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:
Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (January 2020), 32 pages. https://doi.org/10.1145/3371119

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.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/1-ART51
https://doi.org/10.1145/3371119

RIGHTSLINK

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* [Piròg 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

RIGHTSLINK

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,[1] 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, `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.

[1] The 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.

RIGHTSLINK

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

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 INTERACTION TREES

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 `itree E R`. The parameter `E : Type → 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. `Ret r` corresponds to the trivial computation that immediately halts and produces `r` as its result. `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 `Vis A e k` constructor (`A` is often left implicit). Here, `e : E 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 → itree E T`, which produces the rest of the computation given the response from the environment. The tree-like nature of interaction trees stems from the `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 `IO`, which represents simple input/output interactions, each carrying a natural number. Then we can define an ITree computation `echo` that loops forever, echoing each input received to the output:

[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 `Ret`, `Tau`, and `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 `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.

RIGHTSLINK

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

CoFixpoint echo : itree IO void :=
  Vis Input (fun x ⇒ Vis (Output x) (fun _ ⇒ echo)).
```

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:

```
CoFixpoint spin : itree IO void := Tau spin.
```

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 ⇝ F` 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.
```


RIGHTSLINK

Interaction tree operations

```
itree E A : Type
Tau  : itree E A → itree E A
Ret  : A → itree E A
Vis  : {R} (E R) → (R → itree E A) → itree E A
bind : itree E A → (A → itree E B) → itree E B
trigger : E A → itree E A
```

Events and subevents

```
E, F : Type → Type
(e : E R)            R  is the result type of event  e
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

```
eutt (r : A → B → Prop) :
itree E A → itree E B → Prop
```

Strong and weak bisimulation

```
_ ≅ _ : itree E A → itree E A → Prop
_ ≈ _ := eutt eq.
```

Parametric functions

```
E ⇝ F  :=  ∀ (X:Type), E X → F X
```

Monadic interpretation

```
`{Monad M} `{MonadIter M}
interp : (E ⇝ M) → (itree E ⇝ M)
```

Standard event types

name	*events*	*handler type*
`emptyE`	*none*	`∀ M, emptyE ⇝ M`
`stateE S`	`Get  Put`	`(stateE S) ⇝ stateT S`
`mapDefaultE K V d`	`Insert LookupDefault Remove`	`` `{Map K V map} (mapDefaultE K V d) ⇝ (stateT map) ``

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 → itree E S) : itree E S :=
  (cofix bind_ u := match u with
                    | Ret r ⇒ k r
                    | Tau t ⇒ Tau (bind_ t)
                    | Vis e k ⇒ Vis e (fun x ⇒ bind_ (k x))
                    end) t.

Notation "x ← t1 ;; t2" := (bind t1 (fun x ⇒ 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 ≅ 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 ≈ u` when `t` and `u` are equivalent up to `Tau`. For instance, we have the defining equation `Tau t ≈ 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,

RIGHTSLINK

```
Context {E : Type → Type} {A B : Type} (r : A → B → Prop).

Inductive euttF (sim : itree E A → itree E B → Prop) : itree E A → itree E B → Prop :=
| EqRet a b (REL: r a b) : euttF sim (Ret a) (Ret b)
| EqVis {R} (e : E R) k1 k2 (REL: ∀v, sim (k1 v) (k2 v)) : euttF sim (Vis e k1) (Vis e k2)
| EqTau t1 t2 (REL: sim t1 t2) : euttF sim (Tau t1) (Tau t2)
| EqTauL t1 ot2 (REL: euttF sim t1 ot2) : euttF sim (Tau t1) ot2
| EqTauR ot1 t2 (REL: euttF sim ot1 t2) : euttF sim ot1 (Tau t2).

Lemma euttF_monotone t1 t2 sim sim' (IN: euttF sim t1 t2) (LE: sim <2= sim') : euttF sim' t1 t2.

Definition eutt : itree E A → itree E B → Prop := nu euttF.
```

Fig. 4. Heterogeneous weak bisimulation for ITrees.

such as the one used in our compiler correctness proof (Section 5). If we have `t1 : itree E A` and `t2 : itree E B` and some relation `r : A → B → Prop`, we can define `eutt r` ("equivalence up to `Tau` modulo `r`"), which is the same as ≈ except that two leaves `Ret a` and `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 ≈ is defined as `eutt eq`, where `r` is instantiated to the Leibniz equality relation `eq`. It is straightforward to generalize ≅ in the same way.

Figure 4 gives the formal definition of `eutt r` as a nested coinductive–inductive structure. The inner inductive `euttF` relation is parameterized by `sim`, a relation on subtrees. It defines a binary relation on nodes of an ITree demanding that `Ret a` relates to `Ret b` only when `r a b` holds. Two `Vis` nodes are related only if they are labeled with identical events and their continuation subtrees are related by `sim` for every value the environment could return, *i.e.*, any value of type `R`. `EqTau` relates `Tau t1` and `Tau t2` whenever `t1` and `t2` are related by `sim`, while `EqTauL` and `EqTauR` allow to strip off asymmetrically one extra `Tau` on either side. Note that `EqTau` and `EqVis` appeal *coinductively* to the `sim` relation whereas `EqTauL` and `EqTauR` appeal *inductively* to `euttF`. This means that `eutt` can peel off only a finite number of `Tau`'s from one or both trees before having to align them using `sim`.

It is easy to show that `euttF` acts monotonically on relations, which allows us to define `eutt` as its greatest fixed point using the `nu` operator. To define `nu` and to work with coinductive predicates like `eutt`, we use the 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 `bind`—are quite challenging to prove. For these, we need an appropriate strengthening of the coinductive hypothesis that lets us reason about `eutt` up to closure under transitivity and `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) ≅ *is an equivalence relation.*
(2) If `r` *is an equivalence relation, then so is* `eutt r`.
(3) ≈ *is an equivalence relation (corollary of (2)).*
(4) `t1` ≅ `t2` *implies* `t1` ≈ `t2`.

Equational reasoning. Fortunately, clients of the ITrees library can treat the definition of `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.

RIGHTSLINK

Monad Laws

```
(x ← ret v ;; k x) ≅ (k v)
(x ← t ;; ret x) ≅ t
(x ← (y ← s ;; t) ;; u) ≅ (y ← s ;; x ← t ;; u)
```

Structural Laws

```
(Tau t) ≈ t
(x ← (Tau t) ;; k) ≈ Tau (x ← t ;; k)
(x ← (Vis e k1) ;; k2) ≈
        (Vis e (fun y ⇒ (k1 y) ;; k2))
```

Congruences

```
t1 ≅ t2 →      Tau t1 ≅ Tau t2
k1 ≈̂ k2 →   Vis e k1 ≈ Vis e k2
t1 ≈ t2 ∧ k1 ≈̂ k2 → bind t1 k1 ≈ bind t2 k2
```

Fig. 5. Core equational theory of ITrees.

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) : 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 KTree `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
```


RIGHTSLINK

```
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

```
id_ >>> k ≈ k
k >>> id_ ≈ k
(i >>> j) >>> k ≈ i >>> (j >>> k)
pure f >>> pure g ≈ pure (f ∘ g)
inl_ >>> case_ h k ≈ h
inr_ >>> case_ h k ≈ k
(inl_ >>> f) ≈ h ∧ (inr_ >>> f) ≈ k →
                   f ≈ case_ h k
```

Fig. 7. Categorical Laws for KTrees and Handlers (cat is denoted by ">>>")

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

RIGHTSLINK

```
(* The type of
     state events *)
Variant stateE (S : Type)
  : Type → Type :=
| Get : stateE S S
| Put : S → stateE S unit.
```

```
(* State monad transformer *)
Definition stateT (S:Type) (M:Type → Type) (R:Type) : Type :=
  S → M (S * R).
Definition getT (S:Type) : stateT S M S := fun s ⇒ ret (s, s).
Definition putT (S:Type) : S → stateT S M unit :=
  fun s' s ⇒ ret (s', tt).
```

```
(* Handler for state events *)
Definition h_state (S:Type) {E}
  : (stateE S) ⇝ stateT S (itree E) :=
  fun _ e ⇒ match e with
            | Get   ⇒ getT S
            | Put s ⇒ putT S s
            end.
```

```
(* Interpreter for state events *)
Definition interp_state {E S}
  : itree (stateE S) ⇝ stateT S (itree E) :=
  interp h_state.
```

Fig. 8. Interpreting state events

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.

```
interp_state (ret x) s ≈ ret (s, x)
interp_state (x ← t;; k x) s1 ≈ '(s2, x) ← interp_state t s1;; interp_state (k x) s2
```

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 correponding `stateE` events.

```
interp_state get s ≈ ret (s,s)
interp_state (put s') s ≈ 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:

```
interp_state (x ← get ;; y ← get ;; k x y) s ≈ interp_state (x ← get ;; k x x) 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`.

RIGHTSLINK

```
Definition interp {E M : Type → Type} `{MonadIter M} {R : Type} (handler : E ⇝ M)
  : itree E R → M R := iter (fun t : itree E R ⇒
                               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

```
id_   : E ⇝ itree E          (* trigger *)
cat   : (F ⇝ itree G) →      (* interp *)
        (E ⇝ itree F) → (E ⇝ itree G)
case_ : (E ⇝ itree G) →
        (F ⇝ itree G) → (E +' F ⇝ itree G)
inl_  : E ⇝ itree (E +' F)
inr_  : F ⇝ itree (E +' F)
```

Fig. 10. Event Handler operations

```
interp h (trigger e)   ≅ h _ e
interp h (Ret r)       ≅ ret r
interp h (x ← t;; k x) ≅
  x ← (interp h t);; interp h (k x)
```

Fig. 11. Some properties of `interp`. See also Figure 7 for equations in terms of `cat`.

It remains to show how to instantiate the `MonadIter` typeclass, which provides the `iter` combinator used by `interp`. We defer this discussion to Section 4, as it will benefit from a closer look at events and handlers.

3.3 The Algebra of Events and ITree Event Handlers

The `handle_state` handler interprets computations with events drawn from the specific type `stateE S`. More generally, we often want to combine multiple kinds of events in one computation. For instance, we might want both `stateE S` and `IO` events, or access to two different types of state at the same time. Fortunately, it is straightforward to define `E +' F`, the disjoint union of the events `E` and `F`. The definition comes with inclusion operations `inl1 : E ⇝ E +' F` and `inr1 : F ⇝ E +' F`.[4] The `emptyE` event type, with no events, is the unit of `+'`.

The corresponding operations on handlers manipulate sums of event types: `case_` combines handlers for different event types into a handler on their sum, while `inl_` and `inr_` are `inl1` and `inr1` turned into event handlers.

```
Definition case_ {E F M} : (E ⇝ M) → (F ⇝ M) → (E +' F) ⇝ M
  := fun f g _ e ⇒ match e with
                   | inl1 e1 ⇒ f _ e1
                   | inr1 e2 ⇒ g _ e2
                   end.

Definition inl_ {E F} : E ⇝ itree (E +' F)
Definition inr_ {E F} : F ⇝ itree (E +' F)
```

Recall that the general type of an event handler is `E ⇝ 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 ⇝ itree F) → (F ⇝ itree G) → (E ⇝ itree G)
  := fun f g _ e ⇒ interp g (f _ e).

Definition id_ {E} : E ⇝ itree E := @trigger E.
```

[4] The 1 in `inl1` and `inr1` reminds us that `E` and `F` live in `Type → Type`.

RIGHTSLINK

```
iter : (A → itree E (A + B)) → (A → itree E B)
loop : (C + A → itree E (C + B)) → A → itree E B
mrec : (E ⇝ itree (E +' F)) → (E ⇝ itree F)
```

Fig. 12. Summary of recursion combinators

```
CoFixpoint iter (body : A → itree E (A + B))
  : A → itree E B :=
  fun a ⇒ ab ← body a ;;
          match ab with
          | inl a ⇒ Tau (iter body a)
          | inr b ⇒ Ret b
          end.
```

```
Definition loop (body : C + A → itree E (C + B))
    : A → itree E B :=
  fun a ⇒ iter (fun ca ⇒
            cb ← body ca ;;
            match cb with
            | inl c ⇒ Ret (inl (inl c))
            | inr b ⇒ Ret (inr b)
            end) (inr a).
```

Fig. 13. Iteration combinators: `iter` and `loop`.

The equivalence relation for handlers `h ≈̂ k` is defined as `∀ A (e: E A), (h A e) ≈ (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 : itree IO A` and we want to `bind` it with a continuation `k` of type `A → 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 `inc: IO ⇝ X +' IO +' Y` (given by `inl_ ∘ inr_`) we can first interpret `t` using the handler `fun e ⇒ trigger (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 -< F`, that can automatically synthesize inclusions such as `inc`. It generically derives an instance of `trigger : E ⇝ itree F` whenever there is a structural subevent inclusion `E ⇝ 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, `iter` and `loop`, and a recursion combinator `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 *iteration*, `iter`, whose implementation is shown in Figure 13. Given `body : A → itree E (A + B)` and a starting state `a:A`, `iter body a` is a computation that produces either a new state from which to iterate the `body` again (after a `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 **`cofix`**.

Defining fixpoint combinators as functions allows us to prove their general properties *once and for all*. The equations for `iter`, given below, imply that continuation trees form an iterative

RIGHTSLINK

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.

```
                iter f  ≈  f >>> case_ (iter f) id_                        (fixed point)
          iter f >>> g  ≈  iter (f >>> bimap id_ g)                         (parameter)
iter (f >>> case_ g inr_)  ≈  f >>> case_ (iter (g >>> case_ f inr_)) id_  (composition)
          iter (iter f)  ≈  iter (f >>> case_ inl_ id_)                     (codiagonal)
```

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:

```
Class MonadIter (M : Type → Type) `{Monad M} :=
  iter : ∀A B, (A → M (A + B)) → A → 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 (`_ → 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 transfomers, *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 : Type → Type` gives the signature of a recursive function, or, using multiple constructors, a block of mutually recursive functions. For example, `D := ackermannE` represents a function with two `nat` arguments and a result of type `nat`.

```
Inductive ackermannE : Type → Type :=
| Ackermann : nat → nat → ackermannE nat.
```

A *recursive event handler* for `D` is an event handler of type `D ⤳ itree (D +' 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_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`-ing `Ackermann` events, without any requirement to ensure the well-foundedness of the definition.

```
Definition h_ackermann : ackermannE ⤳ itree (ackermannE +' emptyE) :=
  fun _ '(Ackermann m n) ⇒ if m =? 0 then Ret (n + 1)
                           else if n =? 0 then trigger (inl1 (Ackermann (m-1) 1))
                           else (ack ← trigger (inl1 (Ackermann m (n-1))) ;;
```


RIGHTSLINK

```
(* Interpret an itree in the context of a mutually recursive definition (rh) *)
Definition mrec {D E} (rh : D ⇝ itree (D +' E)) : D ⇝ 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 ⇝ itree (D +' E)`, it produces a handler `D ⇝ 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 ⇝ 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)

RIGHTSLINK

```
(* Imp Syntax ---------------------------------- *)
Inductive expr : Set := ... (* omitted *)

Inductive imp : Set :=
| Assign (x : var) (e : expr)
| Seq    (a b : imp)
| If     (i : expr) (t e : imp)
| While  (t : expr) (b : imp)
| Skip.

(* Imp Events ---------------------------------- *)
Variant ImpState : Type → Type :=
| GetVar (x : var) : ImpState value
| SetVar (x : var) (v : value) : ImpState unit.

Context {E : Type → Type} `{ImpState -< E}.

(* Imp Denotational semantics ------------------ *)
(* ITree representing an expression *)
Fixpoint denote_expr (e:expr) : itree E value :=
 match e with
 | Var v    ⇒ trigger (GetVar v)
 | Lit n    ⇒ ret n
 | Plus a b ⇒ l ← denote_expr a ;;
               r ← denote_expr b ;; ret (l + r)
 | ...
 end.
```

```
(* Imp Denotational semantics cont'd ----------- *)
(* ITree representing an Imp statement *)
Fixpoint denote_imp (s : imp) : itree E unit :=
  match s with
  | Assign x e  ⇒ v ← denote_expr e ;; trigger (SetVar x v)
  | Seq a b     ⇒ denote_imp a ;; denote_imp b
  | If i t e    ⇒ v ← denote_expr i ;;
      if is_true v then denote_imp t else denote_imp e
  | While t b   ⇒
      iter (fun _ ⇒ v ← denote_expr t ;;
                    if is_true v
                    then denote_imp b ;; ret (inl tt)
                    else ret (inr tt))
  | Skip        ⇒ ret tt
  end.

(* Imp state monad semantics ------------------- *)
(* Translate ImpState events into mapE events *)
Definition h_imp_state {F: Type → Type} `{mapE var 0 -< F}
  : ImpState ~> itree F := ...(* omitted *)

(* Interpret ImpState into (stateT env (itree F)) monad *)
Definition interp_imp {F A} (t : itree (ImpState +' F) A)
  : stateT env (itree F) A :=
  let t' := interp (bimap h_imp_state id_) t in
    interp_map t'.
```

Fig. 15. Syntax and denotational semantics of Imp. The `While` case uses `iter`; `GetVar` and `SetVar` events are interpreted into the monad `stateT env (itree E)`, where `env` is a finite map from `var` to `value`.

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 `imp → env → (env * unit)`. However, it is not possible to give a semantics to `while` using this naïve 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 `imp → stateT env (itree F) unit`, or, equivalently, `imp → env → itree F (env * 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 `denote_expr` and `denote_imp`, which result in trees of type `itree E unit`. The typeclass constraint `ImpState -< E` indicates that `E` permits `ImpState` actions, a refinement of `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 `iter` combinator (see Section 4) to first run the guard expression, then either continue to loop (by returning `inl tt` to the `iter` combinator) or signal that it is time to stop (by returning `inr tt`).

The second stage of the semantics is `interp_imp`, which takes ITrees containing `ImpState` events and produces a computation in the state monad. It first invokes a handler for `ImpState`, `h_imp_state`,

RIGHTSLINK

```
(* Asm syntax ------------------------------------ *)
Variant instr : Set := ... (* omitted *)

Variant branch {label : Type} : Type :=
| Bjmp (_ : label)             (* jump to label *)
| Bbrz (_ : reg) (yes no : label) (* cond. jump *)
| Bhalt.

Inductive block (label : Type) : Type :=
| bbi (_ : instr) (_ : block) (* instruction *)
| bbb (_ : branch label).     (* final branch *)

Definition bks A B := fin A → block (fin B).

(* Control-flow subgraph: entries A and exits B. *)
Record asm (A B : nat) : Type :=
{ internal : nat
; code     : bks (internal + A) (internal + B) }.

(* Asm events ------------------------------------ *)
Variant Reg : Type → Type :=
| GetReg (x : reg) : Reg value
| SetReg (x : reg) (v : value) : Reg unit.

Inductive Memory : Type → Type :=
| Load  (a : addr) : Memory value
| Store (a : addr) (v : value) : Memory unit.

Context {E : Type → Type}.
Context `{Reg -< E} `{Memory -< E}.

(* Asm denotational semantics -------------------- *)
Definition denote_instr (i:instr) : itree E unit
  := ... (* omitted *)
```

```
(* Asm denotational semantics cont'd ------------------------ *)
Definition denote_br (b:branch (fin B)):itree E (fin B) :=
  match b with
  | Bjmp l ⇒ ret l
  | Bbrz v y n ⇒
      val ← trigger (GetReg v) ;;
      if val ?= 0 then ret y else ret n
  | Bhalt ⇒ exit
  end.

Fixpoint denote_bk {L} (b : block L) : itree E L :=
  match b with
  | bbi i b ⇒ denote_instr i ;; denote_bk b
  | bbb b   ⇒ denote_br b
  end.

Definition denote_bks (bs:bks A B):ktree E (fin A) (fin B)
  := fun a ⇒ denote_bk (bs a).

Definition den_asm {A B}:asm A B → ktree E (fin A) (fin B)
  := fun s ⇒ loop (denote_bks (code s)).

(* Asm state monad semantics --------------------------------- *)
Definition h_reg {F: Type → Type} `{mapE reg 0 -< F}
  : Reg ⇝ itree F := (* omitted *)
Definition h_mem {F: Type → Type} `{mapE addr 0 -< F}
  : Memory ⇝ itree F := (* omitted *)

Definition interp_asm {F A} (t:itree (Reg+'Memory+'F) A)
  : memory → registers → itree F (memory*(registers*A)) :=
  let h := bimap h_reg (bimap h_mem id_) in
  let t' := interp h t in
  fun mem regs ⇒ interp_map (interp_map t' regs) mem.
```

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.

RIGHTSLINK

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 `nat`s, 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):

```
Definition app_asm     (ab : asm A B) (cd : asm C D) : asm (A + C) (B + D).
Definition loop_asm    (ab_ : asm (I + A) (I + B))   : asm A B.
Definition pure_asm    (f : A → B)                   : asm A B.
Definition relabel_asm (f : A → B) (g : C → D) (bc : asm B C) : asm A D.
```

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.

RIGHTSLINK

```
Definition seq_asm {A B C} (ab : asm A B) (bc : asm B C): asm A C
  := loop_asm (relabel_asm swap id_ (app_asm ab bc)).

(* Auxiliary for if_asm *)
Definition cond_asm (e : list instr) : asm 1 (1 + 1)
  := ... (* omitted *)

Definition if_asm {A} (e : list instr) (t : asm 1 A) (f : asm 1 A)
  : asm 1 A
  := seq_asm (cond_asm e) (relabel_asm id_ merge (app_asm t f)).

Definition while_asm (e : list instr) (p : asm 1 1) : asm 1 1
  := loop_asm (relabel_asm id_ merge
               (app_asm (if_asm e
                         (relabel_asm id inl_ p)
                         (pure_asm inr_))
                        (pure_asm inl_))).
```

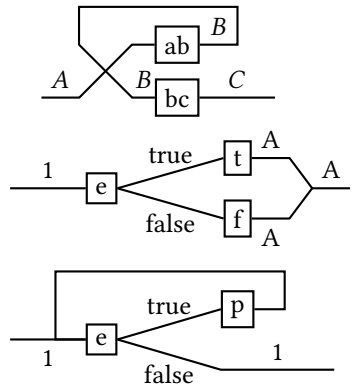

Fig. 17. High-level control flow in Asm

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.

```
den_asm (app_asm ab cd)   ≈̂  bimap (den_asm ab) (den_asm cd)
  den_asm (loop_asm ab)   ≈̂  loop (den_asm ab)
den_asm (relabel_asm f g bc) ≈̂  (pure f >>> den_asm bc >>> pure g)
  den_asm (pure_asm f)    ≈̂  pure_ktree f
```

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 `asm A B` with `asm B C` places them in parallel, swaps their entry labels to get a sub-CFG of type `asm (B+A) (B+C)`, and then internalizes the intermediate label `B` via `loop_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 `seq_asm` combinator is indeed the sequential composition of denotations of its arguments (up to `Tau`):

```
Lemma seq_asm_correct {A B C} (ab : asm A B) (bc : asm B C) :
  (den_asm (seq_asm ab cd)) ≈̂ (den_asm ab >>> den_asm bc).
```

The `while_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 `tmp_if` is non-zero after evaluating the expression `e`. This is expressed at the level of KTrees via the `loop` operator. In the code below, `label_case l` analyzes the shape of the label `l`, and `l1` and `l2` are two distinct label constants corresponding to the loop entry or exit, respectively.

```
Lemma while_asm_correct (e : list instr) (p : asm 1 1)
  : denote_asm (while_asm e p)
  ≈̂ loop (fun l:fin (1 + 1) ⇒
      match label_case l with
      | inl _ ⇒ denote_list e ;; v ← trigger (GetReg tmp_if) ;;
                if (v:value) then Ret l2 else (denote_asm p l1;; Ret l1)
      | inr _ ⇒ Ret l1
```


RIGHTSLINK

```
    end).
```

Most importantly, the proof of `while_asm_correct` is again purely equational, relying solely on the theory of KTrees and correctness equations of the low-level linking combinators (`app_asm_correct`, `seq_asm_correct`, *etc.*).

5.4 Compiler Correctness

The compiler itself is entirely straightforward. It compiles Imp statements using the linking combinators along with `compile_assign` and `compile_expr`, both of which are simple (and omitted). We pass to `compile_expr` the name of a target register (here, just `0`), into which the value of the expression will be computed; it stores intermediate results in additional Asm registers as needed.

```
Fixpoint compile (s : stmt) {struct s} : asm 1 1 :=
  match s with
  | 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 compile_correct (s : stmt) : equivalent s (compile s).
```

Figure 18 unpacks the definition of `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 `itree (ImpState +' E) A` and `itree (Reg +' Memory +' E) B`, are bisimilar if, when run in `Renv`-related initial states, they produce computations that are equivalent up to `Tau` and both terminate in states related by `state_invariant TT`. The relation `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 `state_invariant`. Here, `TT` is the trivial relation on the output label of Asm, since a statement has a unique exit point; in general the `RAB` relation parameter in `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 `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 `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 `interp_imp` and `interp_asm` that are induced by virtue of being compositionally defined from `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

RIGHTSLINK

```
(* Relate an Imp env to an Asm memory *)
Definition Renv (g_imp : Imp.env) (g_asm : Asm.memory) : Prop :=
  ∀ k v, alist_In k g_imp v ↔ alist_In k g_asm v.

Definition sim_rel l_asm n: (Imp.env * value) → (Asm.memory * (Asm.registers * unit)) → Prop :=
    fun '(g_imp', v) '(g_asm', (l_asm', _)) ⇒
      Renv g_imp' g_asm' ∧         (* we don't corrupt any of the Imp state *)
      alist_In n l_asm' v ∧        (* we get the right value in register n *)
      (∀ m, m < n → ∀v,  alist_In m l_asm v ↔ alist_In m l_asm' v).
                                   (* we don't mess with anything on the "stack" *)

Context {A B : Type}.           (* Imp / Asm intermediate result types *)
Context (RAB : A → B → Prop). (* Parameter that relates intermediate results *)

(* Relate Imp to Asm intermediate states. *)
Definition state_invariant (a : Imp.env * A) (b : Asm.memory * (Asm.registers * B)) :=
    Renv (fst a) (fst b) ∧ (RAB (snd a) (snd (snd b))).

Definition bisimilar {E} (t1 : itree (ImpState +' E) A) (t2 : itree (Reg +' Memory +' E) B) :=
    ∀ g_asm g_imp l, Renv g_imp g_asm
    → eutt (state_invariant RAB) (interp_imp t1 g_imp) (interp_asm t2 g_asm l).

(* Imp / Asm program equivalence *)
Definition TT : unit → fin 1 → Prop := fun _ _ ⇒ True.
Definition equivalent (s:stmt) (t:asm 1 1) : Prop := bisimilar TT (denote_stmt s) (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 `read_int` command and each `Output` event as a call to `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 `nat` type as OCaml's `int` type.
[8] Thanks to its dependent type, the OCaml extraction of `Vis` uses OCaml's `Obj.t` as the domain of the embedded continuations, so handlers should be written with care, otherwise type-safety could be jeopardized.
[9] http://www.deepspec.org.
[10] https://github.com/vellvm/vellvm

RIGHTSLINK

```
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

RIGHTSLINK

```
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 : ∀X (e : E X) k, is_trace_of (Vis e k) (TEventEnd e)
| TraceVisContinue : ∀X (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 : ∀X 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 nondeterminism 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 $\bot$ (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` $\sqsubseteq$ `u` iff $\forall$ `tr, is_trace_of t tr → is_trace_of u tr`.

Definition 2 (Trace Equivalence). `t` $\equiv$ `u` iff `t` $\sqsubseteq$ `u` and `u` $\sqsubseteq$ `t`.

Using these definitions, we can show that trace equivalence coincides with weak bisimulation, *i.e.*, that `t1` $\approx$ `t2` $\iff$ `t1` $\equiv$ `t2`.

Trace refinement shows that `spin` $\sqsubseteq$ `t`. Since all events in the ITree are visible in the trace, if `u` $\sqsubseteq$ `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 `Tau`s into/from `u`. In many cases, different notions of refinement may be more useful. For instance, one may want nondeterministic behavior to be

RIGHTSLINK

refined by possible deterministic behaviors. One way to do this is to directly allow the `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 `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 *free monad* to define monad instances modularly. Transporting his definition to our setting, we would obtain the following:

```
CoInductive Free (E : Type → Type) (R:Type) :=
| Ret : R → Free E R
| Vis : E (Free E R) → Free E R.
```

This version of the `Vis` constructor directly applies the functor `E` to the coinductively defined type `Free E R` itself. However, this type violates the strict positivity condition enforced by Coq: certain choices of `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 `Vis` constructor. The resulting "freer" monad (called `FFree` in their work) is essentially identical to our ITrees—the difference being that, because they work in Haskell, which admits nontermination by default, it needs no `Tau` constructor.

When considered up to strong bisimulation, ITrees form the free *completely iterative monad* [Aczel et al. 2003] with respect to a functor of the form `fun X ⇒ F X + X`, where the second component corresponds to `Tau` nodes. Quotiented by weak bisimulation, ITrees define a free *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 `mrec` in yet more general category-theoretic settings. The ITrees library makes such results concretely applicable to formally verified systems.

ITrees are a form of resumptions, which originated from concurrency theory [Milner 1975]. More precisely, ITrees can be obtained by applying a coinductive resumption monad transformer [Cenciarelli and Moggi 1993; Piròg 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

RIGHTSLINK

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 $\mathsf{op} \in \Sigma$, 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 `stateE` (Figure 8) defines an effect signature Σ, and its constructors `Get` and `Put s` define the operations. Plotkin and Pretnar used the notation $\mathsf{op}(x : X.\, M)$, corresponding to the ITrees `Vis op (fun x:X ⇒ 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 $\mathtt{get}(x : S.\, \mathtt{get}(y : S.\, kxy)) = \mathtt{get}(x : S.\, kxx)$. 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 $\mathtt{handler}\{\mathtt{return}\ x \mapsto f(x), (\mathsf{op}(y; \kappa) \mapsto h(y, \kappa))_{\mathsf{op} \in \Sigma}\}$. 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 `Vis` constructor, which is a function of type `∀ X, E X → (X → itree E R) → M R` for `M` an iterative monad. However, Coq prevents us from creating a general-purpose interpeter 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 `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 ($\bot \sqsubseteq t$ 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.

RIGHTSLINK

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 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.

```
Inductive General (S:Set) (T : S → Set) (X : Set) : Set :=
| RetG (x : X)
| VisG (s:S) (k : T s → General S T X).
```

Its interface replaces our single `E : Type → Type` parameter with `S : Type` and a type family `S → 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.

RIGHTSLINK

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:

```
CoInductive world : Type :=
  World (io : string → list eventval → option (eventval * world))
```

Here the `string` and list of `eventval`s 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:

```
CoInductive world E : Type := World (io : ∀{A:Type}, E A → option (A * 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,

RIGHTSLINK

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 `Tau`s represent internal steps, which may lead to divergence. Though, as we showed in Section 5, users of ITrees are mostly insulated from `Tau`s 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

RIGHTSLINK

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

Andreas Abel, Stephan Adelsberger, and Anton Setzer. 2017. Interactive programming in Agda–Objects and graphical user interfaces. *Journal of Functional Programming* 27 (2017).

Peter Aczel, Jirí Adámek, Stefan Milius, and Jiri Velebil. 2003. Infinite trees and completely iterative theories: a coalgebraic view. *Theor. Comput. Sci.* 300, 1-3 (2003), 1–45. https://doi.org/10.1016/S0304-3975(02)00728-4

RIGHTSLINK

Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University. http://www.cs.indiana.edu/~amal/ahmedsthesis.pdf

Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. 2017. Partiality, Revisited. In *Foundations of Software Science and Computation Structures*, Javier Esparza and Andrzej S. Murawski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 534–549.

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

Andrew W. Appel. 2011. Verified Software Toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software (ESOP'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 1–17. http://dl.acm.org/citation.cfm?id=1987211.1987212

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

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (Jan 2015), 108–123.

Nick Benton, Lars Birkedal, Andrew Kennedy, and Carsten Varming. 2010. Formalizing Domains, Ultrametric Spaces and Semantics of Programming Languages. (July 2010). https://www.microsoft.com/en-us/research/publication/formalizing-domains-ultrametric-spaces-and-semantics-of-programming-languages/

Nick Benton, Andrew Kennedy, and Carsten Varming. 2009. Some Domain Theory and Denotational Semantics in Coq. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 115–130.

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

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* 1, 2 (2005), 1–18. https://doi.org/10.2168/LMCS-1(2:1)2005

Robert Cartwright and Matthias Felleisen. 1994. Extensible Denotational Language Specifications. In *Symposium on Theoretical Aspects of Computer Software*, Vol. LNCS. Springer-Verlag, 244–272.

Pietro Cenciarelli and Eugenio Moggi. 1993. *A Syntactic Approach to Modularity in Denotational Semantics.* Technical Report. In Proceedings of the Conference on Category Theory and Computer Science.

James Chapman, Tarmo Uustalu, and Niccolò Veltri. 2015. Quotienting the Delay Monad by Weak Bisimilarity. In *Theoretical Aspects of Computing - ICTAC 2015*, Martin Leucker, Camilo Rueda, and Frank D. Valencia (Eds.). Springer International Publishing, Cham, 110–125.

Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems (ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 41–60. https://doi.org/10.1007/978-3-642-37036-6_3

Adam Chlipala. 2007. A Certified Type-preserving Compiler from Lambda Calculus to Assembly Language. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 54–65. https://doi.org/10.1145/1250734.1250742

Adam Chlipala. 2010. A Verified Compiler for an Impure Functional Language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 93–106. https://doi.org/10.1145/1706299.1706312

Adam Chlipala. 2017. Infinite Data and Proofs. In *Certified Programming with Dependent Types*. MIT Press. http://adam.chlipala.net/cpdt/html/Cpdt.Coinductive.html

Nils Anders Danielsson. 2012. Operational semantics using the partiality monad. In *In: International Conference on Functional Programming 2012, ACM Press*. Citeseer.

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description).. In *CADE (Lecture Notes in Computer Science)*, Amy P. Felty and Aart Middeldorp (Eds.), Vol. 9195. Springer, 378–388. http://dblp.uni-trier.de/db/conf/cade/cade2015.html#MouraKADR15

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

Carlos Eduardo Giménez. 1996. *Un Calcul De Constructions Infinies Et Son Application A La Verification De Systemes Communicants.* Ph.D. Dissertation. École Normale Supérieure de Lyon.

Eduardo Giménez. 1995. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs*, Peter Dybjer, Bengt Nordström, and Jan Smith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 39–59.

Sergey Goncharov and Lutz Schröder. 2011. A Coinductive Calculus for Asynchronous Side-effecting Processes. *CoRR* abs/1104.2936 (2011). arXiv:1104.2936 http://arxiv.org/abs/1104.2936

RIGHTSLINK

Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Maciej Piróg. 2017. Unifying Guarded and Unguarded Iteration. In *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 517–533. https://doi.org/10.1007/978-3-662-54458-7_30

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified concurrent abstraction layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 646–661. https://doi.org/10.1145/3192366.3192381

Tatsuya Hagino. 1989. Codatatypes in ML. *Journal of Symbolic Computation* 8, 6 (1989), 629 – 650. https://doi.org/10.1016/S0747-7171(89)80065-3

Peter Hancock. 2000. *Ordinals and interactive programs*. Ph.D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/376

Peter Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Computer Science Logic*, Peter G. Clote and Helmut Schwichtenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 317–331.

Masahito Hasegawa. 1997. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *Typed Lambda Calculi and Applications*, Philippe de Groote and J. Roger Hindley (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 196–213.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 1–17. https://doi.org/10.1145/2815400.2815428

Aquinas Hobor. 2008. *Oracle Semantics*. Ph.D. Dissertation. Princeton, NJ, USA. Advisor(s) Appel, Andrew W. AAI3333851.

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/2429069.2429093

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

P. Johann, A. Simpson, and J. Voigtländer. 2010. A Generic Operational Metatheory for Algebraic Effects. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*. 209–218. https://doi.org/10.1109/LICS.2010.29

André Joyal, Ross Street, and Dominic Verity. 1996. Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society* 119, 3 (1996), 447–468. https://doi.org/10.1017/S0305004100074338

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

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 59–70.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2019)*. ACM, New York, NY, USA, 234–248. https://doi.org/10.1145/3293880.3294106

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 179–192. https://doi.org/10.1145/2535838.2535841

Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing in Coq*. Electronic textbook. https://softwarefoundations.cis.upenn.edu/qc-current/index.html

RIGHTSLINK

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Information and Computation* 207, 2 (2009), 284 – 304. https://doi.org/10.1016/j.ic.2007.12.004 Special issue on Structural Operational Semantics (SOS).

Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. 2018. Modular Verification of Programs with Effects and Effect Handlers in Coq. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*. 338–354. https://doi.org/10.1007/978-3-319-95582-7_20

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

Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. 2010. Toward a Verified Relational Database Management System. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/1706299.1706329

Coq development team. 2018. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.8.1.

Coq development team. 2019. *The Coq proof assistant reference manual. The Gallina specification language. Co-inductive types, Caveat*. LogiCal Project. https://coq.inria.fr/distrib/V8.9.0/refman/language/gallina-specification-language.html#caveat Version 8.9.0.

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

Stefan Milius. 2005. Completely iterative algebras and completely iterative monads. *Inf. Comput.* 196, 1 (2005), 1–41. https://doi.org/10.1016/j.ic.2004.05.003

Robin Milner. 1975. Processes: A Mathematical Model of Computing Agents. In *Logic Colloquium '73*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 157 – 173. https://doi.org/10.1016/S0049-237X(08)71948-7

Eugenio Moggi. 1989. Computational lambda-calculus and monads. 14–23. Full version, titled *Notions of Computation and Monads*, in Information and Computation, 93(1), pp. 55–92, 1991.

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

Keiko Nakata and Tarmo Uustalu. 2010. Resumptions, Weak Bisimilarity and Big-Step Semantics for While with Interactive I/O: An Exercise in Mixed Induction-Coinduction. In *Proceedings Seventh Workshop on Structural Operational Semantics, SOS 2010, Paris, France, 30 August 2010*. 57–75. https://doi.org/10.4204/EPTCS.32.5

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg.

Ulf Norell. 2007. Towards a practical programming language based on dependent type theory.

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 589–615.

Simon L Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Papers Presented at the Symposium*. ACM Press.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2018. *Logical Foundations*. Electronic textbook. Version 5.5. http://www.cis.upenn.edu/~bcpierce/sf.

Maciej Piròg and Jeremy Gibbons. 2014. The Coinductive Resumption Monad. *Electronic notes in theoretical computer science*. 308 (2014), 273–288.

Gordon Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Foundations of Software Science and Computation Structures*, Furio Honsell and Marino Miculan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–24.

Gordon Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures*, Mogens Nielsen and Uffe Engberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 342–356.

Gordon D Plotkin. 2004a. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming* 60-61 (2004), 3 – 15. https://doi.org/10.1016/j.jlap.2004.03.009 Structural Operational Semantics.

Gordon D. Plotkin. 2004b. A structural approach to operational semantics. *J. Log. Algebr. Program.* 60-61 (2004), 17–139.

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94.

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

RIGHTSLINK

Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. 2016. *Monad Transformers and Algebraic Effects: What binds them together.* Technical Report CW699. Department of Computer Science, KU Leuven.

Anton Setzer. 2006. Object-oriented programming in dependent type theory. *Trends in functional programming*. 7 (2006).

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

Wouter Swierstra. 2008. Data Types à la Carte. *Journal of Functional Programming* 18, 4 (2008), 423–436.

Tarmo Uustalu and Niccolò Veltri. 2017. The Delay Monad and Restriction Categories. In *Theoretical Aspects of Computing – ICTAC 2017*, Dang Van Hung and Deepak Kapur (Eds.). Springer International Publishing, Cham, 32–50.

Philip Wadler. 1992. Monads for functional programming. In *Program Design Calculi, Proceedings of the NATO Advanced Study Institute on Program Design Calculi, Marktoberdorf, Germany, July 28 - August 9, 1992*. 233–264. https://doi.org/10.1007/978-3-662-02880-3_8

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 357–368. https://doi.org/10.1145/2737924.2737958

RIGHTSLINK