Interaction Trees
Representing Recursive and Impure Programs in Coq

LI-YAO XIA, University of Pennsylvania, USA
YANNICK ZAKOWSKI, University of Pennsylvania, USA
PAUL HE, University of Pennsylvania, USA
CHUNG-KIL HUR, Seoul National University, Republic of Korea
GREGORY MALECHA, BedRock Systems, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA
STEVE ZDANCEWIC, University of Pennsylvania, USA

Interaction trees (ITrees) are a general-purpose data structure for representing the behaviors of recursive programs that interact with their environments. A coinductive variant of "free monads," ITrees are built out of uninterpreted events and their continuations. They support compositional construction of interpreters from event handlers, which give meaning to events by defining their semantics as monadic actions. ITrees are expressive enough to represent impure and potentially nonterminating, mutually recursive computations, while admitting a rich equational theory of equivalence up to weak bisimulation. In contrast to other approaches such as relationally specified operational semantics, ITrees are executable via code extraction, making them suitable for debugging, testing, and implementing software artifacts that are amenable to formal verification.

We have implemented ITrees and their associated theory as a Coq library, mechanizing classic domain- and category-theoretic results about program semantics, iteration, monadic structures, and equational reasoning. Although the internals of the library rely heavily on coinductive proofs, the interface hides these details so that clients can use and reason about ITrees without explicit use of Coq’s coinduction tactics.

To showcase the utility of our theory, we prove the termination-sensitive correctness of a compiler from a simple imperative source language to an assembly-like target whose meanings are given in an ITree-based denotational semantics. Unlike previous results using operational techniques, our bisimulation proof follows straightforwardly by structural induction and elementary rewriting via an equational theory of combinators for control-flow graphs.

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:

Authors’ addresses: Li-yao Xia, University of Pennsylvania, Philadelphia, PA, USA; Yannick Zakowski, University of Pennsylvania, Philadelphia, PA, USA; Paul He, University of Pennsylvania, Philadelphia, PA, USA; Chung-Kil Hur, Seoul National University, Seoul, Republic of Korea; Gregory Malecha, BedRock Systems, Boston, MA, USA; Benjamin C. Pierce, University of Pennsylvania, Philadelphia, PA, USA; Steve Zdancewic, University of Pennsylvania, Philadelphia, PA, USA.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/1-ART51
https://doi.org/10.1145/3371119

1 INTRODUCTION

Machine-checked proofs are now feasible at scale, for real systems, in a wide variety of domains, including programming language semantics and compilers [Kumar et al. 2014; Leroy 2009, etc.], operating systems [Gu et al. 2016; Klein et al. 2009, etc.], interactive servers [Koh et al. 2019, etc.], databases [Malecha et al. 2010, etc.], and distributed systems [Hawblitzel et al. 2015; Wilcox et al. 2015, etc.], among many others. Common to all of these is the need to model and reason about interactive, effectful, and potentially nonterminating computations. For this, most work to date has relied either on operational semantics, represented as (small- or large-step) transition relations defined on syntax, or on trace models, implemented as predicates over lists or streams of observable events. These representations have their advantages: they are expressive, since nearly any semantic feature can be modeled by transition systems or traces when combined with appropriate logical predicates; and they fit smoothly with inductive reasoning principles that are well supported by interactive theorem provers. But they also have significant drawbacks. Operational semantics aren’t very compositional, often requiring auxiliary syntactic constructs (such as program counters, substitution functions, or evaluation contexts) to specify desired behavior; such syntactic clutter makes proofs unwieldy and brittle. Moreover, relational specifications are not executable, which precludes running the model, either for testing or as a reference implementation.

We propose a new alternative called interaction trees (ITrees), a general-purpose data structure and accompanying theory for modeling recursive, effectful computations that can interact with their environment. ITrees allow us to give denotational semantics for effectful and possibly nonterminating computations in Gallina, the specification language of Coq [2018], despite Gallina’s strong purity and termination constraints. Such “shallow” representations abstract away many syntactic details and reuse metalanguage features such as function composition and substitution rather than defining them again, making this approach inherently more robust to changes than relational “deep” embeddings. Moreover, ITrees work well with Coq’s extraction capabilities, making it compatible with tools such as QuickChick [Lampropoulos and Pierce 2018] for testing and allowing us to easily link the extracted code against non-Coq components such as external libraries, so that we can directly execute systems modeled using ITrees. This combination of features makes ITrees a good foundation for formal verification of interactive systems.

The problem of representing effectful programs in pure functional languages is nearly as old as functional programming itself. Our design for interactions trees and their accompanying theory draws heavily on a large body of prior work, ranging from monadic interpreters [Moggi 1989; Steele 1994] to free monads [Swierstra 2008] and algebraic effects [Plotkin and Power 2003]. Our core data structure, the ITree datatype itself, is a coinductive variant of the free monad. Related structures have been studied previously as the program monad in the FreeSpec [Letan et al. 2018] system in Coq, I/O-trees [Hancock and Setzer 2000] and the general monad [McBride 2015] in dependent type theory, and the freer monad [Kiselyov and Ishii 2015] in Haskell. ITrees are a natural generalization of Capretta’s [2005] delay monad and are an instance of resumption monads [Piróg and Gibbons 2014], which have been extensively studied from a category-theoretic point of view. Section 8 gives a thorough comparison of ITrees with these and other related approaches.

The use of a coinductive rather than an inductive structure represents a significant shift in expressiveness, enabling ITrees to represent nonterminating computations without needing to resort to step-indexed approaches such as fuel. Further, by including a “silent effect” (Tau), ITrees can express silently diverging computations and avoid the non-compositionality of guardedness conditions within Coq. After the fact, we can quotient ITrees by these silent steps, providing a generic definition of weak bisimulation. This also enables us to mechanize classic results from the theory of iteration [Bloom and Ésik 1993], which, to our knowledge, have not previously been
applied in the context of machine-checked formalizations. At a practical level, this means that we can easily define semantics for mutually recursive components of an interactive system and reason about them compositionally.

Using a coinductive structure comes with some practical tradeoffs. In particular, Coq’s evaluation of coinductive terms is driven by context, rather than the term itself, which means that proofs must rely on explicit (or tactic-driven) rewriting. Coq’s support for coinductive proofs is also notoriously limited [Hur et al. 2013], but we have gone to some pains to encapsulate the use of coinduction behind the ITrees library interface. Users of the ITrees library should rarely, if ever, need to write their own coinductive proofs.

Contributions. Our main contribution is the design and implementation of a library that enables formal modeling and reasoning about interactive, effectful, and potentially nonterminating computations. Though it rests on a rich body of existing theory, our work is the first to simultaneously address four significant challenges. (1) It focuses on coinductively defined trees whose representation is compatible with program extraction. (2) It offers a powerful equational theory of monadic interpreters. (3) It is realized concretely as a practical Coq library, paying careful attention to proof-engineering details that can be glossed over in pen-and-paper proofs. And finally, (4) it comes with a demonstration that the library is usable in practice, in the form of a novel compiler correctness proof. Our open-source development is publicly available, and all of the results claimed here have been formally proved. Our experience suggests that ITrees are an effective way to work with impure and interactive systems in Coq.

The rest of the paper develops these contributions in detail.

Section 2 introduces interaction trees and establishes that ITrees form a monad with several useful notions of equivalence, including variants of strong and weak bisimulation. It also introduces KTrees (“continuation ITrees”), a point-free representation of functions returning ITrees that is convenient for equational reasoning.

Section 3 explains how to compositionally give semantics to the events of an ITree via monadic event handlers, starting with the familiar example of interpretation into the state monad. It then describes the rich algebraic structure of events and handlers exposed by the library.

Section 4 demonstrates how ITrees support recursion and iteration, allowing us to implement a general fixpoint operator, \texttt{mrec}, whose properties are also described equationally.

Section 5 illustrates the use of ITree-based denotational semantics with an extended case study. We verify the correctness of a compiler from Imp (a simple imperative source language) to Asm (a simple assembly language). In the example we define the semantics of both languages as ITrees by structural recursion on the syntax and the ITree recursion combinators. We then prove the equivalence of the denotations by structural induction on the programs, leveraging the ITree library to completely hide the coinductive nature of the proof. The final result is a termination-sensitive bisimulation.

Section 6 shows (by example) that ITrees are compatible with Coq’s extraction mechanism. Event handlers can easily be written outside of Coq, allowing Coq-generated code to be linked with external libraries for the purposes of debugging, testing, and implementation.

Section 7 compares ITrees to more familiar trace-based semantics by defining the set of event traces of an ITree and showing that two ITrees are weakly bisimilar iff their sets of traces coincide. This correspondence means that ITree-based developments can easily (and formally) be connected with non-executable models based on small-step operational semantics or similar formalisms.

---

1The link to the ITrees GitHub repository is https://github.com/DeepSpec/InteractionTrees. The version of the library as of this publication is on the popl20 branch.
Interaction trees are a datatype for representing computations that can interact with an external environment. We think of such computations as producing a sequence of visible events—interactions—each of which might carry a response from the environment back to the computation. The computation may also eventually halt, yielding a final value, or diverge by continuing to compute internally but never producing a visible event.

Figure 1 shows the definition of the type \( \text{itree } E \rightarrow R \). The parameter \( E \rightarrow \text{Type} \) is a type of external interactions: it defines the interface by which a computation interacts with its environment, as we explain below. \( R \) is the result type of the computation: if the computation ever halts, it will return a value of type \( R \).

ITrees are defined coinductively\(^2\) so that they can represent potentially infinite sequences of interactions or divergent behaviors. They are built using three constructors. \( \text{Ret } r \) corresponds to the trivial computation that immediately halts and produces \( r \) as its result. \( \text{Tau } t \) corresponds to a silent step of computation that does something internal, producing no visible events, and then continues as \( t \). Representing silent steps explicitly allows ITrees to represent diverging computations without violating Coq’s guardedness condition [Chlipala 2017; Giménez 1995].\(^3\)

The final and most interesting way to build an ITree is with the \( \text{Vis } A \rightarrow E \rightarrow R \) constructor (\( A \) is often left implicit). Here, \( e : E \rightarrow A \) is a visible external event, including any outputs provided by the computation to its environment, and \( A \) (for answer) is the type of data that the environment provides in response to the event. The constructor also specifies a continuation, \( k : A \rightarrow \text{itree } E \rightarrow T \), which produces the rest of the computation given the response from the environment. The tree-like nature of interaction trees stems from the \( \text{Vis } \) constructor, since the continuation \( k \) can behave differently for different values of type \( A \). Importantly, the continuation is represented as a meta-level (\( i.e., Gallina \)) function, which means both that we can embed computation in an ITree and that the resulting datatype is extractable and contains executable functions.

As a concrete example of external interactions, suppose we choose \( E \) to be the following type \( \text{IO} \), which represents simple input/output interactions, each carrying a natural number. Then we can define an ITree computation \( \text{echo} \) that loops forever, echoing each input received to the output:

\[\text{coInductive itree } (E: \text{Type } \rightarrow \text{Type}) (R: \text{Type}): \text{Type} := \]

\[| \text{Ret } (r: R) \quad \text{(computation terminating with value } r \text{)} | \]

\[| \text{Tau } (t: \text{itree } E \rightarrow R) \quad \text{("silent" tau transition with child } t \text{)} | \]

\[| \text{Vis } (A: \text{Type}) (e : E \rightarrow A) (k : A \rightarrow \text{itree } E \rightarrow R). \quad \text{(visible event } e \text{ yielding an answer in } A \text{)} | \]

Fig. 1. Simplified presentation of interaction trees.

Section 8 situates ITrees with respect to related work, and Section 9 wraps up with a discussion of limitations and future work.

---

\(^2\)The definition shown here follows Coq’s historical style of using positive coinductive types, which emphasizes the tree-like structure via its constructors. This approach is known to break subject reduction [Giménez 1996] and hence may be deprecated in a future Coq version. Our library therefore uses the recommended negative coinductive form [Hagino 1989; Coq development team 2019] where, rather than defining a coinductive type by providing its constructors, we instead provide its destructors. We use “smart constructors” for \( \text{Ret} \), \( \text{Tau} \), and \( \text{Vis} \), which have the types shown in this figure, so the distinction is mostly cosmetic (though it does impact the structure of proofs). We suppress these and similar details throughout this paper.

\(^3\)The guardedness condition is a syntactic side-condition on \( \text{cofix} \) bodies in Gallina. It ensures that a finite amount of computation suffices to expose the next constructor of the coinductive type. In practice, it means that the results of co-recursive calls must occur under constructors and not be eliminated by pattern matching.

Inductive IO : Type → Type :=
  | Input : IO nat
  | Output : nat → IO unit.

Note that IO is indexed by the expected answer type that will be provided by the environment in each interaction. Conversely, its constructors are parameterized by the arguments to be sent to the environment. Hence, an Input event takes no parameter and expects a nat in return, while an Output event takes a nat but expects a non-informative answer, represented by the unit type. The return type of echo is void, the empty type, since the computation never terminates.

Similarly, it is easy to define an ITree that silently diverges, producing no visible outputs and never returning a value:


Or one that probes the environment until it receives 9 for an answer, at which point it terminates (returning tt, the unique value of type unit):

CoFixpoint kill9 : itree IO unit :=
  Vis Input (fun x ⇒ if x =? 9 then Ret tt else kill9).

The three basic ITree constructors and explicit CoFixpoint definitions provide very expressive low-level abstractions, but working with them directly raises several issues. First, Coq’s syntactic guardedness check is inherently non-compositional, so it is awkward to construct large, complex systems using it. Second, we need ways of composing multiple kinds of events. Third, we often want to model the behavior of a system by interpreting its events as having effects on the environment. For example, a Write event could update a memory cell that a Read event can later access. Finally, to reason about ITrees and computations built from them as above, we would have to use coinduction explicitly. It is easier to work with loop and recursion combinators that are more structured and satisfy convenient equational reasoning principles that can be expressed and proven once and for all. The ITrees library provides higher-level abstractions that address all three of these concerns. Figure 2 provides a synopsis of the library; the details are explained below.

Notation. The library makes extensive use of parametric functions, which have types of the form ∀ (X:Type), E X → F X. We write E as an abbreviation for such types.

2.1 Composing ITree Computations: ITrees are Monads

The type itree E is a monad [Moggi 1989; Wadler 1992] for any E, making it convenient to structure effectful computations using the conventions and notations of pure functional programming. Figure 3 gives the implementation of the monadic bind and ret operations. As shown there, bind t k replaces each Ret r with the new subtree k r. We wrap the Ret constructor as a function ret and introduce the usual sequencing notation x ← e ;; k for bind.

We think of the visible events of an ITree as uninterpreted effects. In this sense, itree E is closely related to the free monad (but technically distinct: see Section 8) where every event of type E A corresponds to an effectful (monadic) operation that can be "triggered" to yield a value of type A:

Definition trigger {E : Type → Type} {A : Type} (e : E A) : itree E A :=
  Vis e (fun x ⇒ Ret x).

Using trigger, we can rewrite the echo example with less syntactic clutter:

CoFixpoint echo2 : itree IO void :=
  x ← (trigger Input) ;; trigger (Output x) ;; Tau echo2.
### Interaction tree operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>itree E A</td>
<td>Type</td>
<td>Interaction tree over types E and A</td>
</tr>
<tr>
<td>Tau : itree E A → itree E A</td>
<td>Type</td>
<td>Tau operation</td>
</tr>
<tr>
<td>Ret : A → itree E A</td>
<td>Type</td>
<td>Reinitialization operation</td>
</tr>
<tr>
<td>Vis : {R} (E R) → (R → itree E A) → itree E A</td>
<td>Type</td>
<td>Visitation operation</td>
</tr>
<tr>
<td>bind : itree E A → (A → itree E B) → itree E B</td>
<td>Type</td>
<td>Binding operation</td>
</tr>
<tr>
<td>trigger : E A → itree E A</td>
<td>Type</td>
<td>Trigger operation</td>
</tr>
</tbody>
</table>

### Events and subevents

- **E, F**: Type
- **E + F**: Disjoint union of events
- **Typeclass E < F**: E is a subevent of F
- **trigger (E < F) : E ~ itree F overloaded trigger**

### Heterogeneous weak bisimulation

\[ \text{eutt} (r : A \rightarrow B \rightarrow \text{Prop}) : \text{itree E A} \rightarrow \text{itree E B} \rightarrow \text{Prop} \]

### Strong and weak bisimulation

\[ \_ \approx \_ : \text{itree E A} \rightarrow \text{itree E A} \rightarrow \text{Prop} \]

\[ \_ \approx := \text{eutt eq.} \]

### Parametric functions

- **E { F := \forall (X:Type), E X \rightarrow F X**

### Monadic interpretation

- **Monad M**
- **MonadIter M**

### Standard event types

<table>
<thead>
<tr>
<th>Name</th>
<th>Events</th>
<th>Handler Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>emptyE</td>
<td>none</td>
<td>( \forall M, \text{emptyE} \sim M )</td>
</tr>
<tr>
<td>stateE S</td>
<td>Get, Put</td>
<td>( \text{(stateE S)} \sim \text{stateT S} )</td>
</tr>
<tr>
<td>mapDefaultE K V d</td>
<td>Insert, LookupDefault, Remove</td>
<td>( \text{(Map K V map)} \sim \text{mapDefaultE K V d} )</td>
</tr>
</tbody>
</table>

*Fig. 2. Main abstractions of the ITrees library (simplified & abridged).*

(* Apply the continuation k to the Ret nodes of the itree t *)

**Definition** bind (E R S) (t : itree E R) (k : R \rightarrow itree E S) : itree E S :=

\[
(\text{cofix bind}_u := \text{match } u \text{ with} \\
| \text{Ret } r \Rightarrow \text{fun } x \Rightarrow \text{bind}_u (k x) \\
| \text{Vis } e k \Rightarrow \text{Vis } e (\text{fun } x \Rightarrow \text{bind}_u (k x)) \\
| \text{end } t).
\]

**Notation** "\( x \leftarrow t1 ;; t2 \) := (bind t1 (fun x \Rightarrow t2))".

**Definition** ret x := Ret x.

*Fig. 3. Monadic bind and ret operators for ITrees.*

### 2.2 ITree Equivalences

Interaction trees admit several useful notions of equivalence even before we ascribe any semantics to the external events. These properties are deceptively simple to state, but the weaknesses of coinduction in Coq make some of them quite difficult to prove.

**Strong and Weak Bisimulations.** The simplest and finest notion of equivalence is strong bisimulation, written \( t1 \equiv t2 \), which relates ITrees \( t1 \) and \( t2 \) when they have exactly the same shape.

The monad laws and many structural congruences hold up to strong bisimulation, but once we introduce loops, recursion, or interpreters, which use Tau to hide internal steps of computation, we need to work with a coarser equivalence. In particular, we want to equate ITrees that agree on their terminal behaviors (they return the same values) and on their interactions with the environment through Vis events, but that might differ in the number of Tau’s. This "equivalence up to Tau" is a form of weak bisimulation: it lets us remove any finite number of Tau’s when considering whether two trees are the same, while infinite Tau’s must be matched on both sides (i.e., this equivalence is termination sensitive). We write \( t \approx u \) when \( t \) and \( u \) are equivalent up to Tau. For instance, we have the defining equation \( \text{Tau } t \approx t \) when \( t \) and \( u \) are equivalent up to Tau. For example, we have the defining equation \( \text{Tau } t \approx t \), which does not hold for strong bisimulation but is crucial for working with general computations modeled as ITrees.

**Heterogeneous Bisimulations.** Both strong and weak bisimulation can be further relaxed to relate ITrees that have different return types, which is needed for building more general simulations,
such as the one used in our compiler correctness proof (Section 5). If we have \( t_1 : \text{itree } E \ A \) and \( t_2 : \text{itree } E \ B \) and some relation \( r : A \rightarrow B \rightarrow \text{Prop} \), we can define \( \text{eutt } r \) ("equivalence up to \Tau\ modulo \( r \)), which is the same as \( \approx \) except that two leaves \( \text{Ret } a \) and \( \text{Ret } b \) are related iff \( r \ a \ b \) holds. Intuitively, two such ITrees produce the same external events and yield results related by \( r \). Indeed \( \approx \) is defined as \( \text{eutt } \text{eq} \), where \( r \) is instantiated to the Leibniz equality relation \( \text{eq} \). It is straightforward to generalize \( \approx \) in the same way.

Figure 4 gives the formal definition of \( \text{eutt } r \) as a nested coinductive–inductive structure. The inner inductive \( \text{euttF} \) relation is parameterized by \( \text{sim} \), a relation on subtrees. It defines a binary relation on nodes of an ITree demanding that \( \text{Ret } a \) relates to \( \text{Ret } b \) only when \( r \ a \ b \) holds. Two \( \text{Vis} \) nodes are related only if they are labeled with identical events and their continuation subtrees are related by \( \text{sim} \) for every value the environment could return, i.e., any value of type \( R \). \( \text{EqTau} \) relates \( \text{Tau } t_1 \) and \( \text{Tau } t_2 \) whenever \( t_1 \) and \( t_2 \) are related by \( \text{sim} \), while \( \text{EqTauL} \) and \( \text{EqTauR} \) allow to strip off asymmetrically one extra \( \text{Tau} \) on either side. Note that \( \text{EqTau} \) and \( \text{EqVis} \) appeal coinductively to the \( \text{sim} \) relation whereas \( \text{EqTauL} \) and \( \text{EqTauR} \) appeal inductively to \( \text{euttF} \). This means that \( \text{eutt} \) can peel off only a finite number of \( \text{Tau} \)’s from one or both trees before having to align them using \( \text{sim} \).

It is easy to show that \( \text{euttF} \) acts monotonically on relations, which allows us to define \( \text{eutt} \) as its greatest fixed point using the \( \text{nu} \) operator. To define \( \text{nu} \) and to work with coinductive predicates like \( \text{eutt} \), we use the \text{paco} library [Hur et al. 2013], which streamlines working with coinductive proofs in Coq.

Although the definition of heterogeneous weak bisimulation is fairly straightforward to state, some of its properties—for instance, transitivity and congruence with respect to \( \text{bind} \)—are quite challenging to prove. For these, we need an appropriate strengthening of the coinductive hypothesis that lets us reason about \( \text{eutt} \) up to closure under transitivity and \( \text{bind} \) contexts. Our Coq library actually uses a yet more general definition that subsumes both strong and weak bisimulation and builds in such "up-to" reasoning to make proofs smoother; we omit these details here and refer the interested reader to the Coq development itself. The upshot is that we can prove the following:

1. \( \approx \) is an equivalence relation.
2. If \( r \) is an equivalence relation, then so is \( \text{eutt } r \).
3. \( \approx \) is an equivalence relation (corollary of (2)).
4. \( t_1 \approx t_2 \) implies \( t_1 \approx t_2 \).

Equational reasoning. Fortunately, clients of the ITrees library can treat the definition of \( \text{eutt } r \) and its instances as black boxes—they never need to look at the coinductive machinery beneath this layer of abstraction. Instead, clients should reason equationally about ITrees. Figure 5 summarizes the most frequently used equations, each of which corresponds to a lemma proved in the library.
The monad laws, structural laws, and congruences let us soundly rearrange an ITree computation—typically to put it into a form where a semantically interesting computation step, such as the interpretation of an event, takes place. Much of the functionality provided by the ITrees library involves lifting this kind of equational reasoning to richer settings, allowing us to work with combinations of different kinds of events and interpretations of their effects.

One pragmatic consideration is that Coq’s `rewrite` and `setoid_rewrite` tactics, which let us rewrite using an equivalence (for instance, replacing the term `C[t1]` with `C[t2]` when we know that `t1 ≈ t2`), only work if the context is proper, meaning that it respects the equivalence. Coq’s `Proper` typeclass registers such contexts with the rewriting tactics. The congruence rules of Figure 5 establish that the ITree constructors themselves are proper functions. We prove instances of `Proper` for all of the operations, such as those in Figure 2, so that we can rewrite liberally. Even so, definitions written in monadic style make heavy use of anonymous functions, which tend to thwart the `setoid_rewrite` tactic’s ability to find the correct `Proper` instances. It is therefore useful to further raise the level of abstraction to simplify rewriting, as we show next.

### 2.3 KTrees: Continuation Trees

To improve equational reasoning principles and leverage known categorical structures for recursion, the ITrees library provides an abstraction for point-free definitions, centered around functions of the form `_ → itree E _`. We can think of these as *impure* Coq functions that may generate events from `E` or possibly diverge. As we will show, they enjoy additional structure that we can exploit to generically derive more ways of composing ITrees computations.

We call types of the form `A → itree E B` *continuation trees*, or KTrees for short:

```
Definition ktree (E : Type → Type) (A B : Type) := A → itree E B.
```

```
Definition eq_ktree {E} {A B : Type} : ktree E A B → ktree E A B → Prop := fun h1 h2 ⇒ ∀ a, h1 a ≈ h2 a.

Infix "≈" := eq_ktree
```

Whereas an `itree E R` directly produces an outcome (Ret, Tau, or Vis), a KTee `k : ktree E A B` first expects some input `a : A` before continuing as an ITree `(k a)`. Equivalence on KTrees, written `≈`, is defined by lifting weak bisimulation pointwise to the function space.

Two KTrees `h : ktree E A B` and `k : ktree E B C` can be composed using `bind`; the result is written `(h >>> k) : ktree E A C`.

```
(* Composition of KTrees *)
Definition cat {E} {A B C : Type} := ktree E A B → ktree E B C → ktree E A C := fun h k ⇒ (fun a ⇒ bind (h a) k).

Infix ">>>" := cat
```
Interaction Trees

id_ : A → itree E A

cat : (B → itree E C) →
      (A → itree E B) → (A → itree E C)

case_ : (A → tree E C) →
       (B → itree E C) →
       (A + B → itree E C)

inl_ : A → itree E (A + B)
inr_ : B → itree E (A + B)
pure : (A → B) → (A → itree E B)

Fig. 6. KTree operations

KTree composition has a (left and right) identity, id_ (equal to ret), and is associative; the proof follows from the monad laws for itree. Together, these facts mean that KTrees are the morphisms of a category, the Kleisli category of the monad itree E.

This category has more structure that we expose as part of the ITrees library interface. The pure operator lifts a Coq function trivially into an event-free KTree computation. We can also easily define an eliminator for the sums type, case_ and corresponding left inl_ and right inr_ injections (effectful variants of the sum type constructors inl and inr). The names of those operations are suffixed with an underscore so as not to conflict with id, inl, and inr from the standard library, as well as for the visual uniformity of case_ with inl_ and inr_. These operations and their types are summarized in Figure 6. They satisfy the equational theory given in Figure 7.

The laws relating case_, inl_, and inr_ mean that KTree is a cocartesian category. The Kleisli and cocartesian categorical structures are represented using typeclasses. These structures allow us to derive, generically, other useful operations and equivalences. For example, the following operations bimap and swap are defined from case_, inl_, and inr_. The KTree bimap f g : ktree E (A + B) (C + D) applies the KTree f : ktree E A B if its input is an A, or g : ktree E C D if its input is a C; the KTree swap : ktree E (A + B) (B + A) exchanges the two components of a sum. As we will see below, event handlers also have a cocartesian structure, which lets us re-use the same generic metatheory for them.

Similarly, the KTree category is just one instance of a Kleisli category, which can be defined for any monad M. Monadic event interpreters, introduced next, build on these structures, letting us (generically) lift the equational theory of KTrees to event interpreters too. This compositionality is important for scaling equational reasoning to situations involving many kinds of events.

3 SEMANTICS OF EVENTS AND MONADIC INTERPRETERS

To add semantics to the events of an ITree, we define an event handler, of type E ➞ M for some monad M. Intuitively, it defines the meaning of an event of E as a monadic operation in M. An interpreter folds such an event handler over an ITree; a good interpretation of ITrees is one that respects itree E’s monadic structure (i.e., it commutes with ret and bind).

Events and handlers enjoy a rich mathematical structure, a situation well known from the literature on algebraic effects (see Section 8). Our library exploits this structure to provide compositional reasoning principles and to lift the base equational theory of ITrees to their effectful interpretations.

3.1 Example: Interpreting State Events

Before delving into the general facilities provided by the ITrees library, it is useful to see how things play out in a familiar instance. The code in Figure 8 demonstrates how to interpret events into a state monad. The event type stateE S defines two events: Get, which yields an answer of state type S, and putE, which takes a new state of type S and yields unit.

In the figure, the state monad transformer operations getT and putT implement the semantics of reading from and writing to the state in terms of the underlying monad M, using its ret. The
function handle_state is a handler for stateE events: it maps events of type stateE S R into monadic computations of type stateT S (itree E) R, i.e., S → itree E (S * R), taking an input state to compute an output state and a result. Given this handler, we define the interp_state function, which folds the handler across all of the visible events of an ITree of type itree (stateE S) R to produce a semantic function of type stateT S (itree E) R. The definition of interp_state is an instance of interp (see Section 3.2 below), specialized to a state monad.

To prove properties about the resulting interpretation, we need to show that interp_state is a monad morphism, meaning that it respects the ret and bind operations of the ITree monad.

\[
\text{interp\_state \ (ret \ x) \ s \ \approx \ ret \ (s, x)}
\]
\[
\text{interp\_state \ (x \ \leftarrow \ t; ; \ k \ \leftarrow \ s) \ \approx \ (\text{interp\_state \ t} \ \text{s} \ ; ; \ \text{interp\_state \ (k \ \leftarrow \ x)} \ \text{s} 2)
\]

We next prove that handle_state implements the desired behaviors for the get and put operations, which are short-hands for the trigger of the corresponding stateE events.

\[
\text{interp\_state \ get \ s \ \approx \ ret \ (s, s)}
\]
\[
\text{interp\_state \ (put \ s') \ s \ \approx \ ret \ (s', \ tt)}
\]

These equations allow us to use put and get’s semantics when reasoning about stateful computations. They are also sufficient to derive useful equations when verifying programs optimizations—for instance, we can remove a redundant get as follows:

\[
\text{interp\_state \ (x \ \leftarrow \ \text{get} ; ; \ y \ \leftarrow \ \text{get} ; ; \ k \ \leftarrow \ x \ y) \ \approx \ \text{interp\_state \ (x \ \leftarrow \ \text{get} ; ; \ k \ \leftarrow \ x)} \ \text{s}
\]

3.2 Monadic Interpreters

The interp_state function above is an instance of a general interp function that is defined for any monad M, provided that M supports an iteration operator, iter, of type (A → M (A + B)) → A → M B. (The first argument is a loop body that takes an A and produces either another A to keep looping with or a final result of type B.) Figure 9 shows the definition of interp. It takes a handler : E ↠ M and loops over a tree of type itree E R. At every iteration, the next constructor of the tree is interpreted by handler, yielding the remaining tree as a new loop state.

The core properties of interp, summarized in Figure 11, are generalizations of the laws for interp_state. In particular, interp preserves the monadic structure of ITrees, and its action on trigger e is to apply the handler to the event e.
Definition interp \{E \bowtie M : Type \rightarrow Type\} \{R : Type\} (handler : E \leadsto M) : itree E R \rightarrow M R := iter (fun t : itree E R \Rightarrow 
match t with
| Ret r ⇒ ret (inr r)
| Tau t ⇒ ret (inl t)
| Vis e k ⇒ bind (handler _ e) (fun a ⇒ ret (inl (k a)))
end).

Fig. 9. Interpreting events via a handler

definition id_ : E \leadsto itree E (* trigger *)
definition cat : (F \leadsto itree G) \rightarrow (* interp *) (E \leadsto itree F) \rightarrow (E \leadsto itree G)
definition case_ : (E \leadsto itree G) \rightarrow (F \leadsto itree G) \rightarrow (E + F \leadsto itree G)
definition inl_ : E \leadsto itree (E + F)
definition inr_ : F \leadsto itree (E + F)

Fig. 10. Event Handler operations

Recall that the general type of an event handler is E \leadsto M. When M has the form itree F, we can think of such a handler as translating the E events into F events. We call handlers of this type ITree event handlers. Like KTrees, event handlers form a cocartesian category where composition of handlers uses interp, and the identity handler is trigger. The interface is summarized in Figure 10.

Definition cat {E F G} : (E \leadsto itree F) \rightarrow (F \leadsto itree G) \rightarrow (E \leadsto itree G)

Definition id_ {E} : E \leadsto itree E := @trigger E.

4The 1 in inl1 and inr1 reminds us that E and F live in Type \rightarrow Type.
iter \( (A \rightarrow \text{itree } E (A + B)) \rightarrow (A \rightarrow \text{itree } E B) \)

loop \( (C + A \rightarrow \text{itree } E (C + B)) \rightarrow (A \rightarrow \text{itree } E B) \)

mrec \( (E \rightsquigarrow \text{itree } (E + ' F)) \rightarrow (E \rightsquigarrow \text{itree } F) \)

Fig. 12. Summary of recursion combinators

```
CoFixpoint iter (body : A \rightarrow \text{itree } E (A + B)) : A \rightarrow \text{itree } E B :=
fun a \Rightarrow
  ab
  match ab with
  | inl a \Rightarrow Tau (iter body a)
  | inr b \Rightarrow Ret b
  end
end.
```

Definition loop (body : C + A \rightarrow \text{itree } E (C + B)) : A \rightarrow \text{itree } E B :=
fun a \Rightarrow iter (fun ca \Rightarrow
  cb
  match cb with
  | inl c \Rightarrow Ret (inl (inl c))
  | inr b \Rightarrow Ret (inr b)
  end)
(inr a).

Fig. 13. Iteration combinators: iter and loop.

The equivalence relation for handlers \( h \approx k \) is defined as \( \forall \ A\ (e : E A), \ (h A e) \approx (g A e) \), i.e., pointwise weak bisimulation. It admits the same equational theory (and derived constructs) as for KTrees, hence we reuse the same notations for the operations (see Figure 7).

Subevents. When working with ITrees at scale, it is often necessary to connect ITrees with fewer effects to ITrees with more effects. For instance, suppose we have an ITree \( t : \text{itree } IO A \) and we want to bind it with a continuation \( k \) of type \( A \rightarrow \text{itree } (X + ' IO + ' Y) B \) for some event types \( X \) and \( Y \). A priori, this isn’t possible, since the types of their events don’t match. However, since there is a natural structural inclusion \( \text{inc} : IO \rightsquigarrow X + ' IO + ' Y \) (given by \( \text{inl}_- \circ \text{inr}_- \)) we can first interpret \( t \) using the handler \( \text{fun } e \Rightarrow \text{trigger} \ (\text{inc } e) \) and then bind the result with \( k \).

Since the need for such structural inclusions arises fairly often, the ITrees library defines a typeclass, written \( E \prec F \), that can automatically synthesize inclusions such as \( \text{inc} \). It generically derives an instance of \( \text{trigger} : E \rightsquigarrow \text{itree } F \) whenever there is a structural subevent inclusion \( E \rightsquigarrow F \). We will see in the case study how this flexibility is useful in practice.

4 ITERATION AND RECURSION

While Coq does provide support for coinduction and corecursion, its technique for establishing soundness relies on syntactic mechanisms that are not compositional. To make working with ITrees more tractable to clients, our library provides first-class abstractions to express corecursion as well as reasoning principles for these abstractions that hide the brittle nature of Coq’s coinduction. From the point of view of a library user, recursive definitions using these combinators need only to typecheck, even when they lead to divergent behaviors.

Our library exports two iteration constructs, \( \text{iter} \) and \( \text{loop} \), and a recursion combinator \( \text{mrec} \) (Figure 12). They are mutually inter-derivable, but they permit rather distinct styles of recursive definitions.

4.1 Iteration

The first function is a combinator for \( \text{iteration} \), \( \text{iter} \), whose implementation is shown in Figure 13. Given \( \text{body} : A \rightarrow \text{itree } E (A + B) \) and a starting state \( a : A \), \( \text{iter} \ \text{body} \ a \) is a computation that produces either a new state from which to iterate the \( \text{body} \) again (after a \( \text{Tau} \)), or a final value to stop the computation. This operator makes no assumption on the shape of the loop body, a marked improvement over the intensional guardedness check required by \( \text{cofix} \).

Defining fixpoint combinators as functions allows us to prove their general properties \( \text{once and for all} \). The equations for \( \text{iter} \), given below, imply that continuation trees form an iterative
category [Bloom and Ésik 1993]. The fixed point identity unfolds one iteration of the iter loop; the parameter identity equates a loop followed by a computation with a loop where that computation is part of its last iteration; the composition identity equates a loop whose body sequences two computations \( f, g \) with a loop sequencing them in reverse order, prefixed by a single iteration of \( f \); and the codiagonal identity merges two nested loops into one.

\[
\begin{align*}
\text{iter } f & \cong f \gg\gg \text{case}_\text{ (iter } f \text{)} \text{id}_\text{} \quad \text{(fixed point)} \\
\text{iter } f \gg\gg g & \cong \text{iter } (f \gg\gg \text{bimap id}_\text{ g}) \quad \text{(parameter)} \\
\text{iter } (f \gg\gg \text{case}_\text{ g } \text{inr}_\text{}) & \cong f \gg\gg \text{case}_\text{ (iter } (g \gg\gg \text{case}_\text{ f } \text{inr}_\text{)) } \text{id}_\text{} \quad \text{(composition)} \\
\text{iter } (\text{iter } f) & \cong \text{iter } (f \gg\gg \text{case}_\text{ inl}_\text{ id}_\text{}) \quad \text{(codiagonal)}
\end{align*}
\]

The proofs of these equations makes nontrivial use of coinductive reasoning for weak bisimulation; carrying them out in a proof assistant is a significant contribution of this work. Nevertheless, that complexity is entirely hidden from users of the library, behind the simple interface exposed by these equations, whose expressiveness we’ll demonstrate in our case study in Section 5.

The iter implementation shown in Figure 13 is specialized to the itree \( E \) monad. However, we can generalize to other monads and characterize the abstraction using the following typeclass:

\[
\text{Class MonadIter } (M : \text{Type} \rightarrow \text{Type}) \{\text{Monad } M\} := \\
\text{iter } : \forall A B, (A \rightarrow M (A + B)) \rightarrow A \rightarrow M B.
\]

Good implementations of the MonadIter interface must satisfy the iterative laws. In a total language such as Coq, this limits the possible implementations. Base instances include ITrees and the predicate monad \( \_ \rightarrow \text{Prop} \) where we can tie the knot using the impredicative nature of Prop. In addition, we can lift MonadIter through a wide variety of monad transformers, e.g., stateT \( S M \) where \( M \) is an instance of MonadIter.

**Traced categories.** Figure 13 also shows the loop combinator, an alternative presentation of recursion that is derivable from iter. We can think of the \( C \) part of the body’s input and output types as input and output “ports” that get patched together with a “back-edge” by iter. We use loop in Section 5 to model linking of control-flow graphs. This loop combinator equips KTrees with the well-studied structure of a traced monoidal category [Hasegawa 1997; Joyal et al. 1996].

### 4.2 Recursion

Figure 14 shows the code for a general mutual-recursion combinator, mrec. The combinator uses a technique developed by McBride [2015] to represent recursive calls as events. Here, an indexed type \( D : \text{Type} \rightarrow \text{Type} \) gives the signature of a recursive function, or, using multiple constructors, a block of mutually recursive functions. For example, \( D := \text{ackermannE} \) represents a function with two nat arguments and a result of type nat.

\[
\text{Inductive ackermannE } : \text{Type} \rightarrow \text{Type} := \\
| \text{Ackermann } : \text{nat} \rightarrow \text{nat} \rightarrow \text{ackermannE } \text{nat}.
\]

A recursive event handler for \( D \) is an event handler of type \( D \sim \text{itree } (D + \prime E) \), so it can make recursive calls to itself via \( D \) events, and perform other effects via \( E \) events. As an example, the handler \( h_{\text{ackermann}} \) pattern-matches on the event Ackermann \( m n \) to extract the two arguments of the function, and to refine the result type to nat. The body of the function makes recursive calls by trigger-\( \text{ing} \) Ackermann events, without any requirement to ensure the well-foundedness of the definition.

\[
\text{Definition } h_{\text{ackermann}} : \text{ackermannE } \rightarrow \text{itree } (\text{ackermannE } + \prime \text{emptyE}) := \\
\text{fun } (\text{Ackermann } m n) \Rightarrow \text{if } m =? \text{0 then Ret } (n + 1) \\
\qquad \text{else if } n =? \text{0 then trigger } (\text{inl}_1 (\text{Ackermann } (m-1) 1)) \\
\qquad \text{else } (\text{ack} \leftarrow \text{trigger } (\text{inl}_1 (\text{Ackermann } m (n-1)))) ;
\]

(* Interpret an itree in the context of a mutually recursive definition (rh) *)

**Definition**

```
mrec {D E} (rh : D \sim itree (D +' E)) : D \sim itree E :=
  fun R d ⇒
    iter (fun t : itree (D +' E) R ⇒
      match t with
      | Ret r ⇒ Ret (inr r)
      | Tau t ⇒ Ret (inl t)
      | Vis (inl1 d) k ⇒ Ret (inl (bind (rh _ d) k))
      | Vis (inr1 e) k ⇒ bind (trigger e) (fun x ⇒ Ret (inl (k x)))
    end) (rh _ d).
```

Fig. 14. Mutual recursion via events

```
trigger (inl1 (Ackermann (m-1) ack))
```

The `mrec` combinator ties the knot. Given a recursive handler `D \sim itree (D +' E)`, it produces a handler `D \sim itree E`, where all `D` events have been handled recursively.

**Definition**

```
ackermann : nat → nat → itree emptyE nat :=
  fun m n ⇒ mrec h_ackermann (Ackermann m n).
```

The implementation of `mrec` in Figure 14 works similarly to `interp`, applying the recursive handler `rh` to events in `D`. However, whereas `interp` directly uses the ITree produced by the handler as output, `mrec` adds it as a prefix of the ITree to be interpreted recursively: the `inl1 d` branch returns `bind (rh _ d) k`, which will be processed in subsequent steps of the `iter` loop.

For reasoning, `mrec` is also characterized as a fixed point by an *unfolding equation*, which applies the recursive handler `rh : D \sim itree (D +' E)` once, and interprets the resulting ITree with `interp`, where `D` events are passed to `mrec` again, and `E` events are passed to the identity handler, *i.e.*, `trigger`, which keeps events uninterpreted.

```
mrec rh d ≈ interp (case_ (mrec rh) id_) (rh d)
```

In fact, `mrec` is an analogue of `iter`, equipping event handlers themselves with the structure of an iterative category. It satisfies the same equations as `iter`, relating event handlers instead of KTrees.

## 5 Case Study: Verified Compilation of IMP to ASM

To demonstrate the compositionality of ITree-based semantics and the usability of our Coq library, we use ITrees to formalize and verify a compiler from a variant of the IMP language from *Software Foundations* [Pierce et al. 2018] to a simple assembly language, called Asm.

We begin by explaining the denotational semantics of IMP (Section 5.1) and Asm (Section 5.2). It is convenient to define the semantics in stages, each of which justifies a different notion of program equivalence. The first stage maps syntax into ITrees, thereby providing meaning to the control-flow constructs of the language, but not ascribing any particular meaning to the events corresponding to interactions with the memory. The second stage interprets those events as effects that manipulate (a representation of) the actual program state.

We then give a *purely inductive* proof of the correctness of the compiler (Sections 5.3 and 5.4). The denotational model enables us to state a termination-sensitive bisimulation and prove it purely equationally in a manner not much different from traditional compilers for terminating languages with simpler denotational semantics [Pierce et al. 2018]. The correctness proof relates the semantics of IMP to the semantics of Asm after their events have been appropriately interpreted into state monads (a necessity, since compilation introduces new events that correspond to reading and writing intermediate values). Since IMP programs manipulate one kind of state (global variables)
and Asm programs manipulate two kinds of state (registers and the heap), the proof involves building an appropriate simulation relation between Imp states and Asm states.

To streamline, we identify Imp global variables with Asm heap addresses and assume that Imp and Asm programs manipulate the same kinds of dynamic values. Neither assumption is critical.

5.1 A Denotational Semantics for Imp

The syntax and the semantics for Imp is given in Figure 15. In the absence of while, a denotational semantics could be defined, by structural recursion on statements, as a function from an initial environment to a final environment; the denotation function would have type \( \text{Imp} \rightarrow \text{env} \rightarrow (\text{env} \times \text{Unit}) \). However, it is not possible to give a semantics to while using this naive denotation because Gallina’s function space is total. The usual solution is to revamp the semantics dramatically, e.g., by moving to a relational, small-step operational semantics (Section 8.5 discusses other approaches). With ITrees, the denotation type becomes \( \text{ImpState} \rightarrow \text{stateT env (itree F) unit} \), or, equivalently, \( \text{Imp} \rightarrow \text{env} \rightarrow \text{itree F (env \times \text{Unit})} \), which allows for nontermination. It is also more flexible, since the semantics can be defined generically with respect to an event type parameter \( F \), which can later be refined if new effects are added to the language or if we want to compose ITrees generated as denotations of Imp programs with ITrees obtained in some other way.

Figure 15 shows how the Imp semantics are structured. We first define \( \text{denote_expr} \) and \( \text{denote_imp} \), which result in trees of type \( \text{itree E unit} \). The typeclass constraint \( \text{ImpState} \prec E \) indicates that \( E \) permits \( \text{ImpState} \) actions, a refinement of \( \text{stateE} \) that provides events for reading and writing individual variables; we would follow the same strategy to add other events such as IO. The meanings of expressions and most statements are straightforward, except for While. This relies on the \( \text{iter} \) combinator (see Section 4) to first run the guard expression, then either continue to loop (by returning \( \text{inl tt} \) to the \( \text{iter} \) combinator) or signal that it is time to stop (by returning \( \text{inr tt} \)).

The second stage of the semantics is \( \text{interp_imp} \), which takes ITrees containing \( \text{ImpState} \) events and produces a computation in the state monad. It first invokes a handler for \( \text{ImpState, h_imp_state} \).
which often comes with a number of administrative reduction rules. Instead, we represent these
we need not necessarily consider the impact of

the target of our compiler is


syntax for the language, which is split into two levels: basic blocks and control-flow subgraphs.

Fig. 16. Syntax and semantics of Asm

to translate the Imp-specific GetVar and SetVar events into the general-purpose mapE events provided
by the ITrees library (the bimap operator propagates other events untouched). It then uses interp_map
to define their meaning in terms of actual lookup and set operations on the type env, a simple finite
map from var to value. The final semantics of an Imp statement s is obtained simply by composing
the two functions: interp_imp (denote_imp s).

Factoring the semantics this way is useful for proofs. For instance, to prove the soundness of a
syntactic program transformation from s to s’ it suffices to show that denote_imp s ≈ denote_imp s’;
we need not necessarily consider the impact of interp_imp. We will exploit this semantic factoring
in the compiler proof below by reasoning about syntactic “linking” of Asm code before its state-
transformer semantics is considered.

This style of denotational semantics avoids defining a syntactic representation of machines,
which often comes with a number of administrative reduction rules. Instead, we represent these
administrative reductions using Gallina functions and Tau transitions in the semantics (though
these are hidden by iter). As we will see in Section 5.4, this uniform representation will allow us
to reason up to Tau and completely ignore these steps in our proofs.

5.2 A Denotational Semantics for Asm

The target of our compiler is Asm, a simple assembly language that represents computations as
collections of basic blocks linked by conditional or unconditional jumps. Figure 16 gives the core
syntax for the language, which is split into two levels: basic blocks and control-flow subgraphs.
A basic block (block) is a sequence of straight-line instructions followed by a branch that transfers control to another block indicated by a label. As with IMP expressions, the denotation of instructions is mostly uninteresting, so we omit them.

A control-flow subgraph, or “sub-CFG,” (asm in Figure 16) represents the control flow of a computation. These are open program fragments, represented as sets of labeled basic blocks. The labels in a sub-CFG are separated into three groups: entry labels, from which the code in a sub-CFG can start executing, exit labels, where the control flow leaves the sub-CFG, and internal labels, which are invisible outside of the subgraph. A sub-CFG has a block of code for every entry and internal label; control leaves the subgraph by jumping to an exit label. Labels are drawn from finite domains, e.g., fin A and fin B where A and B are nats, as seen in bks.5

Figure 16 presents the denotation of Asm programs, which factors into two parts, just as we saw for IMP. The GetReg and SetReg events represent accesses of register state, and Load and Store accesses of memory.6 Once again, we give meaning to the control-flow constructs of the syntax independently of the state events. The result of denote_bks is a ktree that maps each entry label of type fin A to an itree that returns the label of the next block to jump to. The den_asm function first computes the denotation of each basic block and then wires the blocks together using loop, hiding the internal labels in the process.

The stateful semantics of Asm programs is given by interp_asm, which, like interp_imp, realizes the register and memory as finite maps using interp_map. As a result of this nesting, the “intermediate state” of an Asm computation is a value of type memory * (register * A). Because they are built compositionally from interpreters, it is very easy to prove that both interp_imp and interp_asm are monad morphisms in the sense that they commute (up to Tau) with ret and bind, a fact that enables proofs by rewriting.

5.3 Linking of Control-Flow Subgraphs

We now turn to the compilation of IMP to Asm. The compiler and its proof are each split into two components. The first phase handles reasoning about control flow by embedding sub-CFGs into KTrees. In the second phase, we perform the actual compilation and establish its functional correctness by reasoning about the quotienting of the local events.

For the first phase, we first implement a collection of reusable combinators for linking sub-CFGs. These combinators correspond to the operations on KTrees described in Section 2.3, which can be seen in this context as presenting a theory of graph linking at the denotational level. Here are the signatures of the four essential ones (their implementations are straightforward):

\[
\begin{align*}
\text{Definition} & \quad \text{app_asm} \quad (ab : \text{asm} A B) \quad (cd : \text{asm} C D) : \text{asm} (A + C) (B + D). \\
\text{Definition} & \quad \text{loop_asm} \quad (ab_ : \text{asm} (I + A) (I + B)) : \text{asm} A B. \\
\text{Definition} & \quad \text{pure_asm} \quad (f : A \rightarrow B) : \text{asm} A B. \\
\text{Definition} & \quad \text{relabel_asm} \quad (f : A \rightarrow B) \quad (g : C \rightarrow D) \quad (bc : \text{asm} B C) : \text{asm} A D.
\end{align*}
\]

Two sub-CFGs can be placed beside one another while preserving their labels, via app_asm. Linking of compilation units is performed by loop_asm: it connects a subset of the exit labels I as back edges to the imported labels, also named I, and internalizes them. Visible labels can be renamed with relabel_asm. Finally, pure_asm creates, for every label a : A, a block that jumps immediately to f a. Together, relabel_asm and pure_asm provide the plumbing required to use the combinators app_asm and loop_asm effectively.

5. The finiteness of labels is useful for Asm program transformations and is faithful to “real” assembly code, but this restriction is not actually necessary. The correctness proof is independent of this choice, so we sweep the details under the rug. 6. An additional Done event (not shown) represents halting the whole program for blocks terminated by a Bhalt instruction via exit, but we omit it for the purposes of this exposition.

The correspondence between these core Asm combinators and operations on KTrees is given by the following equations, which commute the denotation function inside the combinator.

\[
\begin{align*}
\text{den}_\text{asm} (\text{app}_\text{asm} \ ab \ cd) & \approx \text{bimap} \ (\text{den}_\text{asm} \ ab) \ (\text{den}_\text{asm} \ cd) \\
\text{den}_\text{asm} (\text{loop}_\text{asm} \ ab) & \approx \text{loop} \ (\text{den}_\text{asm} \ ab) \\
\text{den}_\text{asm} (\text{relabel}_\text{asm} \ f \ g \ bc) & \approx (\text{pure} \ f \ggg \text{den}_\text{asm} \ bc \ggg \text{pure} \ g) \\
\text{den}_\text{asm} (\text{pure}_\text{asm} \ f) & \approx \text{pure}_\text{ktree} \ f
\end{align*}
\]

Equipped with these primitives, building more complex control-flow graphs becomes a diagrammatic game. Figure 17 shows how to use the primitives to build linking operations for sub-CFGs that mimic the control-flow operations provided by Imp. For instance, sequential composition of \(\text{asm} \ A \ B\) with \(\text{asm} \ B \ C\) places them in parallel, swaps their entry labels to get a sub-CFG of type \(\text{asm} \ (B+A) \ (B+C)\), and then internalizes the intermediate label \(B\) via \(\text{loop}_\text{asm}\).

We emphasize that while these control-flow graphs are specific to Imp, their definitions do not depend on Imp’s or even Asm’s state-transformer semantics. We can reason about control-flow independently of other events. For instance, the denotation of the \(\text{seq}_\text{asm}\) combinator is indeed the sequential composition of denotations of its arguments (up to \(\tau\)):

**Lemma** seq_asm_correct \(\{A \ B \ C\} \ (ab : \text{asm} \ A \ B) \ (bc : \text{asm} \ B \ C) : \ (\text{den}_\text{asm} \ (\text{seq}_\text{asm} \ ab \ cd)) \approx (\text{den}_\text{asm} \ ab \ggg \text{den}_\text{asm} \ bc)\).

The \(\text{while}_\text{asm}\) combinator is, naturally, more involved. As illustrated in Figure 17, it constructs the control-flow graph of a while loop given the list of instructions for the test condition and the compilation unit corresponding to the body of the loop. The type of \(p\) represents a compilation unit with a single imported label (the target to jump to when the loop body finishes) and a single exported label (the entry label for the top of the loop body). The correctness of the combinator establishes that its denotation can be viewed as an entry point that runs the body if a variable \(\text{tmp}_\text{if}\) is non-zero after evaluating the expression \(e\). This is expressed at the level of KTrees via the \(\text{loop}\) operator. In the code below, \(\text{label}_\text{case}\ 1\) analyzes the shape of the label 1, and 11 and 12 are two distinct label constants corresponding to the loop entry or exit, respectively.

**Lemma** while_asm_correct \(\{e : \text{list instr}\} \ (p : \text{asm} \ 1 \ 1) : \ (\text{den}_\text{asm} \ (\text{while}_\text{asm} \ e \ p)) \approx \text{loop} \ (\text{fun} \ l: \text{fin} \ (1 + 1) \Rightarrow \\
\text{match} \ \text{label}_\text{case} \ 1 \ \text{with} \\
| \ \text{inl} \ _ & \Rightarrow \text{denote}_\text{list} \ e \ ;; \ v \leftarrow \text{trigger} \ (\text{GetReg} \ \text{tmp}_\text{if}) \ ;; \\
| \ \text{if} \ (v:\text{value}) \ \text{then} \ \text{Ret} \ 12 \ \text{else} \ (\text{denote}_\text{asm} \ p \ 11 ;; \ \text{Ret} \ 11) \\
| \ \text{inr} \ _ & \Rightarrow \text{Ret} \ 11)\).
Most importantly, the proof of $\text{while asm correct}$ is again purely equational, relying solely on the theory of KTrees and correctness equations of the low-level linking combinators ($\text{app asm correct}$, $\text{seq asm correct}$, etc.).

### 5.4 Compiler Correctness

The compiler itself is entirely straightforward. It compiles IMP statements using the linking combinators along with $\text{compile assign}$ and $\text{compile expr}$, both of which are simple (and omitted). We pass to $\text{compile expr}$ the name of a target register (here, just $\theta$), into which the value of the expression will be computed; it stores intermediate results in additional ASM registers as needed.

```lean
Fixpoint compile (s : stmt) {struct s} : asm 1 1 :=
| Skip ⇒ id_asm
| Assign x e ⇒ raw_asm_block (after (compile_assign x e) (Bjmp l1))
| Seq l r ⇒ seq_asm (compile l) (compile r)
| If e l r ⇒ if_asm (compile_expr 0 e) (compile l) (compile r)
| While e b ⇒ while_asm (compile_expr 0 e) (compile b)
end.
```

The top-level compiler correctness theorem is phrased as a bisimulation between the IMP program and the corresponding ASM program, which simply requires them to have "equivalent" behavior.

**Theorem** $\text{compile correct}$ ($s$ : stmt) : equivalent $s$ (compile $s$).

Figure 18 unpacks the definition of $\text{equivalent}$, which requires the ITree denotations of $s$ and its compilation to be bisimilar. Two ITrees $t1$ and $t2$, representing IMP and ASM computations respectively, of types $\text{itree (ImpState + E) A}$ and $\text{itree (Reg + Memory + E) B}$, are bisimilar if, when run in $\text{Renv}$-related initial states, they produce computations that are equivalent up to $\text{Tau}$ and both terminate in states related by $\text{state invariant} \ TT$. The relation $\text{Renv}$ formalizes the assumption that the IMP environment and ASM memory have the same contents when viewed as maps from IMP variables / ASM addresses to values, and it is implied by $\text{state invariant}$. Here, $\text{TT}$ is the trivial relation on the output label of ASM, since a statement has a unique exit point; in general the $\text{RAB}$ relation parameter in $\text{state invariant}$ is used to ensure that both computations jump to the same label, which is needed to prove that loops preserve the state invariant.

Since the compiler introduces temporary variables, the bisimulation does not hold over the uninterpreted ITrees. To prove that expressions are compiled correctly, we need to explain how reads and writes of ASM registers relate to the computations done at the IMP level. The relation $\text{sim rel}$ establishes the needed invariants, which ensure that the code generated by the compiler (1) doesn’t corrupt the ASM memory, (2) uses registers in a “stack discipline,” and (3) computes the IMP intermediate result $v$ into the target register $n$. These properties are used to prove the correctness of $\text{compile expr}$.

Crucially, despite correctness being termination sensitive, the proofs follow by structural induction on the IMP terms: all coinductive reasoning is hidden in the library. As in the first phase, the reasoning here follows by rewriting, this time using the bisimilarity relation and equations about $\text{interp imp}$ and $\text{interp asm}$ that are induced by virtue of being compositionally defined from $\text{interp state}$.

Setting aside the usual design of the simulation relation, the resulting proofs are slightly verbose, but extremely elementary. They mostly consist in successive rewrites to commute the denotation with the various combinators, and some elementary semantic reasoning where events are reached
\[ \text{Relate an Imp env to an Asm memory} \]

**Definition**
\[
\text{Renv } (g_{\text{imp}} : \text{Imp.env}) (g_{\text{asm}} : \text{Asm.memory}) : \text{Prop} := \\
\forall k v, \text{alist\_In } k g_{\text{imp}} v \leftrightarrow \text{alist\_In } k g_{\text{asm}} v.
\]

**Definition**
\[
\text{sim\_rel } l_{\text{asm}} n : (\text{Imp.env} \times \text{value}) \rightarrow (\text{Asm.memory} \times (\text{Asm.registers} \times \text{unit})) \rightarrow \text{Prop} := \\
\text{fun } '(g_{\text{imp}}', v) '(g_{\text{asm}}', (l_{\text{asm}}', _)) \Rightarrow \\
\text{Renv } g_{\text{imp}}' g_{\text{asm}}' \wedge \\
(\forall m, m < n \rightarrow \forall v, \text{alist\_In } m l_{\text{asm}} v \leftrightarrow \text{alist\_In } m l_{\text{asm}}' v).
\]

(* we don't corrupt any of the Imp state *)

(* we get the right value in register n *)

(* we don't mess with anything on the "stack" *)

**Context**
\( (A : \text{Type}) \). (* Imp / Asm intermediate result types *)

**Context**
\( (RAB : A \rightarrow B \rightarrow \text{Prop}) \). (* Parameter that relates intermediate results *)

\[ \text{(Relate Imp to Asm intermediate states. *)} \]

**Definition**
\[
\text{state\_invariant } (a : \text{Imp.env} \times A) (b : \text{Asm.memory} \times (\text{Asm.registers} \times B)) := \\
\text{Renv } (\text{fst } a) (\text{fst } b) \wedge (RAB (\text{snd } a) (\text{snd } (\text{snd } b))).
\]

**Definition**
\[
\text{bisimilar } E (t1 : \text{itree} (\text{ImpState} + E) A) (t2 : \text{itree} (\text{Reg} + E) B) := \\
\forall g_{\text{asm}} g_{\text{imp}}, \text{Renv } g_{\text{imp}} g_{\text{asm}} \rightarrow \\
\text{eutt } (\text{state\_invariant } RAB) (\text{interp\_imp } t1 g_{\text{imp}}) (\text{interp\_asm } t2 g_{\text{asm}}).
\]

(* Imp / Asm program equivalence *)

**Definition**
\[
\text{TT} : \text{unit} \rightarrow \text{fin } 1 \rightarrow \text{Prop} := \text{fun } _ _ \Rightarrow \text{True}.
\]

**Definition**
\[
\text{equivalent } (s : \text{stmt}) (t : \text{asm } 1 1) : \text{Prop} := \text{bisimilar } \text{TT} (\text{denote\_stmt } s) (\text{den\_asm } t f1).
\]

Fig. 18. The simulation relations for the compiler correctness proof.

to prove that the simulation relation is preserved. We believe that these kind of equational proofs can be automated to a large degree, a perspective we would like to explore in further works.

6 EXTRACTING ITREES

One of the big benefits of ITrees is that they work well with Coq’s extraction facilities. If we extract the echo definition from Section 2, we obtain the code shown at the top of Figure 19.\(^7\) The itree type extracts as a lazy datatype and observe forces its evaluation.

To actually run the represented computation, we provide a driver that traverses the itree, forcing all of its computation and providing handlers for any visible events that remain in the tree. The OCaml function run does exactly that, where, for the sake of this example, we interpret each Input event as a call to OCaml’s \text{read\_int} command and each Output event as a call to \text{print\_int}.\(^8\) This kind of simple event handling already suffices to add basic IO and “printf debugging” to Coq programs, which can be extremely handy in practice. We can, of course, implement more sophisticated event handlers, using the full power of OCaml.

ITrees extractability has played a key role in several different parts of an ongoing research project that seeks to use Coq for Deep Specifications.\(^9\) In particular, our re-implementation of the Vellvm\(^10\) formalization of LLVM, which aims to give a formal semantics for the LLVM IR in Coq, heavily uses ITrees exactly as proposed in this paper to build a denotational semantics for LLVM IR code. The Vellvm semantics has many layers of events and handlers (for global data, local data, interactions with the memory model, internal and external functions calls, etc.), and the LLVM

\(^7\) For simplicity, here we also extract Coq’s \text{nat} type as OCaml’s \text{int} type.

\(^8\) Thanks to its dependent type, the OCaml extraction of \text{Vis} uses OCaml’s \text{Obj.t} as the domain of the embedded continuations, so handlers should be written with care, otherwise type-safety could be jeopardized.


\(^10\) https://github.com/vellvm/vellvm
let rec echo = lazy (Vis (Input, (fun x -> lazy (Vis ((Output (Obj.magic x)), (fun _ -> echo)))))

(* OCaml handler -----(not extracted) ------------------------------------------ *)
let handle_io e k = match e with
| Input -> k (Obj.magic (read_int ()))
| Output x -> print_int x ; k (Obj.magic ())

let rec run t = match observe t with
| Ret r -> r
| Tau t -> run t
| Vis (e, k) -> handle_io e (fun x -> run (k x))

Fig. 19. OCaml extracted from the echo example (top); OCaml handler and “driver” loop (bottom).

IR control-flow graphs are a richer version of the Asm language (Section 5.2). The flexibility of using ITree-based interpreters means that Vellvm can define a relational specification that accounts for nondeterministic features of the LLVM (such as undef) but that can also be refined into an implementation. We are able to extract an executable interpreter that performs well enough to test small- to medium-sized LLVM code samples (including recursion, loops, etc.). All but the outermost run driver are extracted from Coq, as in the echo example.

We are also using ITrees as executable specifications to model the semantics of web servers [Koh et al. 2019]. The ITree representation serves two purposes: (1) ITrees model the interactive operations of the web server in a way that can be connected via Princeton’s VST framework [Appel 2011, 2014] to a C implementation, and (2) the model can also be used for property-based testing with QuickChick [Lampropoulos and Pierce 2018]. The ability to link against handlers written in OCaml means that the testing framework can be used to test real web servers like Apache across the network, in addition to linking against our own web servers. Here again, the performance of the extracted executable has been good enough that we have felt no need to do any optimization on the ITree representation.

7 RELATING ITREES AND TRACE SEMANTICS

We have shown that ITrees provide a way to define denotational semantics for possibly diverging, effectful programs in Coq. In this approach, we use monadic interpreters that produce ITrees as a semantic representation of program behaviors, which we can then reason about equationally.

A more common approach to defining language semantics in Coq (and in other proof assistants) is via a deep embedding, in which the program’s operational semantics are specified relationally. For example, the CompCert project [Leroy 2009] takes this approach, where the fundamental transition relation is given by step : state → event → state → Prop. Here, a state is a representation of the current program state, and the event type contains information about both the outputs to the environment and the inputs that the program might receive from the environment. The proposition step s1 e s2 holds when it is possible for the system to transition from state s1 to state s2 while producing the observable event e. The meaning of a complete program is given by the set of all finite sequences of events, called traces, generated by the transitive closure of the step relation.

An important distinction in trace semantics is that the step relation quantifies over possible inputs that it might receive from the environment. Unlike ITrees, whose Vis e k constructors expose the continuation k as a function, relational semantics with input events are fundamentally not
Inductive trace (E : Type → Type) (R : Type) : Type :=
| TEnd : trace E R
| TRet : R → trace E R
| TEventEnd : ∀(X), E X → trace E R
| TEventResponse : ∀(X), E X → X → trace E R → trace E R.

Inductive is_trace_of (E : Type → Type) (R : Type) :
| itree E R → trace E R → Prop :=
| TraceEmpty : ∀t, is_trace_of t TEnd
| TraceRet : ∀r, is_trace_of (Ret r) (TRet r)
| TraceTau : ∀t tr, is_trace_of t tr → is_trace_of (Tau t) tr
| TraceVisEnd : ∀(e : E X) k, is_trace_of (Vis e k) (TEventEnd e)
| TraceVisContinue : ∀(e : E X) (x : X) (k tr),
  is_trace_of (k x) tr → is_trace_of (Vis e k) (TEventResponse e x tr).

Fig. 20. ITree traces: is_trace_of t tr means that tr is a possible trace of ITree t.

executable, because of that universal quantification. If we try to shoehorn ITrees into a small-step transition relation in this style, the Vis case becomes:

Inductive step {E R} : (itree E R) → {X & E X * X} → (itree E R) → Prop :=
| step_vis : ∀e (x:X) k, step (Vis e k) (existT _ X (e, x)) (k x).

We instantiate the event type as an existential package containing a pair of the event e : E X and a response x : X, which is universally quantified—the step can take place for any x of type X provided by the environment. This propositional encoding of inputs means that operational semantics developed in this style cannot be extracted from Coq. Consequently, such semantic definitions cannot easily be used as implementations or executable tests. The CompCert project goes to some pains to implement a separate interpreter that corresponds to their small-step semantics to aid with debugging.

On the other hand, traces are sometimes convenient, particularly when there is inherent non-determinism in the specification of a system’s behaviors. Rather than defining a trace of an ITree using a small-step semantics, like CompCert, it is more natural to think of an ITree as directly denoting a set of possible traces—finite prefixes of paths through the tree that record Vis events and corresponding responses from the environment. This definition is shown in Figure 20.

The trace datatype is intuitively a list of events. TEnd marks a partial trace; it corresponds to spin or ⊥ (we cannot distinguish the two, a problem of using finite traces). TRet r denotes a computation that finished, producing the value r. TEventResponse e x t corresponds to a Vis event e to which the environment responded with the answer x, then continues with trace t, and TEventEnd e corresponds to a situation in which the ITree is waiting for a response from the environment (perhaps one that will never come, i.e., if it has an event of type E void).

The is_trace_of predicate leads to a natural notion of trace refinement, and thus a different characterization of ITree equivalence.

Definition 1 (Trace Refinement). t ⊑ u iff ∀tr, is_trace_of t tr → is_trace_of u tr.

Definition 2 (Trace Equivalence). t ≡ u iff t ⊑ u and u ⊑ t.

Using these definitions, we can show that trace equivalence coincides with weak bisimulation, i.e., that t₁ ≈ t₂ ⇐⇒ t₁ ≡ t₂.

Trace refinement shows that spin ⊑ t. Since all events in the ITree are visible in the trace, if u ⊑ t, then t can be obtained from u by replacing silently diverging behaviors with some visible events, or otherwise inserting/removing a finite number of Taus into/from u. In many cases, different notions of refinement may be more useful. For instance, one may want nondeterministic behavior to be
refined by possible deterministic behaviors. One way to do this is to directly allow the \texttt{is_trace_of}

predicate to allow a choice of possible refinements, effectively interpreting away some events in the refinement definition. A second option is to introduce a more sophisticated version of refinement that uses both the definition above and an event interpreter as described in Section 3.2—the resulting relations should be quite similar to those studied by Johann et al. [2010].

8 RELATED WORK

The problem of accommodating effectful programming in purely functional settings is an old one, and a variety of approaches have been explored, monads and algebraic effects being two of the most prominent. We concentrate on these two techniques, beginning with general background and then focusing on the closest related work.

8.1 Monads, Monad Transformers, and Free Monads

Moggi’s seminal paper [1989] introduced monads as one way to give meaning to imperative features in purely functional programs. Monads were subsequently popularized by Wadler [1992] and Peyton Jones and Wadler [1993] and have had huge impact, especially in Haskell. However, it was soon recognized that composing monads to combine multiple effects was not straightforward. Monad transformers [Moggi 1990] are one way to obtain more compositionality; for example, Liang et al. [1995] showed how they can be used to build interpreters in a modular way. The \texttt{interp_state}

function from Section 3 is an example of building an event interpreter using a monad transformer in this style. In our case, not all monads are suitable targets for interpretation: we require them to support recursion in the sense that their Kleisli category is iterative. Correspondingly, not all monad transformers can therefore be used to build interpreters.

Sweirstra’s Datatypes à la Carte [2008] showed how to use a \texttt{free monad} to define monad instances modularly. Transporting his definition to our setting, we would obtain the following:

\begin{verbatim}
CoInductive Free (E : Type → Type) (R:Type) :=
| Ret : R → Free E R
| Vis : E (Free E R) → Free E R.
\end{verbatim}

This version of the \texttt{Vis} constructor directly applies the functor \texttt{E} to the coinductively defined type \texttt{Free E R} itself. However, this type violates the strict positivity condition enforced by Coq: certain choices of \texttt{E} would allow one to construct an infinite loop.

Subsequent work by Apfelmus [2010], Kiselyov et al. [2013] and Kiselyov and Ishii [2015] showed how free monads can be made more liberal by exposing the continuation in the \texttt{Vis} constructor. The resulting “freer” monad (called \texttt{FFree} in their work) is essentially identical to our \texttt{ITrees}—the difference being that, because they work in Haskell, which admits nontermination by default, it needs no \texttt{Tau} constructor.

When considered up to strong bisimulation, \texttt{ITrees} form the free \textit{completely iterative monad} [Aczel et al. 2003] with respect to a functor of the form \texttt{fun X ⇒ F X + X}, where the second component corresponds to \texttt{Tau} nodes. Quotiented by weak bisimulation, \texttt{ITrees} define a free \textit{pointed monad} [Uustalu and Veltri 2017]. There is a rich literature on the theory of iteration [Bloom and Ésik 1993; Goncharov et al. 2017; Milius 2005], studying the properties of operators such as \texttt{mrec} in yet more general category-theoretic settings. The \texttt{ITrees} library makes such results concretely applicable to formally verified systems.

\texttt{ITrees} are a form of resumptions, which originated from concurrency theory [Milner 1975]. More precisely, \texttt{ITrees} can be obtained by applying a coinductive resumption monad transformer [Cenciarelli and Moggi 1993; Piróg and Gibbons 2014] to the delay monad of Capretta [2005]. Other variations of the resumption monad transformer have been used to model effectful and concurrent
 programs [Goncharov and Schröder 2011; Nakata and Uustalu 2010]. In particular, Nakata and Uustalu [2010] also used coinductive resumptions in Coq to define the semantics of Imp augmented with input-output operations. They also defined termination-sensitive weak bisimilarity ("equivalence up to taus") using mixed induction-coinduction. However, their semantics was defined as an explicitly coinductive relation, with judicious introductions of Tau. Their Coq development was specialized to Imp’s global state and was not intended to be used as a general-purpose library. In contrast, our semantics are functional (denotational, definitional) interpreters, and we encapsulate nontermination (Tau is an internal implementation detail) using recursion operators that are compatible with Coq’s extraction mechanisms.

8.2 Algebraic Effects and Handlers

Algebraic effects are a formalism for expressing the semantics of effectful computations based on the insight by Plotkin and Power that many computational effects are naturally described by algebraic theories [2001; 2002; 2003]. The idea is to define the semantics of effects equationally, with respect to the term model generated by operations \( op \in \Sigma \), the signature of an algebra. When combined with the notion of an effect handler, an idea originally introduced by Cartwright and Felleisen [1994] and later investigated by Plotkin and Pretnar [2013], algebraic effects generalize to more complex control effects yet still justify equational reasoning. The monoidal structure of algebraic effects is well known [Hyland et al. 2006]; more recent work has studied the relationship between monad transformers and modular algebraic effects [Schrijvers et al. 2016].

In our setting, an event interface such as \( \text{stateE} \) (Figure 8) defines an effect signature \( \Sigma \), and its constructors Get and Put \( s \) define the operations. Plotkin and Pretnar used the notation \( \text{op}(x:X. M) \), corresponding to the ITrees \( \text{Vis} \, \text{op} \, (\text{fun} \, x:X \Rightarrow M) \) construct, and called it "operation application". They axiomatized the intended semantics of effects via equations on operation applications—for example, the fact that two get operations can get collapsed into one was expressed by the equation
\[
\text{get}(x:S. \text{get}(y:S. kxy)) = \text{get}(x:S. kxx).
\]
For ITrees, we prove such equations relative to an interpretation of the events, as in Section 3.1.

The handlers of algebraic effects specify the data needed to construct an interpretation of the effect; they have the form \( \text{handler}\{\text{return} \ x \mapsto f(x), (\text{op}(y; \kappa) \mapsto h(y, \kappa))_{\text{op}\in\Sigma}\} \). In terms of our notation, the return component of the handler specifies the Ret case of an interpreter, and the sum over operation interpretations is written using a dependent type. Here, \( h \) corresponds to the most general elimination form for the ITree \( \text{Vis} \) constructor, which is a function of type \( \forall X, E \, X \rightarrow (X \rightarrow \text{itree E} \, R) \rightarrow M \, R \) for \( M \) an iterative monad. However, Coq prevents us from creating a general-purpose interpreter parameterized by such a type—it needs to see the definition of the handler’s body to verify the syntactic guardedness conditions.

In a language such as Eff [Bauer and Pretnar 2015], which supports algebraic effects natively, the operational semantics plumbs together the continuations with the appropriate handlers, scoping them according to the dynamic semantics of the language. In our case, we must explicitly invoke functions like \( \text{interp_state} \) as needed, possibly after massaging the structure of events so that they have the right form.

Johann et al. [2010] studied the contextual equivalences induced by interpretations of standard effects. Most saliently, their paper developed its theory in terms of observations of “computation trees,” which are “incompletely known” ITrees—they are inductively defined, and hence finite, but may also include \( \bot \) leaves that denote (potential) divergence. Johann et al. showed how to endow the set of computation trees with a CPO structure based on approximation (\( \bot \subseteq t \) for any tree \( t \)) and use that notion to study contextual equivalences induced by various interpreters. The techniques proposed there should be adaptable to our setting: instead of working with observational partial orders, we might choose to work more directly with the ITree structures themselves.
8.3 Effects in Type Theory

Most of the work discussed above was done either in the context of programming languages with support for general recursion or in a theoretical "pen and paper" setting, rendering these approaches fundamentally different to the ITree library which is formalized in a total language. Work more closely related to ITrees is that undertaken in the context of dependent type theory.

The earliest work on mixing effects with type theory was done by Hancock and Setzer [2000], followed by Hancock’s dissertation [Hancock 2000]. This line of work, inspired by monads and especially Haskell’s IO monad, showed how to encode such constructs in Martin-Löf type theory. Those theories, in contrast to ITrees, do not allow silent steps of computation, instead integrating guarded or sized coinductive types as part of a strong discipline of total functional programming. The benefit of this is that strong bisimilarity is the only meaningful notion of equivalence; the drawback is that they cannot handle general recursion. Later work on object encodings [Setzer 2006] did consider recursive computations, though it did not study their equational theory or the general case of implementing interpreters within the type theory, as we have done. More recently, Abel et al. [2017] have demonstrated the applicability of these ideas in Agda. Although their paper includes a proof of the correctness of a stack object (among other examples), they do not focus on the general equational theory of such computations.

As mentioned previously, Capretta proposed using the “delay monad” to encode general recursion in a type theory, as we do here, though his paper used strong bisimulation as the notion of equivalence. The delay monad can be seen as either an ITree without the Vis constructor or, isomorphically, an ITree of type itree emptyE R. The main theoretical contribution of that paper was showing that the monad laws hold and that the resulting system is expressive enough to be Turing complete. Subsequent work explored the use of the delay monad for defining operational semantics [Danielsson 2012] and studied how to use quotient types [Chapman et al. 2015] or higher inductive types [Altenkirch et al. 2017] to define equivalence up to Tau, which we take as the basis for most of our equational theory. Because we are working in Coq, which does not have quotient types or higher inductive types, we must explicitly use setoid rewriting, requiring us to prove that all morphisms respect the appropriate equivalences.

McBride [2015], building on Hancock’s earlier work, used what he called the “general monad” to implement effects in Agda. His monad variant is defined inductively as shown below.

\[
\text{Inductive General (S:Set) (T : S \rightarrow Set) (X : Set) : Set :=} \\
\mid \text{RetG (x : X)} \\
\mid \text{VisG (s:S) (k : T s \rightarrow General S T X).}
\]

Its interface replaces our single E : Type \rightarrow Type parameter with S : Type and a type family S \rightarrow Type to calculate the result type of the event. McBride proposed encoding recursion as an (uninterpreted) effect, as we present in Section 4. In particular, he shows how to give a semantics to recursion using first a “fuel”-based (a.k.a. step-indexed) model and then by translation into Capretta’s delay monad. The latter can be seen as a version of our interp_mrec, but one in which all of the effects must be handled. Our coinductively defined interaction trees also support a general fixpoint combinator directly, which is impossible for the General monad.

The FreeSpec Coq library, implemented by Letan et al. [2018], uses a “program monad” to model components of complex computing systems. The program monad is essentially an inductive version of itree\(^{11}\) (without Tau). What we call “events,” the FreeSpec project calls “interfaces.” The FreeSpec project is primarily concerned with modeling first-order, low-level devices for which general recursion is probably not needed. Its library offers various composition operators, including

\(^{11}\) The original version of FreeSpec also included a bind constructor, but, following our ITrees development, it was removed in favor of defining bind.
a form of concurrent composition, and it includes a specification logic that helps prove (and automate proofs of) properties about the systems being modeled. However, due to FreeSpec’s use of the inductive definition, such systems must be structured as acyclic graphs. Nevertheless, FreeSpec doesn’t eschew coinduction altogether—as we explain below, it, like CompCert, defines the environment in which the program runs coinductively. FreeSpec’s handlers are thus capable of expressing diverging computations, but it does not support the equational reasoning principles that we propose.

8.4 Composition with the Environment

An idea that is found in several of the works discussed above is the need to characterize properties of the program’s environment. Recall the kill9 program from earlier, which halts when the input is 9 but continues otherwise. One might wish to prove that, if the environment never supplies the input 9, the program goes on forever. In a more realistic setting like CompCert, one might wish to make assertions about externally supplied functions, such as OS calls, malloc or memcpy, or to reason about the accumulated output on some channel such as the terminal.

The behavior of the environment is, in a sense, dual to the behavior of the program. CompCert, for example, formulates the environment as a coinductively defined “world,” whose definition is (a richer version of) the following:

\[
\text{CoInductive } \text{world} : \text{Type} := \text{World} (\text{io} : \text{string} \rightarrow \text{list eventval} \rightarrow \text{option (eventval} \times \text{world})
\]

Here the string and list of eventvals are the outputs of the event (they are provided by the program), and the result (if any) is a returned value and a new world. The environment’s state is captured in the closure of the io function. Transliterating this type to our setting we arrive at:

\[
\text{CoInductive } \text{world } E : \text{Type} := \text{World} (\text{io} : \forall A : \text{Type}, E A \rightarrow \text{option (A} \times \text{world } E)).
\]

Letan et al. [2018] use a definition very close to this (without the option) to define a notion of “semantics” for the program monad. Given such a definition, one can define a world that satisfies a certain property (for example, one that never produces 9 as an answer) and use it to constrain the inputs given to the program, by “running” the program under consideration in the given world. CompCert defines “running” via a predicate called possible_trace that matches the answers provided by the io function to the events of the program trace.

The CertiKOS project [Gu et al. 2015, 2018] takes the idea of composing a program with its environment even further. Their Concurrent Certified Abstraction Layers (CCAL) framework also uses a trace-based formulation of semantics. In their context, traces are called logs and (concurrent) components are given semantics in terms of sets of traces. Each component (e.g., a thread) can be separately given a specification in terms of its interface to (valid) external environments, which encode information about the scheduler and assumptions about other components in the context. A layer interface can “focus” on subsets of its concurrently executing components; when it is focused on a single, sequential thread, the interface is a deterministic function from environment interactions (as represented by the log) to its next action. The parallel layer composition operation links two compatible layers by “running” them together (as above) according to the schedule (inputs to one component can be provided by outputs of the other). In this case, one thread’s behaviors influence another thread’s environment. They formulate such interactions in terms of concepts from game semantics, which gives rise to a notion of refinements between layer specifications. Layers have the symmetric monoidal structure familiar from algebraic effects.

We conjecture that the sequential behavior of the CCAL system could be expressed in terms of ITrees and that the concurrent composition operations of the framework could be defined on top of that. Our KTree combinators already offer a rich notion of composition, including general,
mutually recursive linking, which is similar to that offered by CCAL. Moreover, we can define similar “running” operations directly on ITrees, rather than on traces, coordinating multiple ITrees via an executable scheduler. This means that, besides proving properties of the resulting system, we can extract executable test cases [Koh et al. 2019].

8.5 Formal Semantics
There are a plethora of techniques used to describe the semantics of programming languages within proof assistants. In evaluating these techniques, we need to consider both the simplicity of the definitions and their robustness to language extensions. The former is important because complex models are difficult to reason about, while the latter is important because seemingly small changes sometimes cascade through a language, invalidating previous work.

Denotational semantics translate the object language (e.g., Imp or Asm) into the meta-language (e.g., Gallina), seeking to leverage the existing power of the proof assistant. Chlipala [2007] also uses denotational semantics to verify a compiler, but in a simpler setting with a normalizing source language, and with models individually tailored to the intermediate languages. In contrast, ITrees serve as a common foundation for both semantics in our case-study compiler, and an equational theory enabling the verification of a termination-sensitive theorem. As we saw in Section 5.1, impure features such as nontermination can make this difficult, as proof assistants often include only a total function space. One way to circumvent this limitation is via a “fuel”-based semantics, where computations are approximated to some finite amount of unwinding. Owens et al. [2016] use this approach to develop functional big-step semantics. To reason about nonterminating executions, Owens et al. [2016] leverages the classical nature of the HOL logic to assert that, if no amount of fuel is sufficient for termination, then the computation diverges. They further show how oracle semantics [Hobor 2008] can be used to enrich this language with both IO and nondeterminism. In practice, the approach is quite similar to ITrees, except that we can omit the fuel and instead directly construct the infinite computation tree. With ITrees, events encode oracle queries and Taus represent internal steps, which may lead to divergence. Though, as we showed in Section 5, users of ITrees are mostly insulated from Taus when using the combinators from Section 4.

The approach of Owens et al. [2016] is reminiscent of traditional step indexing [Ahmed 2004], in which the meaning of a program is described by a set of increasingly accurate approximations. Coinductive interaction trees enable us to describe an entire, possibly infinite, computation once and for all. Post-facto, ITrees can be easily approximated by a collection of trees or traces (Section 7), providing a means to recover step-indexed reasoning if desired.

Formalizations of more classic domain-theoretic denotational models exist [Benton et al. 2010, 2009]. Unfortunately, the learning curve for this style of denotational semantics was widely considered to be quite steep. The complexity of domain theoretic models prompted exploring more operational approaches to formalizing semantics [Plotkin 2004a,b]. Big-step operational semantics share a similar flavor to denotational semantics as they both connect terms directly to their meaning. Unfortunately, interpreting big-step semantics inductively prevents them from representing divergent computations. Some works [Chlipala 2010; Delaware et al. 2013] avoid the issue of nontermination entirely, ascribing semantics only to terminating executions. Charguéraud [2013] provides a technique for avoiding the problem by duplicating the semantics both inductively and coinductively. They argue that such duplication can be automated and therefore should not be overly burdensome. The functional style of this “pretty big step” semantics is quite similar to functional denotational semantics, and thus bears a resemblance to ITrees. ITrees avoid the need to duplicate the semantics by giving a data representation rather than a propositional representation.

Leroy and Grall [2009] give an in-depth discussion relating inductive and coinductive semantic styles, providing an inductive judgment for “terminates in a value (and a trace)” and a coinductive
judgment for “diverges (with an infinite trace).” Relating these semantics can be difficult, and the proofs sometimes rely on classical logic.

9 CONCLUSION

Interaction trees are a promising basis on which to build denotational semantics for impure and recursive computations in theorem provers like Coq. We have established a solid theoretical foundation and demonstrated a usable realization of the theory in Coq, as well as its practical use via a demonstrative verified compiler example.

A natural question concerns the ease of transferring this work to other proof assistants. While the theory does not depend on Coq, adapting the choices made through the conception of the library to other languages for formal verification may require some non-trivial work.

We leverage mainly three features of Coq: extraction, coinductive types, and higher-order types. While extraction (or, alternatively, executability) is available in most popular modern proof systems, the two other characteristics can raise challenges in adapting this work. More specifically, Lean [de Moura et al. 2015] lacks coinductive types, making it seemingly inadequate to the task. Isabelle/HOL [Nipkow et al. 2002] lacks higher-order types, which appears to be a serious obstacle to a faithful translation of our work. An expert might find a way to encode the ITree generic event types, but we are unsure about the feasibility of the task: this might be a stumbling block. Finally, Agda [Norell 2007] would be perfectly suitable, and some related work of a similar structure, discussed in Section 8, actually enjoy a formalization in this proof assistant.

Many directions for further exploration remain. Other kind of effects, such as nondeterminism and concurrency, are instrumental in the modeling of some systems: developing simulations and reasoning principles for those represent a valuable challenge. We are accumulating empirical evidence that ITrees are both expressive enough to be adequate in various targets for formalization, while also being very convenient to work with. Their compositional and modular nature seems to lead to better proofs than traditional approaches, notably when it comes to reasoning about control flow. Building formal bridges and comparisons to related approaches such as domain theoretic denotations, operational semantics, step-indexed-based approaches, or game semantics models would be a great opportunity to attempt to ground this empirical evidence. Finally, the versatility of ITrees make them a potential fit in many contexts. Stress-testing their viability and scalability is a major avenue we are beginning to explore.

Acknowledgements

This work was funded by the National Science Foundation’s Expedition in Computing The Science of Deep Specification under the awards 1521602 (Appel), 1521539 (Weirich, Zdancewic, Pierce) with additional support by the NSF projects Verified High Performance Data Structure Implementations, Random Testing for Language Design, award 1421243 (Pierce), ONR grant REVOLVER award N00014-17-1-2930, and by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (2017R1A2B2007512). We are especially thankful to Joachim Breitner and Dmitri Garbuzov for early contributions to this work. We are grateful to all the members of the DeepSpec project for their collaboration and feedback, and we greatly appreciate the the reviewers’ comments and suggestions.

REFERENCES


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](https://doi.org/10.1007/978-3-642-78034-9)


