in writing the interpretation in this manner: now, each stage of interpreting Reg and Memory are clearly distinct, and are compositional. The \text{over} annotation also eliminates the need for manual annotation. It is also extensible in the sense that adding another stage of interpretation is straightforward—the event signature is not rigid.

6.2.2 Structural Rules for Free. In the original IMP-to-ASM proof of correctness, one had to show that structural properties hold for each layer of interpretation. For instance, if one wanted to use the \text{InterpRet} law with layers \text{stateT} \ \text{reg} \ (\text{stateT} \ \text{memory} \ (\text{itree} \ E)) \ , an instance would need to be proven for each of \text{itree} \ E,\text{stateT} \ \text{memory} \ (\text{itree} \ E), and \text{stateT} \ \text{reg} \ (\text{stateT} \ \text{memory} \ (\text{itree} \ E)) (or, less compositionally, one single, ad hoc instance that essentially combines all three proofs into one). This does not scale, especially when given a large stack interpreters with many laws. However, now we get these properties for free.

That shows how definitions can be simplified. How about proofs? The following is a structural lemma about how the interpretation function for ASM commutes with \text{bind}.

\[
\text{Lemma} \quad \text{interp}_\text{asm}_\text{bind} : \forall \ (R \ S) \ (t : \text{itree} \ E \ R) \ (k : R \rightarrow \text{itree} \ E \ S), \\
\text{interp}_\text{asm} \ (\text{bind} \ t \ k) \approx (x \leftarrow \text{interp}_\text{asm} \ t ;; \ \text{interp}_\text{asm} \ (k \ x)).
\]

Now compare the old proof to the new proof, as shown to the right. The old proof of the lemma had to refer explicitly to the bind commutation property at each layer (\text{interp}_\text{bind} and \text{interp}_\text{state_bind}), which are specific to the layers and cannot be composed with each other. In addition, it used the \text{eutt_clo_bind} lemma to perform rewriting under the monadic
bind, while the intuitive reasoning principle should be “commute the bind operator under interp twice”.

The new proof, only has to apply the same bind commutation lemma using the ibind tactic, which essentially unfolds to invoking ‘rewrite interp_bind’ twice, while inferring the correct setoid instances for rewriting the InterpBind rule.

At each layer of interpretation, we have the same bind commutation property that holds, and we do not need to conjecture the structural property to hold or reprove it for each combination of layers.

6.2.3 Commuting layers of interpretation. To determine that we indeed have flexibly composable structural rules, we can modify the structure of an existing development and see how hard it is to “port” the proofs to the new structure. To that end, we modified the existing imp to asm compiler development by swapping the order in which registers and memory events are interpreted. Of course such a change necessitates certain modifications: for instance, we had to re-order the arguments to the relation connecting the imp and asm state monads. However, beyond those simple changes, most of the proofs go through with minimal differences—the most complicated change being a couple instances where the IgnoreTrigger rule applies at a different position in the stack.\footnote{The supplementary Coq material contains the proofs.}

The old proofs (like the one shown above) would be far less resilient to such changes, and therefore much more difficult to maintain than the new ones enabled by this framework.

7 RELATED WORK

Monads, Monad Transformers, and Modular Interpreters. The approach of building modular interpreters from monads and monad transformers derives from an expansive literature. Moggi’s seminal papers [30, 32] about using monads to characterize imperative features in a pure, functional setting was consequently popularized by Wadler [45] and Peyton Jones [34]. Monad transformers [31] were then adopted as a way of composing various effects: notably Liang, et al. [25] showed how to build modular interpreters in that style, which we have adopted and formalized here. Swierstra [43], Apfelmus [4], Kiselyov, et al. [21], and Kiselyov and Ishii [20] have showed how to use the free monad to define modular monad instances. Interaction Trees [46] combine these ideas together, and provide an instance where each interpretation layer forms an iterative monad. In our framework, we maintain that the range of interpretation is a stack of iterative monads and show how in the practical setting, the laws for iteration can be composed and lifted through monadic interpreters.

Nesting interpreters are also a prominent feature of Johann and Ghani’s work [18]. That work introduces two separate constructs for building layered monads: a base interpreter and a second interpreter for nesting. Our general interp definition subsumes both definitions, but uses a distinct base instance. Our eqmR framework shows how to build a formalized equational theory that works nicely with the lifting and plays well with Coq-style typeclasses, and so can be seen as a mechanized version of their results.

Algebraic Effects and Handlers. Interaction trees and languages with support for algebraic effects [6, 17, 36–38] share similar goals, namely flexible, programmable construction of semantics. As such, our work has taken inspiration from that literature. There are significant differences, however. Programming languages that implement algebraic effects, like Eff [7], are working in an ambient environment that allows nontermination, whereas ITrees are crafted to fit with Coq’s total semantics. The handlers for algebraic effects are more general than those possible with ITrees. In particular, the ITree handlers do not have access to the continuation of the event, which makes them less
expressive (giving the handler access to the continuation would be hard to realize in Coq because its totality checker would not be able to observe that manipulations of the continuation are sufficiently guarded to define valid cofixpoints).

The “algebraic” reasoning part of algebraic effects is often used in that setting to characterize equations that hold for particular effects; for instance, for state effects the sequence \( \text{put } s; \text{put } t \) is equivalent to just \( \text{put } t \) (since the second \( \text{put} \) overwrites the first). Such equivalences can be proven as theorems about particular monads used for interpreting events. In our context, such a theorem would justify a rewriting rule with respect to the state monad notion of \( \text{eqmR} \). This paper, however, focuses not on such monad-specific properties. Instead, we are looking at how to automatically construct the generic, structure-preserving, parts of equational theories compositionally.

Relational Logics. Benton’s Relational Hoare Logic [8] and Nanevski, et al.’s Relational Hoare Type Theory [33] have been shown to be useful for reasoning about program transformations and properties such as information flow. The subsequent work on predicate transformers [44], Dijkstra Monads [2, 26, 27], and F* [42] give a general framework for building program logics for arbitrary monadic effects. The Dijkstra Monad setting is especially relevant to our approach and is similar in that it builds a general logic for reasoning about monads, but is different in that it does not focus on composing reasoning about effects (i.e., building a compositional theory using interpretation).

Logical Relations and Bisimulations for Monadic Types. There are many techniques for relational reasoning, including binary logical relations [3, 9, 12] and bisimulations [11, 22, 39], and their combination [15, 16]. The EqmR framework defines relations for monads that technically form logical relations. Logical relations over monadic types were developed by Goubault-Larrecq, et al. [13], giving a sound basis for a monadic equivalence akin to a notion of bisimilarity. As in our approach, these techniques build an expressible basis for program logics and program verification [19].

PER Models. The notion of “image” is inspired from PER models of computation, as used in, e.g., models of the lambda calculus with recursion and recursive types [29, Chapter 5], for reasoning about operational equivalences [5], or for giving different “views” of information [1]. In our setting, we use a PER model to gain set-theoretic reasoning about elements in a monadic types, as a way to refine our reasoning principles.

8 DISCUSSION AND CONCLUSION

We have presented a novel, principled approach to the construction of monadic interpreters built in layers from a free-monad structure such as ITrees. The tools we have introduced and formalized in Coq greatly reduce the boilerplate and glue code needed to construct such interpreters, but also provide for free the backbone of the equational theory necessary for any relational reasoning over the resulting structure. Our current implementation provides instances for the structures most commonly used in existing ITrees projects, lifting the equational theory through state and error transformers. However, the zoo of monads is vast: expanding the library to cover additional effects would be a valuable extension.

Among those effects, non-determinism is particularly worthy of attention. Indeed, we have presented in Section 4 the \( \text{propM} \) monad, but looking back at Figure 2, the Vellvm developers use a more general structure that models not just non-deterministic sets of values, but rather sets of computations. This approach relies on a hypothetical \( \text{propM} \) transformer, \( \text{propT}_M A \triangleq M A \to \text{Prop} \).

For Vellvm, the authors have fixed \( M \) to be \( \text{itree E} \), defining a new ad hoc structure rather than a transformer. Interestingly, the reason they did that is probably that they were lacking a generic notion of image for a monadic computation, which we introduced through this paper. Indeed, they define the bind over this structure as:

\[
\text{bind } P K t_b \triangleq \exists t_a, k, P t_a \land t_b \approx \text{bind } t_a k \land (\forall a, \text{Leaves } a t_a \to K a (k a))
\]
This should be read as judging whether an ITree $t_b$ belongs to the bind: there should be a computation $t_a$ in the prefix $P$ and a continuation $k$ such that at any leaf of $t_a$ (i.e., in the image of $t_a$ to use our terminology), the continuation belongs to the nondeterministic set. Without the restriction to the image, this construction is completely ill-behaved. The reason we don’t include $\text{propT}_M$ here is that, even using the image, $\text{propT}_M$ doesn’t lift the monad laws properly—this definition of bind does not associate to the left (an expected artifact of the nondeterminism [27]). Developing a clean non-deterministic transformers for monadic interpreters hence remains an interesting prospect.

With these extensions to the ITrees library in place, we greatly smooth the experience of working with monadic interpreters by generically specifying and transporting all the structural equational boilerplate required. Monadic reasoning is, of course, about more than structure: the extensions in this paper express nothing about monad-specific algebras. This aspect of the reasoning is still fairly ad hoc in the ITrees landscape; combining our contribution with other techniques, such as Dijkstra monads [26, 27], would likely improve even further the viability of monadic interpreters for relational reasoning at scale.

REFERENCES


