Formal Reasoning about Layered Monadic Interpreters

IRENE YOON, University of Pennsylvania, USA
YANNICK ZAKOWSKI, Inria & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), France
STEVE ZDANCEWIC, University of Pennsylvania, USA

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. 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 ITree library, eliminating large amounts of unwieldy boilerplate code and dramatically simplifying proofs.

CCS Concepts: • **Software and its engineering** → **Software libraries and repositories**; • **Theory of computation** → *Logic and verification*; *Equational logic and rewriting*; **Denotational semantics**.

Additional Key Words and Phrases: Coq, monads, coinduction, compiler correctness

ACM Reference Format:
Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022. Formal Reasoning about Layered Monadic Interpreters. *Proc. ACM Program. Lang.* 6, ICFP, Article 99 (August 2022), 29 pages. https://doi.org/10.1145/3547630

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 [Liang and Hudak 2000; Liang et al. 1995; Moggi 1989; Steele 1994]. 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 [Ahman et al. 2017; Maillard et al. 2019, 2020]) that can be used to prove properties ranging from the correctness of program optimizations to information-flow noninterference [Benton 2004].

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 role. They are particularly well suited for defining the semantics of effects when the metalanguage is *pure* and *total*, which is the

Authors' addresses: Irene Yoon, University of Pennsylvania, USA, euisuny@cis.upenn.edu; Yannick Zakowski, Inria & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), Lyon, France, yannick.zakowski@inria.fr; Steve Zdancewic, University of Pennsylvania, USA, stevez@cis.upenn.edu.

2475-1421/2022/8-ART99
https://doi.org/10.1145/3547630

case when embedding a language semantics into dependent type theory, such as in Coq. Moreover, by working with *free monads* [Swierstra 2008] 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 [Letan et al. 2018], as *I/O-trees* [Hancock and Setzer 2000], and as McBride's *general monad* [McBride 2015].

In this paper, we focus on *interaction trees* [Xia et al. 2020] (ITrees), a recent realization of this approach as a Coq library. ITrees are defined as a coinductive variant of the *freer monad* [Kiselyov and Ishii 2015] and are also closely related to *resumption monads* [Piróg and Gibbons 2014]. The core data structure, `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 `itree (E +' F)` can have events drawn from either `E` or `F`, allowing for a modular definition of computations. Second, it is possible to write interpreters for the events in an ITree. We write `h : E ⇝ M` to mean that `h` is an *event handler* that maps events in the signature `E` into computations in the monad `M`. Then `interp h : ∀ A, itree E A → 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 signature of events and event handlers allows for the decomposition of effects into stages, modularizing the semantics in a style akin to *algebraic effects* [Plotkin and Power 2003]. Third, by implementing the `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. The type `itree voidE` (where `voidE` is the empty event signature, and is the unit of `+'`) is isomorphic to Capretta's *delay monad* [Capretta 2005], 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 (given an equivalence relation R), and the relational theory 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 [Xia et al. 2020] illustrated the approach by proving the correctness of a toy compiler from IMP, an imperative language, to 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 [Zakowski et al. 2021]. 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 [Zhao et al. 2012], 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 ITree semantics have

been evaluated in several dimensions. With respect to LLVM IR transformations, simple peephole optimizations admit local, simple proofs, and proving a block fusion transformation correct can be done at a high level of abstraction—verifying a transformation that only impacts control-flow does not depend on the implementation of the state. With respect to front-ends targeting LLVM IR, the HELIX [Zaliva et al. 2020] 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 ITree-based monadic interpreters for formal verification, things do not remain as smooth at the scale of Vellvm and HELIX as they were for IMP and 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 ITree 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 ⤳ M` cannot readily be lifted to a handler `h' : (E +' F) ⤳ 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 [Swierstra 2008] (and re-implemented in the ITree 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, consider a handler `h : E ⤳ stateT S (itree F)` that interprets `E` events into the monad `stateT S (itree F)`, 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)` and `interp h` is of type `itree E ⤳ stateT S (itree F)`. If we have another handler `f : F ⤳ M`, we would like to *compose* the interpreters: `(interp f) ∘ (interp h)` should have type `itree E ⤳ stateT S 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 ITree 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 ITree 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`, and `Prop`.

Section 4 develops new tools for relational reasoning, centered on a typeclass `eqmR` (for "equivalence of monads up to `R`")—a generalization of ITrees weak 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 `eqmR`, and it is a key ingredient needed to define the equational theory.

Section 5 uses `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 provides type instantiations to help disambiguate typeclass resolution. 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 [Yoon et al. 2022] 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.

2 INTERACTION TREES AND MONADIC INTERPRETERS: BACKGROUND AND SHORTCOMINGS

Interaction Trees [Xia et al. 2020] (ITrees) have recently emerged in the Coq ecosystem as a rich toolbox to build compositional and modular monadic interpreters. One of its main benefits comes from its equational framework that allows reasoning about equivalence and refinement of computations. Through this section, we introduce the necessary background information to understand the theoretical and practical limitations that arise when using the framework at scale.

2.1 Interaction Trees: A Free Monad Supporting General Recursion

Interaction Trees are a data structure for representing computations interacting with an external environment through *visible events*. It has a coinductive datatype, modeling potentially diverging computations. Unlike other ways of specifying semantics in Coq (*e.g.* relational operational semantics), ITrees can be extracted into executable programs.

The definition of the ITree datatype, as well as the type signatures of its main combinators, are shown in Figure 1.[1] The datatype takes as its first parameter a signature—described as a family of types `E : Type → Type`—that specifies the set of interactions the computation may have with the environment. The `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 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.

[1]The signature of ITrees is presented with a positive coinductive datatype for expository purposes. The actual implementation is defined in the negative style.

```
CoInductive itree (E: Type → Type) (R: Type) : Type :=
| Ret (r: R)         (* computation terminating with value r *)
| Tau (t: itree E R) (* "silent" tau transition with child t *)
| Vis {A: Type} (e : E A) (k : A → itree E R). (* event e yielding an answer in A *)

Notation "E ⤳ F" := (∀ X, E X → F X).
(* Embedding of pure computations *)
Definition ret {E : Type → Type} {R : Type} (v : R) : itree E R.
(* Sequencing computations *)
Definition bind {E : Type → Type} {T U : Type} (u : itree E T) (k : T → itree E U) : itree E U.
(* Atomic itrees triggering a single event. *)
Definition trigger {E : Type → Type} : E ⤳ itree E.
(* Fixed-point combinator *)
Definition iter {E : Type → Type} {R I: Type} (body : I → itree E (I + R)) : I → itree E R.
```

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

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.[2] 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:[3]

```
_ ← trigger (inr1 (Put 3));; x ← trigger (inr1 Get);; trigger (inl1 (Print x))
```

This computation has type `itree (printE +' cellE) 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, which imply that continuation trees of type `A → itree E B` form a traced monoidal category [Bloom and Ésik 1993], can be also generalized for any arbitrary monad. It corresponds to Kleisli arrows (i.e. functions of type `A → M B` given a monad `M`) forming a traced monoidal category. We call any such monad which satisfies the iterative laws an *iterative monad*.

[2] In 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`)

[3] We write `(x ← t;; k)` and `t >>= k` as notations for `(bind t (fun x ⇒ k))` and `(bind t k)`.

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 +' cellE ~> stateT nat (itree printE) :=
fun _ e n ⇒ match e with
           | inl1 (Print x) ⇒ trigger Print x;; Ret (n, tt)
           | inr1 Get       ⇒ Ret (n, n)
           | inr1 (Put m)   ⇒ Ret (m, tt)
           end.
```

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 can become unwieldy. Let's look at an example of a realistic semantics to see what can go wrong: for instance, the Vellvm project [Zakowski et al. 2021] uses ITrees to formalize a large sequential subset of LLVM IR, an industrial-strength intermediate

When dealing with large languages, the naïve interpretation scheme sketched above, which interprets *all* of its events at once, is undesirable for a couple reasons. First, some effects may be implemented in terms of others: memory operations, for instance, may introduce undefined behavior events. Second, decoupling the interpretation of different categories of events modularizes the monadic structure they introduce, improving the modularity of the semantics and the robustness of the formalization. Such decoupling leads to proof techniques allowing for some of the effects to remain *uninterpreted* during a proof—Zakowski, *et al.* [Zakowski et al. 2021] illustrate this idea by proving a simple block fusion optimization correct independently of the implementation of the memory model.

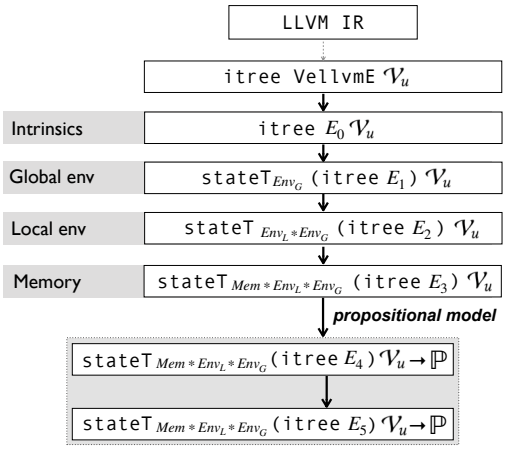

Fig. 2. Vellvm's semantics: a stack of interpreters

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 nondeterministic concretization of `undef` values, and (6) the nondeterministic refinement of undefined behaviors.

```
Definition handle_state {E} :
  stateE ~> stateT S (itree E) :=
    fun _ e s => match e with
                 | Get    => Ret (s, s)
                 | Put s' => Ret (s', tt)
                 end.

Definition pure_state {S E} :
  E ~> stateT S (itree E) :=
    fun _ e s => Vis e (fun x => Ret (s, x)).

Definition interp_state {E} :
  itree (stateE +' E) ~> stateT S (itree E) :=
    interp (case_ handle_state pure_state).
```

Fig. 3. State interpreter from the ITree library

```
Definition handle_local {E} `{FailureE -< E} :
    (LocalE k v) ~> stateT map (itree E) :=
  fun _ e env =>
    match e with
    | LocalWrite k v => ret (add k v env, tt)
    | LocalRead k    =>
      match lookup k env with
      | Some v => Ret (env, v)
      | None   => fail
      end
    end.

Variable (E F G H: Type -> Type).
Context `{FailureE -< G}.
Notation Effin := (E +' F +' LocalE +' G).
Notation Effout := (E +' F +' G).
```

```
Definition E_trigger {M} : ∀R, E R -> stateT
      M (itree Effout) R :=
  fun R e m => r ← trigger e ;; ret (m, r).

Definition F_trigger {M} : ∀R, F R -> stateT
      M (itree Effout) R :=
  fun R e m => r ← trigger e ;; ret (m, r).

Definition G_trigger {M} : ∀R , G R ->
    stateT M (itree Effout) R :=
  fun R e m => r ← trigger e ;; ret (m, r).

Definition interp_local_h := (case_
    E_trigger (case_ F_trigger (case_
    handle_local G_trigger))).

Definition interp_local : itree Effin ~>
    stateT map (itree Effout) :=
interp_state interp_local_h.
```

Fig. 4. Interpreting Vellvm's register map

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 `handle_state`, which is parametric in the leftover ambient signature `E`. The branch implementing `printE` is captured generically in `pure_state`, since the implementation does not depend on the effect that remains uninterpreted. Finally, since `+'` forms a coproduct for an indexed function, a generic `case_` combinator builds the handler used to interpret an arbitrary computation containing `stateE` events, as shown in `interp_state`.

So surely we should be able to happily simplify the definition of `handle_cell` by using the standard library's `stateE` events instead of specialized `cellE`, and directly defining `interp_cell` as `interp_state`? Unfortunately, we cannot! A slight mismatch creeps in: our previous computations have been defined over the `printE +' cellE` signature, while `interp_state` forces `stateE` to be in the head position: `stateE +' printE`. We are 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 setup, 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 `case_` are needed to define `interp_local_h`.

Also, notice the constraint `Failure -< G` in the context, which morally represents "any event `G` that supports failure". This corresponds to the constraint mechanism introduced by Swierstra's Data Types à la Carte [Swierstra 2008] 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 to 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 in 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. 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 pointwise. Such construction breaks the abstraction of layered monadic interpretation.

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 to 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 annotations of `inr1` and `inl1`. To simplify this interpreter, we define a generic function that injects `handle_state` into an arbitrary signature containing `stateE`. There are two main challenges in defining this automatic injection.

First, injecting the `handle_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 `handle_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 `trigger e := Vis e (fun x ⇒ x)`, capturing the idea of a "minimal" impure computation performing an uninterpreted event—in this case, specialized to the ITree monad. The `pure_state` function from Figure 3 mirrors this intent, but in the monad `stateT S (itree E)`; it additionally makes explicit that this minimal computation does not affect the state.

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

```
Class Trigger (E: Type → Type) (M: Type → Type) := trigger: E ⤳ 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 [Spitters and van der Weegen 2011], that proves beneficial for mathematical formalizations in type theory. All typeclasses in our framework use this unbundled style.

Naturally, `ITree.trigger` is an instance for `itree E`, and `pure_state` for `stateT S (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.*, that `∀ t, interp trigger t ≈ t`. Second, when seen as an interpreted computation, it coincides with the effect of the handler on the event, i.e. `∀ h e, interp h (trigger e) ≈ 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 -< 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 `Subevent E F G`, written `E +? G -< F`, provides an isomorphism between the types `E +' G` and `F`:

```
Class Subevent {E F G : Type → Type} : Type := { split : F ⤳ E +' G ;
                                                 merge : E +' G ⤳ 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

```
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} `{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

object manipulated, typically a triggered event or a handler—and infering from this information the complement `G`.

```
Instance Subevent_Base {A B}: A +? B -< A +' B.
Instance Subevent_refl {A} : A +? void1 -< A.
Instance Subevent_Sum_In {A B C D}
  `{A +? D -< B} : (C +' A) +? D -< C +' B.
Instance Subevent_Sum_Out {A B C D}
  `{A +? D -< B} : A +? C +' D -< C +' B.
Instance Subevent_commute {A B C}
  {Sub: A +? B -< C} : B +? A -< 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 `void1` as a unit, allows 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. Events 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 `F` into the same monad:

```
Definition over {E F G M : Type → Type} `{E +? G -< F} `{Trigger G M}
  (h : E ⤳ M) : F ⤳ M :=
  fun _ 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

```
Definition interp_state {E F} `{stateE +? E -< F} : itree F ⤳ stateT S (itree E) :=
  interp (over handle_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`

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. When staging a stack of interpreters, we end up having to interpret from monads such as "stateT S (itree E)": this is the motivation for a notion of *interpretable monads*, which are layered monads that satisfy the structural properties for interpretation. *Interpretable monads* must encompass monads built from ITree through layers of interpretation: they should essentially be iterative monads that can 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.interp` 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.[4]

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 [Johann and Ghani 2009], 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.
```

[4]Because of the 'unbundled' approach, the structural constraints (such as well-formedness properties of their operators) on this multi-parameter typeclass are not apparent here.

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 ITree 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.

4 A COMPOSABLE EQUATIONAL THEORY FOR MONADS

Monadic interpreters come with an alluring 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` ($x \leftarrow \texttt{ret}\ a\ ;;\ k\ x = k\ a$) and to its right ($x \leftarrow ma\ ;;\ \texttt{ret}\ x = ma$). The `bind` operation should additionally be associative ($b \leftarrow (a \leftarrow ma\ ;;\ f\ a)\ ;;\ g\ b = a \leftarrow ma\ ;;\ (b \leftarrow f\ a\ ;;\ g\ b)$).

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 ITree 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 [Xia et al. 2020] establishes the correctness of a compiler. This is achieved by parameterizing `eutt` by a relation on computed values: `eutt` specifies that weak bisimilarity is the prime notion to compare computations, and lifts an arbitrary relation on the computed values in the

$$\frac{Reflexive\ R}{Reflexive\ (\approx_R)}\ \textsc{Refl} \qquad \frac{PER\ R}{PER\ (\approx_R)}\ \textsc{PER}$$

$$\frac{ma \approx_{R_1} mb \quad mb \approx_{R_2} mc}{ma \approx_{R_2 \circ R_1} mc}\ \textsc{RelComp}$$

$$\frac{}{eqmR(\dagger R) \simeq \dagger(eqmR\ R)}\ \textsc{Transpose}$$

$$\frac{ma \approx_{R_1} mb \quad R_1 \subseteq R_2}{ma \approx_{R_2} mb}\ \textsc{Mono}$$

$$\frac{ma \approx_R mb \qquad ma \approx_{R'} mb}{ma \approx_{R \wedge R'} mb}\ \textsc{Conj}$$

$$\frac{m_1 \approx_{RA \otimes RB} m_2}{\mathsf{fmap}\ fst\ m_1 \approx_{RA} \mathsf{fmap}\ fst\ m_2}\ \textsc{ProdFst}$$

$$\frac{m_1 \approx_{RA \otimes RB} m_2}{\mathsf{fmap}\ snd\ m_1 \approx_{RA} \mathsf{fmap}\ snd\ m_2}\ \textsc{ProdSnd}$$

$$\frac{\mathsf{fmap}\ fst\ m_1 \approx_{RA} \mathsf{fmap}\ fst\ m_2 \quad \mathsf{fmap}\ snd\ m_1 \approx_{RB} \mathsf{fmap}\ snd\ m_2}{m_1 \approx_{RA \otimes RB} m_2}\ \textsc{Prod}$$

$$\frac{m_1 \approx_{RA} m_2}{\mathsf{fmap}\ inl\ m_1 \approx_{RA \oplus RB} \mathsf{fmap}\ inl\ m_2}\ \textsc{SumL1}$$

$$\frac{m_1 \approx_{RB} m_2}{\mathsf{fmap}\ inr\ m_1 \approx_{RA \oplus RB} \mathsf{fmap}\ inr\ m_2}\ \textsc{SumR1}$$

$$\frac{m_1 \quad \approx_{RA \oplus RB} \quad m_2 \quad \forall\ a_1\ a_2,\ RA\ a_1\ a_2 \to RC\ (f_1\ a_1)\ (f_2\ a_2) \quad \forall\ b_1\ b_2,\ RB\ b_1\ b_2 \to RC\ (g_1\ b_1)\ (g_2\ b_2)}{\mathsf{fmap}\ (\mathsf{case}\ f_1\ g_1)\ m_1 \approx_{RC} \mathsf{fmap}\ (\mathsf{case}\ f_2\ g_2)\ m_2}\ \textsc{Sum}$$

Fig. 7. OK eqmR Laws (Well-formedness Laws of EqmR)

process. This extension not only allows for relating heterogeneous computations, but provides the foundations for establishing bisimilarity results following a relational program logic style [Benton et al. 2009].

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 $\approx_R$ in lieu of `eqmR R`.

Figure 7 axiomatizes the required behavior for eqmR. Equivalences should be lifted into equivalences, ensuring, in particular, that $\approx_{eq}$ — written $\approx$ in the following — behaves as a suitable tightest notion of equality of computations. We derive this transport from 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 $\dagger R$ for the transposition of R and $\subseteq$ for inclusion of relations).

As mentioned, eqmR is meant to be thought of as the basis of a relational program logic. The indexed relation should therefore be monotone, providing a weakening rule, and ensuring compatibility with the equivalence of relations. From `Mono`, one trivially derives the usual disjunction rule, but the `Conj` rule must be additionally required.

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 computation with respectively a product and a coproduct, eqmR should respect those too—the right column in Figure 7 specifies the necessary introduction and elimination rules.

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

Example 4.1. ITree.

At any signature `E`, `itree` E is known to be a monad. But more specifically, we prove that the strong (`eq_itree`) and weak bisimulation (`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: ($\texttt{stateM}_S\ X \triangleq S \to S * X$). We consider the standard operations: pure computations leave the state untouched while sequencing threads the states.

$$\texttt{ret}^{St}\ v \triangleq \lambda\ s \Rightarrow (s, v) \qquad \texttt{bind}^{St}\ m\ k \triangleq \lambda\ s \Rightarrow \texttt{let}\ (s', a') := m\ s\ \texttt{in}\ f\ a'\ s'$$

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.

$$m_a \approx_R m_b \triangleq \forall s,\ R\ (snd\ (m_a\ s))\ (snd\ (m_b\ s)) \wedge (fst\ (m_a\ s)) = (fst\ (m_b\ s))$$

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 $\texttt{errorM}_E\ R \triangleq E + R$ and equipped with the usual sequencing passing by valid computed value and propagating erroneous states, as depicted to the left in the following:

$$\begin{aligned}\texttt{ret}^{Err}\ v &\triangleq \texttt{inr}\ v\\ \texttt{bind}^{Err}\ m\ k &\triangleq \texttt{match}\ m\ \texttt{with}\\ &\quad \mid \texttt{inl}\ e \Rightarrow \texttt{inl}\ e\\ &\quad \mid \texttt{inr}\ v \Rightarrow k\ v\\ &\quad \texttt{end}\end{aligned} \qquad \begin{aligned}x \approx_R y \triangleq\ &\texttt{match}\ x, y\ \texttt{with}\\ &\mid \texttt{inl}\ \_\ \texttt{inl}\ \_ \Rightarrow \top\\ &\mid \texttt{inr}\ v_1, \texttt{inr}\ v_2 \Rightarrow R\ v_1\ v_2\\ &\mid \_, \_ \Rightarrow \bot\\ &\texttt{end}\end{aligned}$$

To the right is an 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. Nondeterminism.

Nondeterministic computations can be represented as sets of outcomes using Prop: $\texttt{propM}\ R \triangleq R \to \texttt{Prop}$. This propositional account of nondeterminism gives up its computational content, but is in exchange flexible to manipulate, allowing for modelling nondeterminism over infinite sets, as well as for specifying a computation.

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

$$ma \approx_R mb \triangleq (\forall a, ma\ a \to \exists b, mb\ b \wedge R\ a\ b) \wedge (\forall b, mb\ b \to \exists a, ma\ a \wedge R\ a\ b)$$

We highlight the situation of 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 Conj rule and the rules related to the product of relations are invalid.

4.2 Image of Monadic Computations

The proposition $m_a \approx_R m_b$ intuitively asserts that m_a and m_b are compatible computations—weakly bisimilar in the case of ITrees, for instance—and that the relation R is a valid relational postcondition over returned values. The axioms from Figure 7 provide the basis to justify this interpretation of R as a postcondition. 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: $\texttt{ma} \approx_S \texttt{mb} \to (\forall\ \texttt{x}\ \texttt{y},\ \texttt{S}\ \texttt{x}\ \texttt{y} \to \texttt{ka}\ \texttt{x} \approx_R \texttt{kb}\ \texttt{y}) \to \texttt{ma} \gg\!= \texttt{ka} \approx_R \texttt{mb} \gg\!= \texttt{kb}$. This rule mirrors the familiar Hoare-style rule for sequence by quantifying existentially over S, the intermediate postcondition.

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 `itree ChoiceE` where the signature `ChoiceE` provides an event, `choice : ChoiceE bool`, encoding a binary branching indexed by a boolean. The computation `c ≜ b ← trigger choice;; if b then ret 1 else ret 0` triggers this external choice event, and converts the returned boolean into a natural number. Because `c` has type `itree ChoiceE 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 eqmR as a unary logic, we want to define uniformly a notion akin to a strongest postcondition 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 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. Thus we seek to associate to each computation `m : M A` a predicate `image m : A → Prop` over its return type. Concretely, for an ITree `t`, `image t` should be the set of leaves of `t`; for a nondeterministic 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 a given monad, a necessity for incorporating it into general structural laws.

A first intuition on this path is to consider the diagonal of eqmR: suppose `m: M 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 postcondition, all pairs of returned values should belong to `R`, and the square of the `image m`, *i.e.*, $\{ (a, a) \mid \texttt{image m a} \}$ 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:

$$\texttt{image}\ m\ v \stackrel{?}{=} \forall\ R, m \approx_R m \longrightarrow R\ v\ v$$

Interestingly enough, this first attempt turns out to be too naïve when considering nondeterminism; 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 : propM bool)` be the computation that nondeterministically returns a boolean, that is, `m = fun x ⇒ x = true ∨ x = false`. We naturally expect the image of `m` to contain both booleans as well. However, by taking for relation `R = {(true,false);(false,true)}`, we easily see that $m \approx_R m$ holds despite `R`'s diagonal being empty!

Intuitively, considering all relations is inadequate: 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 ITree representing a silently diverging computation self-relates at all postconditions, we would 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:

$$\texttt{imageH}\ m\ a_1\ a_2 \triangleq \forall\ R, PER\ R \longrightarrow m \approx_R m \longrightarrow R\ a_1\ a_2 \qquad \texttt{image}\ m\ a \triangleq \texttt{imageH}\ m\ a\ a$$

We write $a \in m$ as a short-hand for `image` $m\ a$.

4.2.2 Image for `itree`*: A Concrete Characterization.* The semantic characterization of the image of a monadic computation only relies on eqmR, and can be leveraged to axiomatize the desired theory. However, this definition gives few reasoning principles. Therefore, we prove that in the case of the ITree 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:

```
Inductive Leaves {E} {A: Type} (a: A) : itree E A → Prop :=
| LeavesRet: ∀t, t ≈ Ret a → Leaves a t
| LeavesTau: ∀t u, t ≈ Tau u → Leaves a u → Leaves a t
| LeavesVis: ∀{X} (e: E X) (x: X) t k, t ≈ Vis e k → Leaves a (k x) → 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. $a \in ma \iff \texttt{Leaves}\ a\ ma$

4.2.3 The Case of `stateM`*.* We prove that the image of a $\texttt{stateM}_S$ 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 postcondition, it could be hoped that the following rule universally hold:

$$\frac{u \in m \qquad v \in (k\ u)}{v \in (m\ ;;\ k)}\ \textsc{ImageBind}$$

However, this would be misunderstanding what the image expresses: while it universally captures the tightest postcondition 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 postcondition 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 ≜ fun b ⇒ if b then (true, true) else (true, false)
```

The image of `m` contains both booleans since either can be found as returned value for a certain initial state. In particular, `false` ∈ `m`. Now consider the continuation `k ≜ fun v b ⇒ (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, `false` ∈ `k false`. The branches carrying `false` are, however, reachable only from a state set at `false`: since `m` does not return such state, they are unreachable branches. To sum up, `false` ∈ `m`, `false` ∈ `(k false)`, but yet `false` ∉ `(m ≫= 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.

$$\frac{}{x \leftarrow \texttt{ret}\ a\ ;;\ f\ x \approx f\ a}\ \textsc{RetL} \qquad \frac{}{x \leftarrow ma\ ;;\ \texttt{ret}\ x \approx ma}\ \textsc{RetR} \qquad \frac{R_A\ a_1\ a_2}{\texttt{ret}\ a_1 \approx_{R_A} \texttt{ret}\ a_2}\ \textsc{Ret}$$

$$\frac{}{(a \leftarrow ma\ ;;\ b \leftarrow f\ a)\ ;;\ g\ b \approx a \leftarrow ma\ ;;\ (b \leftarrow f\ a\ ;;\ g\ b)}\ \textsc{BindAssoc}$$

$$\frac{ma_1 \approx_{R_A} ma_2 \qquad \forall a_1, a_2, R_A\ a_1\ a_2 \to k_1\ a_1 \approx_{R_B} k_2\ a_2}{x \leftarrow ma_1\ ;;\ k_1\ x \approx_{R_B} x \leftarrow ma_2\ ;;\ k_2\ x}\ \textsc{Bind} \qquad \frac{}{ma \approx_{(\texttt{imageH}\ ma)} ma}\ \textsc{ImageSelf}$$

$$\frac{ma_1 \approx_{R_A} ma_2 \qquad a_1 \in ma_1}{\exists\ a_2, R_A\ a_1\ a_2 \wedge a_2 \in ma_2}\ \textsc{ImageL} \qquad \frac{ma_1 \approx_{R_A} ma_2 \qquad a_2 \in ma_2}{\exists\ a_1, R_A\ a_1\ a_2 \wedge a_1 \in ma_1}\ \textsc{ImageR}$$

Fig. 8. EqmRMonad Laws

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. $a \in ma \iff ma = \mathsf{inr}\ a$.

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 ImageL and ImageR 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 postcondition. Rule ImageSelf ensures part of the intuition we started from: the image should capture all possibly returned values, it should therefore itself be a valid postcondition when self-relating. As illustrated in Section 4.2.3, the ImageBind 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.

$$\frac{ma_1 \approx_{R_A} ma_2 \qquad \forall a_1, a_2, a_1 \in ma_1 \to a_2 \in ma_2 \to R_A\ a_1\ a_2 \to k_1\ a_1 \approx_{R_B} k_2\ a_2}{x \leftarrow ma_1\ ;;\ k_1\ x \approx_{R_B} x \leftarrow ma_2\ ;;\ k_2\ x}\ \textsc{CloBindGen}$$

We illustrate the adequacy of this axiomatization by providing instances.

Lemma 4.5. *The* `itree`, $\texttt{stateM}_S$, $\texttt{errorM}_E$ *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.

$$\frac{\texttt{ret}\ a_1 \approx_{RA} \texttt{ret}\ a_2}{RA\ a_1\ a_2}\ \textsc{RetInv} \qquad \frac{\texttt{fmap}\ f\ ma \approx_R \texttt{fmap}\ g\ mb}{ma \approx_{\lambda a\, b, R\ (f\ a)\ (g\ b)} mb.}\ \textsc{FmapInv} \qquad \frac{b \in x \leftarrow ma\ ;;\ k\ x}{\exists a, a \in ma \wedge b \in k\ a}\ \textsc{BindImageInv}$$

Fig. 9. EqmRMonadInverses Laws

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 postcondition while the FmapInv rule expresses that when `fmap` computations are related, the mapped functions can be pushed down the postcondition.

Interestingly, while we have seen that forward compatibility of the image with `bind` is invalid in monads as common as $\texttt{stateM}_S$, backward compatibility is always valid. If we know that a value is in the image of a `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`, $\texttt{stateM}_S$, $\texttt{errorM}_E$ *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,* $a \in \texttt{ret}\ b \leftrightarrow a = b$.

We have carefully defined the semantic definition of `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 `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:

Lemma 4.9. *Over the* `itree`, $\texttt{stateM}_S$ *and* $\texttt{errorM}_E$ *monads, the diagonal of any self-relating postcondition contains the image:*

$$\frac{ma \approx_R ma \quad a \in ma}{R\ a\ a}\ \textit{BindImageInv}$$

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:

Lemma 4.10. *Over the* `itree` *monad, the following rule, inverting relations between* `bind`*s sharing a common prefix, holds true:*

$$\frac{t \gg\!= k_1 \approx_R t \gg\!= k_2 \quad r \in t}{k_1\ r \approx_R k_2\ r}\ \textit{BindInv}$$

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

4.4 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 manually re-establish such a theory for each structure they consider. To alleviate this painful work, we provide the tools to systematically transport the eqmR structure and its theory via appropriate monad transformers. We spell out all the requirement through an additional series of typeclasses.

$$\frac{R\ a\ b}{\texttt{lift}\ (\texttt{ret}\ a) \approx_R \texttt{ret}\ b}\ \textsc{LiftRet} \qquad \frac{ma \approx_{RA} ma' \qquad \forall a\ a', RA\ a\ a' \to k\ a \approx_{RB} k'\ a'}{\texttt{lift}\ (x \leftarrow ma;;k\ x) \approx_{RB} x \leftarrow \texttt{lift}\ ma';;\texttt{lift}\ (k'\ x)}\ \textsc{LiftBind}$$

$$\frac{ma \approx_R mb}{\texttt{lift}\ ma \approx_R \texttt{lift}\ mb}\ \textsc{LiftEqmR}$$

Fig. 10. Monad morphism laws

$$\frac{\mathsf{EqmROK}\ M}{\mathsf{EqmROK}\ (\texttt{lift}\ M)}\ \mathrm{OK} \qquad \frac{\mathsf{EqmRMonad}\ M}{\mathsf{EqmRMonad}\ (\texttt{lift}\ M)}\ \textsc{OKMon} \qquad \frac{\mathsf{EqmRMonadInverses}\ M}{\mathsf{EqmRMonadInverses}\ (\texttt{lift}\ M)}\ \textsc{OKInv}$$

Fig. 11. Monad transformer well-formedness conditions

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

$$\texttt{lift} : \forall (M : \mathsf{Type} \to \mathsf{Type})(A : \mathsf{Type}), M\ A \to MT\ M\ A$$

For any monad M, `lift` M must define a monad morphism, *i.e.*, an indexed function commuting with the `ret` *a*nd `bind` operations. While standard, these two laws, depicted on the upper part of Figure 10[5], must be stated with respect to eqmR in our setup. Furthermore, `lift` must itself respect 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 $\texttt{stateT}_S\ M\ A \triangleq S \to M\ (S \times A)$. The return and bind definitions are standard, mirroring the ones from $\texttt{stateM}_S$, but leveraging the underlying monadic operations: $ret\ v \triangleq \lambda\ s \Rightarrow \texttt{ret}\ (v, s)$ and $bind\ c\ k \triangleq \lambda\ s \Rightarrow (v, s) \leftarrow c\ s\ ;;\ k\ v\ s$. The lift operator relies on the underlying fmap to reinject the unchanged state: $\texttt{lift}\ c \triangleq \lambda\ s \Rightarrow \mathsf{fmap}\ (\lambda\ x \Rightarrow (s, x))\ c$.

We parameterize the definition of eqmR at any monad transformed by the state transform by a relation on states R_{st}. The relation on computations is then obtained by lifting pointwise R_{st} and a relation on values through the underlying eqmR— it is reminiscent of Goubault-Larrecq's characterization [Goubault-Larrecq et al. 2008] of the state transformer logical relation, and Maillard's relational Dijkstra monads [Maillard et al. 2020]:

$$ma \approx_R mb \triangleq \forall s_1\ s_2,\ (s_1, s_2) \in R_{St} \to ma\ s_1 \approx_{R_{st} \otimes R} mb\ s_2$$

where $R \otimes R'$ is defined as $\lambda\ x\ y \Rightarrow R\ x\ y \wedge R'\ x\ y$. 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 postcondition the product of both relations.

The image admits a similar characterisation as in the case of $\texttt{stateM}_S$, but expressed in terms of the underlying notion of image:

Lemma 4.11. $a \in ma \iff \exists\ s\ s',\ (ma\ s) \in (s', a)$

We prove that $\texttt{stateT}_S$ 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 [Xia et al. 2020].

[5]We omit the implicit argument M in this Figure, writing `lift` in lieu of `lift` M.

$$\frac{}{\texttt{interp}\ h\ (\texttt{ret}\ x) \approx \texttt{ret}\ x}\ \textsc{InterpRet} \qquad \frac{}{\texttt{interp}\ h\ (x \leftarrow t\ ;;\ k\ x) \approx x \leftarrow \texttt{interp}\ h\ t ;; \texttt{interp}\ h\ (k\ x)}\ \textsc{InterpBind}$$

$$\frac{}{\texttt{interp}\ h_E\ (\texttt{trigger}\ e_E) \approx \texttt{lift}\ (h_E\ e_E)}\ \textsc{InterpTrigger} \qquad \frac{\forall\ j, \texttt{interp}\ h\ (f\ j) \approx f'\ j}{\texttt{interp}\ h\ (\texttt{iter}\ f\ i) \approx \texttt{iter}\ f'\ i}\ \textsc{InterpIter}$$

$$\frac{F +? G \prec H \qquad E +? H \prec I}{\texttt{interp}_I\ (\texttt{over}\ h_E)\ (\texttt{trigger}\ e_F) \approx \texttt{lift}\ (\texttt{trigger}\ e_F)}\ \textsc{IgnoreTrigger}$$

Fig. 12. Interpretation Laws (we write h_E for a handler of type $E \rightsquigarrow M$ and e_E for an event of E)

Example 4.6. `errorT` . Similarly, we support the standard generalization of `errorM` to a monad transformer: $\texttt{errorT}_E\ M\ A \triangleq M(E + A)$. The return and bind operators, omitted here, are standard. The lift simply injects the result of the underlying computation into a successful one: lift $\triangleq \lambda\ ma \Rightarrow$ fmap inr ma. Finally, eqmR is parameterized by an arbitrary relation over errors and feeds the coproduct of both relations to the underlying eqmR: $ma \approx_R mb \triangleq ma \approx_{R_{exn} \oplus R} mb$.

We prove that $\texttt{errorT}_E$ preserves all interfaces. The characterization of the image still holds: they are the successful elements of the underlying image.

Lemma 4.12. $a \in ma \iff (\texttt{inr}\ a \in ma)$

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 ITree library's notion of `eutt`. However, in many situations—*e.g.*, when expressing the correctness of a pass of compilation, such as for the IMP-to-ASM compiler (see Section 6)—we need to work across languages and 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 (`A → B → Prop`) to relations at computation-level across monads (`M A → N B → 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 structural properties for the `Interp` typeclass and 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 the equivalence of two computations defined by iteration.

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.
```

One constraint imposed by this definition is that `M` is an `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, `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 [Johann and Ghani 2009]); and we adapt those definitions for use in our setting. Select higher-order functor laws are presented in Figure 13, which shows that `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 `ret`, `bind`, `iter`, and `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 ⋙ is function composition. The structural properties MonadT, IterativeMonadT, and so on, correspond to the typeclasses that capture the structures and well-formedness laws. The complete set of hfmap-related laws and the details of these typeclasses is included in the Coq development.

Although there is a fair amount of effort needed 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 the interpretation laws, iterative monad laws, and higher-order functor laws 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), and a case study of an IMP to ASM compiler to see how eqmR can be used in practice.

$$\frac{}{\texttt{hfmap}\ (\lambda x \Rightarrow x)\, t \approx t}\ \textsc{HfmapId} \qquad \frac{}{\texttt{hfmap}\ (f_1 \ggg f_2)\ t \approx (\texttt{hfmap}\ f_1 \ggg \texttt{hfmap}\ f_2)\ t}\ \textsc{HfmapComp}$$

$$\frac{\mathsf{MonadMorphism} f}{\mathsf{MonadMorphism}(\texttt{hfmap} f)}\ \textsc{HfmapNat}$$

Fig. 13. Select HFunctor Laws

$$\frac{\mathsf{MonadT}\ S \qquad \mathsf{MonadT}\ T}{\mathsf{MonadT}\ (S \ggg T)}\ \textsc{ComposeMonadT} \qquad \frac{\begin{array}{c}\mathsf{IterativeMonadT}\ S \\ \mathsf{IterativeMonadT}\ T\end{array}}{\mathsf{IterativeMonadT}\ (S \ggg T)}\ \textsc{ComposeIterativeMonadT}$$

$$\frac{\mathsf{HFunctor}\ S \quad \mathsf{HFunctor}\ T}{\mathsf{HFunctor}\ (S \ggg T)}\ \textsc{ComposeHFunctor} \qquad \frac{\begin{array}{c}\mathsf{IterativeMonadTLaws}\ S \\ \mathsf{IterativeMonadTLaws}\ T\end{array}}{\mathsf{IterativeMonadTLaws}\ (S \ggg T)}\ \textsc{ComposeIterativeMonadTLaws}$$

$$\frac{\begin{array}{c}\mathsf{HFunctorLaws}\ S \\ \mathsf{HFunctorLaws}\ T\end{array}}{\mathsf{HFunctorLaws}\ (S \ggg T)}\ \textsc{ComposeHFunctorLaws}$$

Fig. 14. Composable Structures and Laws

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 (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.

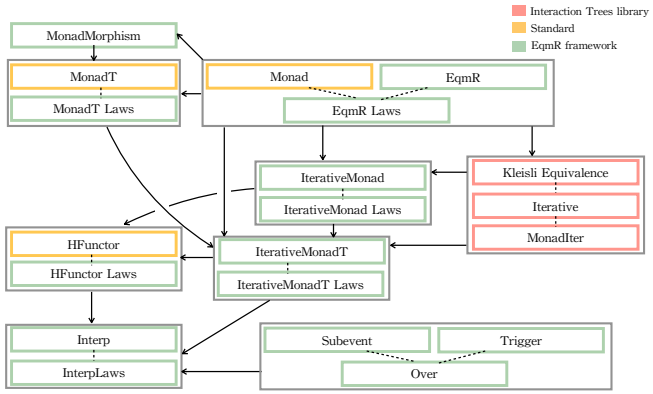

Fig. 15. Typeclass dependencies in the EqmR framework

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 `fun x ⇒ 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 to not 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.

```
Ltac ibind_body TR h t k :=
  match type of h with
  | ∀ T : Type, ?E T → _ T ⇒
     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_
        ⇒
     ibind_rec TR h x_
  | bind ?t ?k ⇒
     ibind_body TR h t k
  end.

Ltac ibind :=
  match goal with
  | ⊢ eqmR _
     (interp (T := ?TR) ?h ?x)
     _ ⇒
     ibind_rec TR h x
  end.
```

Fig. 16. Custom tactics for using the InterpBind rule

The custom tactics for using the InterpBind rule are shown in Figure 16. 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 to apply 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 [Xia et al. 2020], 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 Elegant Staged Interpretations : the ASM Example. Recall from Section 3.3 how the automatic injection of handlers eliminates the need for extra manual annotations. 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.

```
(* [interp_asm] definition without [over] and [interp] *)
Definition interp_asm {E A} (t : itree (Reg +' Memory +' E) A) :=
  let h := bimap h_reg (bimap h_memory (id_ _)) in
  let t' := interp h t in fun mem regs ⇒ interp_map (interp_map t' regs) mem.
```

The bimap h_reg (bimap h_memory (id_ _)) manually injects the handlers for Reg and Memory for the signature of the program itree (Reg +' Memory +' E) A. The application of interp_map and 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 Reg +' Memory +' E for this interp_asm to apply.

The code below is the staged interpretation using over and the new interp combinators.

```
Definition interp_asm {D E F A} `{Reg +? D -< E} `{Memory +? E -< F} (t : itree F A) :=
  (interp (T := stateT memory) (over handle_reg) (interp (T := fun x ⇒ x) (over 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 `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 InterpRet law with layers `stateT reg (stateT memory (itree E))`, an instance would need to be proven for each of `itree E`, `stateT memory (itree E)`, and `stateT reg (stateT memory (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.

This illustrates how definitions can be simplified. What about proofs? Consider the following structural lemma stating how the interpretation function for ASM commutes with `bind`.

```
Lemma interp_asm_bind: ∀{R S} (t: itree E R) (k: R → itree E S),
    interp_asm (bind t k) ≈ (x ← interp_asm t ;; interp_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 (`interp_bind` and `interp_state_bind`), which are specific to the layers and cannot be composed with each other. In addition, it used the `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 'irewrite 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.

```
OLD PROOF:
  intros.
  unfold interp_asm, interp_map.
       cbn.
  repeat rewrite interp_bind.
  repeat rewrite interp_state_bind.

  repeat rewrite bind_bind.
  eapply eutt_clo_bind; [
       reflexivity | ..].
  intros. rewrite H.
  destruct u2 as [g' [l' x]].
  reflexivity.

NEW PROOF:
  intros; unfold interp_asm.
  do 2 ibind.
```

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 instances where the IgnoreTrigger rule applies at a different position in the stack.[6] 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.

[6]The supplementary Coq material contains the proofs.

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 [Moggi 1989, 1991] about using monads to characterize imperative features in a pure, functional setting was consequently popularized by Wadler [Wadler 1990] and Peyton Jones [Peyton Jones and Wadler 1993]. Monad transformers [Moggi 1990] were then adopted as a way of composing various effects: notably Liang, *et al.* [Liang et al. 1995] showed how to build modular interpreters in that style, which we have adopted and formalized here. Swierstra [Swierstra 2008], Apfelmus [Apfelmus 2010], Kiselyov, *et al.* [Kiselyov et al. 2013], and Kiselyov and Ishii [Kiselyov and Ishii 2015] have showed how to use the *free* and *freer monads* to define modular monad instances. A related data structure to freer monads are Tlön Embeddings [Li and Weirich 2022], which use *program adverbs* as a basis to allow more flexibility in computational modeling of effects. Interaction Trees [Xia et al. 2020] are a coinductive freer monad, 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 this 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 [Johann and Ghani 2009]. 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 [Bauer and Pretnar 2013; Hyland et al. 2006; Plotkin and Pretnar 2009; Plotkin and Power 2003; Plotkin and Pretnar 2013] 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 [Bauer and Pretnar 2015], are working in an ambient environment that allows nontermination, whereas ITrees are crafted to fit with the total semantics of Coq. 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 termination checker would not be able to observe that manipulations of the continuation are sufficiently guarded to define valid cofixpoints).

The algebraic reasoning of algebraic effects is often used in that setting to characterize equations that hold for *particular* effects; for instance, for state effects the sequence `put s; put t` is equivalent to `put t` (since the second `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 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 [Benton 2004] and Nanevski, *et al.*'s Relational Hoare Type Theory [Nanevski et al. 2013] have been shown to be useful for reasoning about program transformations and properties such as information flow. The subsequent work on predicate transformers [Swierstra and Baanen 2019], Dijkstra Monads [Ahman et al. 2017; Maillard et al. 2019, 2020], and F* [Swamy et al. 2016] 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 [Ahmed 2004; Benton et al. 2009; Dreyer et al. 2009] and bisimulations [Koutavas and Wand 2006; Lago et al. 2017; Sangiorgi 2012], and their combination [Hermida et al. 2020; Hur et al. 2012]. The eqmR framework defines relations for monads that technically form logical relations. Logical relations over monadic types were developed by Goubault-Larrecq, *et al.* [Goubault-Larrecq et al. 2008], giving a sound basis for a monadic equivalence akin to a notion of bisimilarity. As in our approach, these techniques build an expressive basis for program logics and program verification [Jung et al. 2015].

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 [Mitchell 1996, Chapter 5], for reasoning about operational equivalences [Appel and McAllester 2001], or for giving different *views* of information [Abadi et al. 1999]. 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 and principled approach to the construction of monadic interpreters built in layers from a free(r) 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, and 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 ITree projects, lifting the equational theory through state and error transformers. However, there is a zoo of monads: expanding the library to cover additional effects and monads would be a valuable extension.

Among those effects, nondeterminism is particularly worthy of attention. Indeed, we have presented in Section 4 the `propM` monad, but looking back at Figure 2, the Vellvm developers use a more general structure that models not just nondeterministic sets of *values*, but rather sets of *computations*. This approach relies on a hypothetical `propM` transformer, $\texttt{propT}_M\ A \triangleq M\ A \to \texttt{Prop}$. For Vellvm, the authors have fixed M to be `itree E`, defining a new ad hoc structure rather than a transformer. Interestingly, the reason for this was because they lacked a generic notion of image for a monadic computation, which we introduced through this paper. Indeed, they define the bind over this structure as: $\texttt{bind}\ P\ K\ t_b \triangleq \exists\ t_a,\ k,\ P\ t_a \wedge t_b \approx \texttt{bind}\ t_a\ k \wedge (\forall\ a,\ \texttt{Leaves}\ a\ t_a \to K\ a\ (k\ a))$. 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 `propT` here is that, even using the image, $\texttt{propT}_M$ doesn't lift the monad laws properly—this definition of `bind` does not associate to the left (an expected artifact of the nondeterminism [Maillard et al. 2020]). Thus, developing clean nondeterministic transformers for monadic interpreters remains an interesting prospect.

With these extensions to the ITree library in place, we lighten the bureaucratic boilerplate of structural rules when programming and reasoning about nested monadic interpreters. Of course, monadic reasoning can be about more than its structural rules: the extensions in this paper express nothing about monad-specific algebras. This aspect of the reasoning is still fairly ad hoc in the ITree landscape; combining our contribution with other techniques, such as Dijkstra monads [Maillard et al. 2019, 2020], would likely further improve the viability of relational reasoning for monadic interpreters at scale.

ACKNOWLEDGMENTS

We thank Paul He for reading and providing comments on our paper, and we thank the anonymous reviewers for their feedback. This material is based upon work supported by the National Science Foundation under Grant No. 1521539 and the ONR under Grant No. N00014-17-1-2930. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or the ONR.

DATA AVAILABILITY STATEMENT

The ideas in this paper are packaged as a Coq development [Yoon et al. 2022] and all of the properties presented here have been proved in Coq. The accompanying material is available at: https://zenodo.org/record/6913915.

REFERENCES

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 147–160. https://doi.org/10.1145/292540.292555

Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra Monads for Free. *SIGPLAN Not.* 52, 1 (jan 2017), 515–529. https://doi.org/10.1145/3093333.3009878

Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State.* Ph. D. Dissertation. USA. AAI3136691.

Heinrich Apfelmus. 2010. The Operational Monad Tutorial. *The Monad.Reader* Issue 15 (2010).

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (sep 2001), 657–683. https://doi.org/10.1145/504709.504712

Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* 10. https://doi.org/10.1007/978-3-642-40206-7_1

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001 Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.

Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. *SIGPLAN Not.* 39, 1 (jan 2004), 14–25. https://doi.org/10.1145/982962.964003

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. 2009. Relational Semantics for Effect-Based Program Transformations: Higher-Order Store. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (Coimbra, Portugal) *(PPDP '09)*. Association for Computing Machinery, New York, NY, USA, 301–312. https://doi.org/10.1145/1599410.1599447

Stephen L. Bloom and Zoltán Ésik. 1993. *Iteration Theories - The Equational Logic of Iterative Processes.* Springer. https://doi.org/10.1007/978-3-642-78034-9

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* Volume 1, Issue 2 (July 2005). https://doi.org/10.2168/LMCS-1(2:1)2005

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical Step-Indexed Logical Relations. In *Proceedings of the 2009 24th Annual IEEE Symposium on Logic In Computer Science (LICS '09)*. IEEE Computer Society, USA, 71–80. https://doi.org/10.1109/LICS.2009.34

Jean Goubault-Larrecq, Slawomir Lasota, and David Nowak. 2008. Logical relations for monadic types. *Math. Struct. Comput. Sci.* 18, 6 (2008), 1169–1217. https://doi.org/10.1017/S0960129508007172

Peter Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic.* Springer-Verlag, Berlin, Heidelberg, 317–331. https://doi.org/10.1007/3-540-44622-2_21

Claudio Hermida, Uday S. Reddy, Edmund P. Robinson, and Alessio Santamaria. 2020. Bisimulation as a Logical Relation. *CoRR* abs/2003.13542 (2020). arXiv:2003.13542 https://arxiv.org/abs/2003.13542

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The marriage of bisimulations and Kripke logical relations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 59–72. https://doi.org/10.1145/2103656.2103666

Martin Hyland, Gordon Plotkin, and John Power. 2006. Combining effects: Sum and tensor. *Theoretical Computer Science* 357, 1 (2006), 70 – 99. https://doi.org/10.1016/j.tcs.2006.03.013 Clifford Lectures and the Mathematical Foundations of Programming Semantics.

Patricia Johann and Neil Ghani. 2009. A principled approach to programming with nested types in Haskell. *High. Order Symb. Comput.* 22, 2 (2009), 155–189. https://doi.org/10.1007/s10990-009-9047-7

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. *SIGPLAN Not.* 50, 12 (aug 2015), 94–105. https://doi.org/10.1145/2887747.2804319

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, Chung-chieh Shan (Ed.). ACM, 59–70. https://doi.org/10.1145/2503778.2503791

Vasileios Koutavas and Mitchell Wand. 2006. Small bisimulations for reasoning about higher-order imperative programs. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 141–152. https://doi.org/10.1145/1111037.1111050

Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy. 2017. Effectful applicative bisimilarity: Monads, relators, and Howe's method. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005117

Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. 2018. Modular Verification of Programs with Effects and Effect Handlers in Coq. In *FM 2018 - 22nd International Symposium on Formal Methods (LNCS, Vol. 10951)*. Springer, Oxford, United Kingdom, 338–354. https://doi.org/10.1007/978-3-319-95582-7_20

Yao Li and Stephanie Weirich. 2022. Program Adverbs and Tlön Embeddings. *Proc. ACM Program. Lang.* 3, ICFP, Article 101 (2022). https://doi.org/10.1145/3547632

Sheng Liang and Paul Hudak. 2000. Modular Denotational Semantics for Compiler Construction. *Lecture Notes in Computer Science* 1058 (05 2000). https://doi.org/10.1007/3-540-61055-3_39

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. *Proc. ACM Program. Lang.* 3, ICFP, Article 104 (jul 2019), 29 pages. https://doi.org/10.1145/3341708

Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 Relational Program Logics. *Proc. ACM Program. Lang.* 4, POPL, Article 4 (dec 2020), 33 pages. https://doi.org/10.1145/3371072

Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9129)*, Ralf Hinze and Janis Voigtländer (Eds.). Springer, 257–275. https://doi.org/10.1007/978-3-319-19797-5_13

John C. Mitchell. 1996. *Foundations for Programming Languages*. The MIT Press.

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 14–23. https://doi.org/10.1109/LICS.1989.39155

Eugenio Moggi. 1990. *An Abstract View of Programming Languages*. Technical Report ECS-LFCS-90-113. Laboratory for the Foundations of Computer Science, University of Edinburgh.

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

Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. 2013. Dependent Type Theory for Verification of Information Flow and Access Control Policies. *ACM Trans. Program. Lang. Syst.* 35, 2 (2013), 6:1–6:41. https://doi.org/10.1145/2491522.2491523

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/158511.158524

Maciej Piróg and Jeremy Gibbons. 2014. The Coinductive Resumption Monad. In *Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA, June 12-15, 2014 (Electronic Notes in Theoretical Computer Science, Vol. 308)*, Bart Jacobs, Alexandra Silva, and Sam Staton (Eds.). Elsevier, 273–288. https://doi.org/10.1016/j.entcs.2014.10.015

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (York, UK) *(ESOP '09)*. Springer-Verlag, Berlin, Heidelberg, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Appl. Categorical Struct.* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon D Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (Dec. 2013). https://doi.org/10.2168/LMCS-9(4:23)2013

Davide Sangiorgi. 2012. *Introduction to Bisimulation and Coinduction* (2nd ed.). Cambridge University Press, USA.

Bas Spitters and Eelis van der Weegen. 2011. Type classes for mathematics in type theory. *Math. Struct. Comput. Sci.* 21, 4 (2011), 795–825. https://doi.org/10.1017/S0960129511000119

Guy L. Steele. 1994. Building Interpreters by Composing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 472–492. https://doi.org/10.1145/174675.178068

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. https://doi.org/10.1145/2837614.2837655

Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Wouter Swierstra and Tim Baanen. 2019. A Predicate Transformer Semantics for Effects (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 103 (July 2019), 26 pages. https://doi.org/10.1145/3341707

Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. Association for Computing Machinery, New York, NY, USA, 61–78. https://doi.org/10.1145/91556.91592

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees. *Proceedings of the ACM on Programming Languages* 4, POPL (2020). https://doi.org/10.1145/3371119

Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022. *Formal Reasoning About Layered Monadic Interpreters*. https://doi.org/10.5281/zenodo.6913915

Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (aug 2021), 30 pages. https://doi.org/10.1145/3473572

Vadim Zaliva, Ilia Zaichuk, and Franz Franchetti. 2020. Verified Translation Between Purely Functional and Imperative Domain Specific Languages in HELIX. In *Proceedings of the 12th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. https://doi.org/10.1007/978-3-030-63618-0_3

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2103621.2103709