Dijkstra Monads Forever: Termination-Sensitive Specifications for Interaction Trees

LUCAS SILVER, University of Pennsylvania, USA
STEVE ZDANCEWIC, University of Pennsylvania, USA

This paper extends the Dijkstra monad framework, designed for writing specifications over effectful programs using monadic effects, to handle termination sensitive specifications over interactive programs. We achieve this by introducing base specification monads for non-terminating programs with uninterpreted events. We model such programs using interaction trees, a coinductive datatype for representing programs with algebraic effects in Coq, which we further develop by adding trace semantics. We show that this approach subsumes typical, simple proof principles. The framework is implemented as an extension of the Interaction Trees Coq library.

CCS Concepts: • **Theory of computation → Logic and verification**; **Programming logic**; **Hoare logic**; **Program specifications**; **Pre- and post-conditions**; **Program verification**; *Invariants*.

Additional Key Words and Phrases: coinduction, specification, verification, monads, Hoare logic

ACM Reference Format:
Lucas Silver and Steve Zdancewic. 2021. Dijkstra Monads Forever: Termination-Sensitive Specifications for Interaction Trees. *Proc. ACM Program. Lang.* 5, POPL, Article 26 (January 2021), 28 pages. https://doi.org/10.1145/3434307

1 INTRODUCTION

Formal verification of real systems is becoming viable in practice, as demonstrated by many recent examples including databases, operating systems, network servers, and compilers and programming-language semantics. These successes have been made possible by advances in our understanding of how to represent and reason about interactive systems using interactive theorem provers (like Isabelle/HOL and Coq) and dependently-typed programming languages (like F* [Swamy et al. 2011] and Agda [Norell 2007]).

To date, numerous proof frameworks have been developed to facilitate these kinds of verification efforts. In the realm of Coq, prominent examples include YNot [Malecha et al. 2011], FCSL [Sergey et al. 2015], VST [Appel 2014] and Iris [Jung et al. 2016]. In the context of F*, Dijkstra monads [Swamy et al. 2013] have been shown to be an effective way to reason about (impure) programs. Despite the success of these approaches—these tools have been used to verify the correctness of applications ranging from concurrent mailbox protocols, to cryptography primitives [Protzenko et al. 2020], to Rust library code [Jung et al. 2017]—there remain improvements to be made.

For one thing, most of these systems are rooted in Floyd-Hoare logic, which makes them most naturally suited to proving *partial correctness*. Termination and other liveness properties are considered only separately, outside the framework, or not at all. (A notable exception is Hoffman *et.*

Authors' addresses: Lucas Silver, lucsil@seas.upenn.edu, University of Pennsylvania, Philadelphia, PA, USA; Steve Zdancewic, stevez@cis.upenn.edu, University of Pennsylvania, Philadelphia, PA, USA.

2475-1421/2021/1-ART26
https://doi.org/10.1145/3434307

al.'s work on verified resource analysis [Carbonneaux et al. 2017].) For another, proof support for *interactive programs*—programs that exchange data with their environments—remains challenging, because the possibility of such interactions complicates both specifications and reasoning. On top of these issues, the *way* in which the program semantics is represented also matters. For instance, relationally-specified operational semantics, as used in VST (via CompCert) and in Iris, cannot be executed—it's not possible to extract an executable program from the semantics described that way. This makes such frameworks incompatible with tools like QuickChick [Lampropoulos and Pierce 2018], which requires executable specifications for testing; it also precludes their use for reasoning about Coq programs (or domain specific languages shallowly-embedded in Coq). Such representation choices matter in practice.

This paper describes a framework for verifying *termination-sensitive* properties of possibly divergent, *interactive* programs in the proof assistant Coq. The first ingredient is the representation of such programs as *interaction trees* (ITrees), following the work of Xia, *et. al.* [2020]. ITrees provide a general-purpose way of defining the semantics of impure, possibly diverging computations, while retaining extractability (and hence executability). The second ingredient is an extension of the methodology for deriving Dijkstra specification monads, proposed by Maillard, *et. al.* [2019]. Dijkstra monads are a natural fit for ITrees: the core ITree datatype itself is a monad, and we express ITree computations by interpreting events into monadic operations, yielding a computation type built out of a stack of monad transformers. Previous work on Dijkstra monads finessed the issue of nonterminating programs—their computation types bottom out in the Identity monad. In contrast, the monads we investigate in this paper bottom out in the ITree monad for some event type family `E`. We pay special attention to the case of Capretta's Delay monad [Capretta 2005], which is what remains of an ITree once all of its externally-visible events have been interpreted away.

The Delay monad precisely characterizes nonterminating behaviors, which is what grants ITrees their expressiveness, but it also means that we must take divergence into account when reasoning about them—this is one significant challenge that we show how to address in this paper. The reward for this effort is that we obtain termination-sensitive specifications that are more expressive than those available with Floyd-Hoare-style partial correctness assertions.

As a simple example, consider the following imperative program that computes the square root of the natural number `n`:

```
Definition nat_sqrt : com :=
  i ::= 0;;;
  WHILE ~(i * i = n) DO i ::= i + 1 END.
```

A partial correctness specification of this program says *"If the program terminates, then* `i * i = n`*."* The stronger specification that we are able to prove instead says *"If there exists a natural number* `k` *such that* `k * k = n`*, then the program terminates and* `i = k`*; otherwise the program diverges."* While partial correctness assertions are suitable for many scenarios, termination-sensitive specifications are essential for proving liveness and availability properties.

Aside from accounting for nontermination, ITrees also permit the description of interactive programs—"uninterpreted" events in an ITree can be viewed as calls to the environment. Combining nontermination with such interaction leads to technical challenges both for defining useful notions of specification—what does it mean to say that an interactive computation meets some specification?—and for establishing metatheoretic properties—like soundness—of the reasoning principles. This is a second significant challenge that we address in this paper though the introduction of *trace specifications*.

As a simple example trace specification, consider the following interactive program that repeatedly queries its environment for a boolean value, stopping only when the result is `false`:

```
Definition queryUntilFalse := WHILE query() DO SKIP.
```

One provably correct specification of this program's behavior says *"Either the program diverges and the environment supplied an infinite stream of* `trues`*; or, the program halts and the environment provided some finite number of* `trues` *followed by* `false`*"*.

In summary, this paper makes the following contributions.

- In Section 4 we show how to extend prior work on Dijkstra Monads to account for the (potentially) nonterminating behavior allowed by the Delay monad. In doing so, we define `DelaySpec`, a specification monad suitable for expressing properties about Delay computations. This lets us apply the Dijkstra monad methodology to derive an appropriate specification monad for computations that combine state and nontermination.
- ITree computations, in addition to the usual monadic `bind` and `ret` operations, support looping via a (typeclass overloaded) family of iteration combinators. In Section 5, we prove the correctness of loop-invariant-style reasoning principles for such `iter` combinators, focusing on the cases of the `Delay` monad and `StateDelay`, its lifting through the state monad transformer.
- Building on `DelaySpec`, Section 6 defines Floyd-Hoare logic-style specifications, with pre- and post-conditions, and accompanying rules for reasoning about `StateDelay` computations. As in the example above, the natural definition is stronger than either the usual "total correctness" or "partial correctness" interpretations of Floyd-Hoare-triples—the post conditions can talk about the termination behavior of the computation explicitly (unlike total correctness, which implies termination, or partial correctness, which assumes it). Moreover, we show how to recover soundness proofs of the usual partial-correctness Hoare logic rules for a simple imperative language whose semantics is defined denotationally via ITrees.
- To reason about interactive ITree computations, we must go beyond the `Delay` and `StateDelay` monads. Unfortunately, the "obvious" generalization of `DelaySpec` to full ITrees fails to satisfy the monad laws, and is therefore unsuitable as a specification monad. The crux of the problem is that `bind` uses a continuation that takes only an element of the parameter type and completely ignores the sequence of events that lead to producing that element. This motivates us to seek a specification in terms of *traces* of an ITree. Section 7 develops the technical machinery needed to define the type of such traces as just another instance of ITrees proper. These traces are connected to ITrees by way of a general notion of ITree *refinement*, which is itself implemented in terms of a generalization of the standard weak simulation relation for ITrees.
- With the above definitions in hand, Section 8 defines `TraceSpec`, a Dijkstra Monad suited to reasoning about possibly nonterminating, interactive ITree programs. Proving the soundness of such specifications is non-trivial and somewhat technical—doing so requires us to build suitable simulation relations to establish that `TraceSpec` is a monad morphism. Once that is done, however, such specifications soundly expose the behavior of the program as a "log" of its interactions with the environment, providing a convenient abstraction for verifying properties like the one about the `queryUntilFalse` example above.

All of the results mentioned above have been formally verified in Coq, and the framework is designed to be used with the Interaction Trees Library.[1]

Our approach to defining Dijkstra Monads for ITrees is inspired by prior research in this area, especially that of Maillard, *et. al.* [2019]. We recapitulate the most relevant aspect of that work as needed throughout the paper; Section 10 gives a deeper comparison to that paper and other related

[1]See https://github.com/DeepSpec/InteractionTrees.

```
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 *)

Definition ret {E R} x : itree E R := Ret x.

Definition bind {E R S} (t : itree E R) (k : R → itree E S) : itree E S := (* omitted *)

Definition eutt {E A B} (R : A → B → Prop) : itree E A → itree E B → Prop := (* omitted *)

Lemma eutt_ret : RR r s → eutt RR (Ret r) (Ret s).
Lemma eutt_vis : (∀ a, eutt RR (k1 a) (k2 a)) → eutt RR (Vis e k1) (Vis e k2).
Lemma eutt_taulr : eutt RR t1 t2 → eutt RR (Tau t1) (Tau t2).
Lemma eutt_taul : eutt RR t1 t2 → eutt RR (Tau t1) t2.
Lemma eutt_taur : eutt RR t1 t2 → eutt RR t1 (Tau t2).

Notation "t1 ≈ t2" := (eutt eq).

Class MonadIter (M : Type → Type) `{Monad M} := iter : ∀A B, (A → M (A + B)) → A → M B.
Instance iter_itree {E} : MonadIter (itree E) := (* omitted *)

Definition interp {E M : Type → Type} `{MonadIter M} {R : Type}
         (handler : ∀A, E A → M A) : itree E R → M R := (* omitted *)

CoFixpoint spin {E : Type → Type} {A : Type}: itree E A := Tau spin.

Inductive voidE : Type → Type := .

Definition Delay (A : Type ) := itree voidE A.
```

Fig. 1. Interaction Trees: Key Types, Operations, and Eutt Properties.

work. Before diving into the details of `DelaySpec` and `TraceSpec`, we first give some background about interaction trees (Section 2) and Dijkstra monads (Section 3).

2 INTERACTION TREES

Interaction trees (ITrees) are data structures intended for representing impure computations in Coq. As shown in Figure 1, they are implemented as a coinductive type[2], parameterized over a result type `R` and an event type family `E : Type → Type`. The `Ret` constructor injects pure results of type `R` into the type of ITrees. The `Tau` constructor represents a silent step of computation at the beginning of an ITree. The `Vis` constructor takes an uninterpreted event `e : E A`, for some type `A`, and a continuation `k : A → itree E R` that should be viewed as the computation to run once the environment has responded to the event `e`. Intuitively, the `Vis` nodes of an ITree correspond to "visible" events (a.k.a. uninterpreted effects) that interact with the ITree's environment to produce a result. The event parameter `E` defines the interface for those interactions. ITrees, and proofs involving them, are implemented using the paco library [Hur et al. 2013] which provides semantic coinductive reasoning principles much easier to work with than Coq's native coinduction.

The type `itree E` is a monad for any `E`. Its return (`ret`) is just the `Ret` constructor, and `bind m k` grafts subtrees generated by `k` at the leaves of `m`. Crucially, ITrees also support looping constructs. The ITrees library characterizes monads that support iteration using the `MonadIter` type class, of which `itree E` is an instance. Intuitively, `iter body a` acts like a do–while loop: it runs the `body`

[2]This presentation is simplified from the ITrees library, which uses an isomorphic definition in terms of so-called negative coinductive types.

computation starting from state `a`, which produces either `inl a'`, signaling that the loop should continue from `a'` or `inr b`, signaling that the loop should halt and produce the result `b`.

The interaction tree datatype is a coinductive cousin of a Free monad [Swierstra 2008], also known as the General [McBride 2015] or Program [Letan et al. 2018] monads. The events of an ITree gain meaning when interpreted into some other monad `M`. This functionality is provided by the `interp` function, which takes a handler that maps events into `M` and lifts it to entire ITrees.

The ITree datatype also generalizes the Delay monad [Capretta 2005]. `Delay A` is just `itree voidE A`, in which the event interface is empty. This type contains only terminating pure computations of the form `Ret r`, possibly with `Tau` nodes in front, and the silently divergent computation `spin`, which is an infinite stream of `Tau`'s.

ITrees come equipped with a notion of heterogeneous weak bisimulation, `eutt RR` ("equivalence up to taus modulo relation RR"), that defines what it means for two ITrees to offer the same interactions with an environment using `RR` to compare return values. We write `t1 ≈ t2`, defined as `eutt eq`, to mean that `t1` and `t2` are homogeneously weakly bisimilar. Intuitively, `t1` and `t2` differ only up to adding or removing any finite number of `Tau` nodes. Note that `spin` $\not\approx$ `Tau`^{n}`(Ret a) ≈ Ret a` for any natural number n. In Figure 1 we provide several lemmas demonstrating how `eutt` interacts with various `itree` constructors.

2.1 Eutt Div

We will now introduce a specialization of the `eutt` relation named `eutt_div` that is designed to equate divergent trees with equivalent computational content yet that have different types. Consider the following ITree.

```
CoFixpoint output_unit : itree IO unit :=
  Vis (Output 5) (fun _ ⇒ output_unit).
```

This denotes an `IO` computation that outputs the number `5` repeatedly, forever with return type `unit`. However, because the computation cannot, under any circumstances, return any value, we are free to give it any return type. The following code also type checks.

```
CoFixpoint output_nat : itree IO nat :=
  Vis (Output 5) (fun _ ⇒ output_nat).
```

In fact, we could give any parameter type `T`. As you can see, these computations contain identical code, but the proposition `output_unit ≈ output_nat` is ill-typed. We can relate trees like this with the following relation.

```
Definition eutt_div {E A B} := @eutt E A B (fun x y ⇒ False).
```

Recall that `bind m k` grafts subtrees generated by `k` onto the leaves of `m`. Because trees that are divergent along all paths have no leaves, `bind` has no meaningful effect on those trees. The `bind` just keeps getting coinductively pushed further along as the result is observed. To take advantage of this, we define the following function

```
Definition div_cast {E R S} (t : itree E R) : itree E S :=
  bind t (fun _ ⇒ spin).
```

This function acts on divergent trees by "casting" its parameter type to any arbitrary type without changing its events. We prove this by simple coinduction in the following lemma.

```
Lemma div_cast_eutt : ∀E A B (t : itree E A),
   diverges t → @eutt_div E A B t (div_cast t).
```

Because `fun x y ⇒ False` is a subrelation of all relations `R`, `eutt_div` is a subrelation of `eutt R`. Therefore, if we need to prove that two divergent trees are bisimilar, it suffices to prove that they are related by `eutt_div`. This is often an easier task because `bind` does not change a divergent tree

```
(* IMP commands *)
Inductive com : Set :=
  | CSkip : com                  (* SKIP *)
  | CAss  : var → aexp → com       (* X := a *)
  | CSeq  : com → com → com        (* c1 ;;; c2 *)
  | CIf   : bexp → com → com → com (* TEST b THEN c1 ELSE c2 FI *)
  | CWhile : bexp → com → com.     (* WHILE b DO c END *)

(* IMP Events *)
Inductive ImpE : Type → Type :=
| GetE (x : var) : ImpE nat
| SetE (x : var) (v : nat) : ImpE unit.

Definition while (step : itree ImpE (unit + unit)) : itree ImpE unit :=
   iter  (fun _ ⇒ step) tt.

Fixpoint denote_imp (c : com) : itree ImpE unit :=
  match c with
  | CSkip      ⇒ ret tt
  | CAss x a   ⇒  v  ← denote_aexp a ;; trigger (SetE x v)
  | CSeq c1 c2 ⇒  denote_imp c1 ;; denote_imp c2
  | CIf b c1 c2 ⇒ v  ← denote_bexp b ;;
                 if (v:bool) then denote_imp c1 else denote_imp c2
  | CWhile b c ⇒
     while (v  ← denote_bexp b ;;
          if (v:bool) then denote_imp c ;; ret (inl tt) else ret (inr tt))
  end.

Definition handle_ImpE : ∀X, ImpE X → stateT st Delay X := (* omitted *)

Definition interp_imp (t : itree ImpE unit) : stateT st Delay unit :=
  interp handle_ImpE t.
```

Fig. 2. IMP denotational semantics (excerpt).

with respect to eutt_div. As we will see in Section 8, div_cast is an essential tool in defining the specification monad for arbitrary event ITrees.

2.2 Denotational Semantics of IMP

To see how ITrees are used, we present the denotational semantics for a simple stateful language using ITrees. A typical path from a language syntax to an ITrees semantics first creates a denotation function from programs to ITrees over an event type with constructors for every effect in the language, and then interprets that ITree into a monad that is a stack of effect monad transformers applied to itree E for some event type family E.

Figure 2 gives (an excerpt of) a denotational semantics for a simple imperative language called IMP (adapted from Software Foundations [Pierce et al. 2018]). The type com defines the syntax of commands, which include SKIP, variable assignment, sequential composition, conditionals, and WHILE loops. The events interface ImpE defines the GetE and SetE events, which model reading and writing to the state. The function denote_imp builds an ITree with ImpE events. The case for assignment uses trigger to create a SetE node. (GetE events are used to read from the global state in the denotations of expressions, which are omitted here.) The denotations of sequential composition and conditionals are built straightforwardly from the ITree's bind operator. CWhile is defined using while, which is a simple wrapper around the ITrees iter combinator.

```
bind (ret a) f ≈ f a
bind m ret ≈ m
bind (bind m f) g ≈ bind m (fun a ⇒ bind (f a) g)
```

Fig. 3. Monad Laws

Once the syntax has been given an ITrees denotation, it is straightforward to complete the semantics by interpreting its events into an appropriate state monad. Here, `st` is a type of maps from `vars` to `nats`. The type of `handle_ImpE` shows that the resulting computation lives in the monad `stateT st Delay unit`, which is equal to `st → Delay (st * unit)`. There are no residual effects: and when given an initial state, an IMP program either diverges or terminates, yielding some updated state (and the unit value).

The ITrees library contains utilities for lifting the equational theory of ITrees through interpreters such as `interp_imp`, which allows for complex, termination sensitive properties of these languages to be proven without any explicit coinduction.

3 DIJKSTRA MONADS

Dijkstra Monads [Maillard et al. 2019; Swamy et al. 2013] are a flexible approach to the specification and verification of effectful programs modeled with monads. They can represent specifications over a wide range of algebraic effects including state, exceptions, and IO. A Dijkstra Monad comes about from the interaction of three objects: (1) A computational monad `M`, which is the type of programs whose properties we want to specify; (2) A specification monad `W`, which is the type of specifications and comes equipped with an order, denoted ≤ that captures the notion of specification refinement; and (3) An effect observation `obs`, which is a function with type `∀ A, M A → W A` which maps programs to the most precise specification that they satisfy. With these pieces in place, and given a particular specification `w` we can define the Dijkstra Monad type.

```
Definition DijkstraMonad (M W : Monad) (obs : ∀A, M A → W A) (A : Type)
  (w : W A) := { m : M A | obs m ≤ w }.
```

This is implemented as a sigma type in Coq. Computation `m` satisfies specification `w` when the above type instantiated with `m` and `w` is inhabited. This type pairs the `m` with the proof that it satisfies `w`. We streamline that by introducing the `DijkstraVerify` type, which has the same parameters but produces the verification condition as a proposition.

```
Definition DijkstraVerify (M W : Monad) (obs : ∀A, M A → W A) (A : Type)
  (w : W A) (m : M A) := obs m ≤ w.
```

The job of the proof engineer is then to write a proof of that verification condition. Previous work provides definitions for `ret` and `bind` for the `DijkstraMonad` type, making it a monad-like structure itself. However, we do not take advantage of this structure and simply use the type to generate verification conditions.

3.1 Specification Monads

The type of specifications in this framework forms a monad with extra structure. They also require an ordering relation ≤ that represents specification refinement.

Given two specifications `w1` and `w2`, `w1 ≤ w2` if all computations that satisfy `w1` must also satisfy `w2`. A specification monad's `bind` function must be monotonic with respect to ≤ in the following way.

```
Lemma monot_bind : ∀(A B : Type) (w1 w2 : W A) (f1 f2 : A → W B),
  w1 ≤ w2 → (∀ a, f1 a ≤ f2 a) → bind w1 f1 ≤ bind w2 f2.
```

The≤ order induces an equivalence on the type of specifications. Specifically, `w1 ≈ w2` if and only if `(w1 ≤ w2) ∧ (w2 ≤ w1)`. Using this equivalence we can define, and prove, the monad laws, as stated in Figure 3, for specification types.

The effect observation `obs` must also be a monad morphism, which means that it respects the structure of both `M` and `W` as follows[3]:

`obs (ret a) ≈ ret a` and `obs (bind w f) ≈ bind (obs w) (fun a ⇒ obs (f a))`

3.2 Base Specification Monads

A key insight of Maillard *et. al.* [2019] is identifying appropriate base specification monads for the identity monad and showing that if we have a computation type equal to a series of monad transformers applied to the identity monad, we can get a specification monad for the whole type by applying the same sequence of transformations to that base specification monad.

The base specification monad primarily used in that work is the type:

`IDSpec A := (A → Prop) → Prop.`

We restrict our attention to monotonic specifications. Exactly what this means and what it accomplishes is discussed in Section 4. The `IDSpec` type is the type of continuations into propositions. It contains elements like `ret a = fun p ⇒ p a`, the set of predicates that accept `a`, `fun p ⇒ True`, the set of all predicates, and `fun p ⇒ ∀ n, even n → p n`, the set of all supersets of the even numbers.

The effect observation from the identity monad to the ID specification type is

`obs a : IDSpec A := fun p ⇒ p a.`

The refinement relation `w1 ≤ w2` is the proposition `∀ p, w2 p → w1 p`. This is counterintuitive, as one might expect the implication to be in the opposite direction. However, this version is correct. Suppose we have a computation `m` and a specification `w`. To say that `m` satisfies `w` is the same as saying that the observation of `m` refines `w`. Observe how that statement reduces[4].

`(obs m≤ w) ↠ (∀ p, w p → obs m p) ↠ (∀ p, w p → p m)`

The final proposition captures the notion that `m` satisfying `w` means that `w` is a set of properties that hold on `m`. In other words, `w` contains no properties that exclude `m`.The fact that `w` is in the assumptions corresponds to the fact that specifications generally ignore many properties about a computation.

With `IDSpec` as a base specification monad, we can construct specification monads for more expressive computations. Applying the state monad transformer to `IDSpec` yields the type

`StateSpec S A := S → ((S * A) → Prop) → Prop.`

We can also derive an ordering relation and an effect observation for this new specification type on top of the ones provided by `IDSpec`.

`w1 ≤ w2 := ∀ s, w1 s≤ w2 s` `obs m s := obs (m s)`

Note that `StateSpec S A` is isomorphic to `((S * A) → Prop) → (S → Prop)`, the type of functions from postconditions to preconditions over stateful computations. With this in mind, the derived effect observation is the weakest precondition function.

So for example, the function `fun s ⇒ put x 1 s` yields as its weakest precondition the set of pairs of predicates `p` and states `s` such that `p (put x 1 s)`.

[3]Note that we are overloading `ret` and `bind` to refer to the corresponding functions in the two different monads.
[4]The ↠ symbol is used to mean "reduces to".

4 DELAY SPECIFICATION MONAD

Maillard *et. al.* [2019] uses the identity monad as the simplest form of computation. However, this is insufficient because computations in Coq are strongly normalizing and all elements of Coq's identity monad terminate. We need to use coinductive types in order to represent possibly divergent computations. First, we will investigate the usage of the `Delay` monad to model these computations. This is sufficient for specifying programs that can be modeled by interpreters whose types are stacks of monad transformers that bottom out leaving no uninterpreted events. Later, we will provide tools to deal with programs with external, uninterpreted events like IO or specific system calls.

4.1 Type Definition

As we saw above, to define a Dijkstra monad, we need to define our computational monad, our specification monad, and an effect observation mapping computations to the most specific specifications that they satisfy. For the computational monad, we use the Delay monad, implemented by the `itree voidE` type. To a first approximation, our specification monad is the type of backwards predicate transformers of the Delay monad, `(Delay A → Prop) → Prop` We consider only predicates that respect the ≈ weak bisimulation equivalence relation. We formalize this notion as follows.

```
Definition resp_eutt {A : Type} (p : Delay A → Prop) :=
  ∀ (t1 t2 : Delay A), t1 ≈ t2 → p t1 → p t2.
```

This restriction corresponds to the notion that `Tau` is a *silent* step of computation. Note that ≈ does distinguish between `spin` and any term that contains a finite list of `Tau` with a base of `Ret`.

In order to satisfy the monotonicity requirement on specification monads, we work only with the sets of predicates that are monotonic. More explicitly,

```
Definition monotonic (w : (Delay A → Prop) → Prop ) := ∀(p p' : Delay A → Prop),
   (∀ t : Delay A, p t → p' t) → w p → w p'.
```

Put together, the full definition of `DelaySpec` is as follows:

```
Definition DelaySpec (A : Type) :=
  { w : (Delay A → Prop) → Prop | monotonic w ∧ ∀p, w p → resp_eutt p }.
```

To avoid including cumbersome projections, the fact that `DelaySpec` is a sigma type is elided in the rest of the paper. However, all specifications used (or generated by) code in this paper are monotonic.

4.2 Monad Structure

Below is the definition for `ret` in the `DelaySpec` monad.

```
Definition ret (a : A) := fun (p : Delay A → Prop) ⇒ p (Ret a).
```

Read this term as the set of predicates that include computations equivalent to `Ret a`. The definition of ilcbind is far more interesting.

```
Definition bind (w : DelaySpec A) (g : A → DelaySpec B) :=
  fun (p : Delay B → Prop) ⇒
    w (fun (t : Delay A) ⇒ (∃ (a : A), Ret a ≈ t ∧ g a p) ∨ (diverges t ∧ p spin)).
```

First note that since all we can do with `w : DelaySpec A` is ask if a predicate over `Delay A` is included, we need to construct a new predicate over `Delay B` that encodes some of the information from the original predicate. Constructing a new predicate brings into scope a new `Delay A` computation `t`. In the case of convergence, we can apply `g` to `a` and produce an element of `DelaySpec B`. We can then test whether the original predicate `p` is contained in `g a`. In the case of divergence, we utilize the fact that spin is the only divergent element of Delay B, and test whether p accepts spin. This can be thought of as a safe casting of t from Delay A to Delay B.

When given this monad structure, taking set equality as the equivalence relation, DelaySpec satisfies all of the monad laws. Verified proofs of the laws are provided in our Coq development. As is to be expected when working with termination sensitive predicates, these proofs rely on classical logic. In particular, we use the Law of The Excluded Middle to prove the element-wise disjunction of convergence and divergence.

```
Lemma div_or_ret_a : ∀(A : Type) (t : Delay A), diverges t ∨ ∃(a : A), t ≈ ret a.
```

We cannot prove such a lemma in intuitionistic logic. Any intuitionistic proof would be a program that takes a tree and determines whether it converges or diverges, which would be a Halting Problem decision procedure.

4.3 Specification Order

The order that gives the DelaySpec monad the full structure of a specification monad is as follows.

```
Definition order (w1 w2 : DelaySpec A) := ∀(p : Delay A → Prop), w2 p → w1 p.
```

Intuitively, w1 ≤ w2 if w1 is a superset of w2. The direction of the implication makes sense for the same reason the direction in the IDSpec order makes sense, as explained in Section 3.

To be a proper specification monad, the bind combinator needs to be monotonic with respect to the order. This can be formally expressed as follows.

```
Lemma monot_bind : ∀(A B : Type) (w1 w2 : W A) (f1 f2 : A → W B),
  w1 ≤ w2 → (∀ a, f1 a ≤ f2 a) → bind w1 f1 ≤ bind w2 f2.
```

The proof that this structure follows this law is straightforward given the monotonicity restriction on specifications.

4.4 Effect Observation

Finally, in order to connect our notions of computation and specification, we need an effect observation from computations to specifications. The effect observation takes a tree to the set of predicates that accept it, exactly as the IDSpec effect observation does.

```
Definition obs (t : Delay A) := fun (p : Delay A → Prop) ⇒ p t.
```

4.5 Lifting to More Effects

Following the work of Maillard, *et. al.* [2019], we can take this base specification monad and apply monad transformers to yield specification monads over more expressive computation types. The monad structure, order, and all effect observations can be lifted automatically to yield more expressive specification types. For instance, by applying the state transformation with state type S to DelaySpec, we obtain the type (Delay (S * A) → Prop) → (S → Prop). Note that given the computational monad stateT S Delay A = S → Delay (S * A), denoted StateDelay, the type Delay (S * A) → Prop is the type of post conditions and S → Prop is the type of preconditions. The automatically generated effect observation is exactly the notion of the weakest precondition of a postcondition, given a program.

```
Definition obs_state (m : S → (Delay (S * A) )) :=
  fun (p : Delay (S * A) → Prop) (st : S) ⇒ p (m st).
```

```
Definition iter_lift {A B : Type} (body : A → Delay (A + B)) : (A + B) → Delay (A + B) :=
  fun x ⇒ match x with
          | inl a ⇒ body a
          | inr b ⇒ ret (inr b) end.

Lemma loop_invar_sound : ∀(A B : Type) (body : A → Delay (A + B) ) (acc : A)
                         (p : Delay B → Prop)
                         (q : Delay (A + B) → Prop )
                         (establishment : q (body acc))
                         (preservation : ∀r, q r → q (bind r (iter_lift body)))
                         (postcondition : (∀ r, q (fmap inr r) → p r)),
      p (iter body acc) ∨ diverges (iter body acc).
```

Fig. 4. Delay Monad Iteration Loop Invariant Lemma

5 LOOP INVARIANTS

5.1 Inference Rule Statement

After generating verification conditions, we are still left with the task of proving them. Simple programs often require only the basic ITrees machinery to verify. However, non-trivial programs will often use the (coinductively defined) `iter` combinator, which is used to implement loops. To reason about such programs it is convenient to introduce loop invariants.

The `iter` combinator is the primary way of introducing potentially non-terminating computations built from ITrees. The term (`iter body init`) denotes a looping computation of type `Delay B`, where the `body : A → Delay (A + B)`, given a current “loop accumulator” of type `A`, produces either the next value of the accumulator, signaling that the loop should continue, or a result of type `B`, signaling that the loop should terminate. The initial loop accumulator is given by `init`. Looping constructs are often tricky to reason about, and `iter` is no exception. Monad iterators obey a useful equational theory. The most relevant of these laws to our work is the `iter_unfold` law, which captures the notion of loop unrolling, written below.

```
iter body init ≈ bind (body init) (case (iter body) id)
```

To facilitate reasoning about `iter`, we implement a loop-invariant proof combinator, embodied by the Lemma shown in Figure 4. It packages up a significant amount of machinery, and its proof is non-trivial; posing and verifying this lemma in Coq is one significant contribution of this work. The structure of the lemma is closely analogous to that of the classic Hoare-logic loop invariant for `while` statements. Intuitively `p` is the predicate we actually want to prove, and `q` is the loop invariant, which takes into account the intermediate accumulator values. The establishment clause `q (body a)` is the proof that the initial accumulator result `body a` satisfies our loop invariant. The preservation clause `∀ r, q r → q (bind r (iter_lift body) )` captures the notion that if you have an intermediate result `r` and apply another iteration of `body` to it, then the invariant is preserved. The `iter_lift` function, defined in Figure 4, bridges the gap between input type of `body` and the type of `r`. Finally, the postcondition clause shows that if the `body` returns a “halt” result `ret (inr b)`, then this is enough to conclude `p` would hold if you "cast" `t` to `Delay B`. Given all of these hypotheses, we can prove that `iter g a` either satisfies `p` or diverges.

5.2 Rooted Well-Foundedness

To prove this inference rule, we need more machinery to reason about `iter` terms. Our specifications can reason about termination and non-termination, so we need to be able to determine whether a tree converges or diverges. There are effectively two different sources of non-termination we need to worry about when dealing with `iter`. Consider two terms

```
Definition t1 := iter (fun n ⇒ ret (inl n)) 0
Definition t2 := iter (fun n ⇒ if n < 10 then ret (inl (n - 1) ) else spin) 0.
```

The tree `t1` diverges because we iterate the body infinitly often. The tree `t2` diverges because, after 10 iterations, the iteration body receives a value it diverges on. We call the former case *external divergence* and the latter case *internal divergence*. External divergence is uniquely troublesome. Used outside of this context, such loop bodies always converge, yet the iterated term still diverges.

With this in mind, an `iter` term evaluates in one of three ways: (1) It iterates a finite number of times and then returns a value. (2) It iterates a finite number of times and returns a divergent tree. (3) It iterates an infinite number of times and is a divergent tree. Furthermore, the hypotheses of our loop invariant inference rule allow us to reason about arbitrarily many iterations of `body` starting from `init` but require information about the intermediate values in order to build the proof. For a typical relational semantics for a programming language, an evaluation derivation would contain such information in an inductive datastructure. To reconstruct this information we interpret the `iter` body as a relation on `A` and leverage inductive and coinductive structures built on top of relations. These structures also provide a framework for detecting external divergence.

The concept of a well-founded relation is well established in both computer science and mathematics. A well-founded relation is a relation R such that there is no infinite sequence $x_0, x_1, ...$ such that for any i, $(x_i, x_{i+1}) \in R$. This allows us to proceed by induction on any chain of elements descending in R. This concept has been used to prove termination for loops. For our purposes, it is useful to generalize this concept to rooted well-foundedness. A relation r is well-founded from root a if there are no infinite length r chains starting from a. This predicate is implemented by the following code.

```
Inductive wf_from (r : A → A → Prop) (a : A) : Prop :=
  | base_wf : (∀ (a' : A), ~r a a') → wf_from r a
  | step_wf : (∀ (a' : A), r a a' → wf_from r a') → wf_from r a.
```

Working in a classical logic framework, this predicate is actually sufficient for our purposes. If the relation derived from the `iter` body is well-founded from its starting value, then it either converges or internally diverges. If the relation is not well-founded, then the existence of that evidence is logically equivalent to a proof of external divergence. However, we construct a coinductive predicate that is closer to our goal for ease of reasoning. Any relational tree that is not well-founded must contain an infinite path. This is formalized as follows.

```
CoInductive not_wf_from (r : A → A → Prop) (a : A) :Prop :=
  | step_nwf (a' : A) : r a a' → not_wf_from r a' → not_wf_from r a.
```

We prefer the `not_wf_from` predicate to the simple negation of `wf_from` because it is easier to prove the divergence of a tree with a coinductive predicate than with an inductive one. We prove that given any relation and starting value, one of these predicates must hold.

```
Lemma classic_wf : ∀(r : A → A → Prop ) (a : A), wf_from r a ∨ not_wf_from r a.
```

When proving the soundness of our loop invariant, we use the following relation.

```
Definition iter_rel {A B : Type} (g : A → Delay (A + B)) :=
  fun a1 a2 ⇒ g a1 ≈ Ret (inl a2).
```

The `iter_rel g` relation accepts a pair `a` and `a'` if running `iter g a` initially produces an `inl a'` result. The rooted well-foundedness, or lack thereof, of `iter_rel g` is closely relatedly to the convergence, or divergence, of `iter g a`.

5.3 Proof of the Loop Invariant Soundness

We now have everything we need to prove our loop invariant inference rule `loop_invar_sound`. Referring to the statement of the rule in Figure 4, we proceed as follows.

Proof. First, we prove divergence in the `not_wf_from (iter_rel g) a` case. It suffices to prove that `iter g a ≈ spin`. We proceed by coinduction with the coinductive hypothesis `∀ a, not_wf_from (iter_rel g) a → iter g a ≈ spin`. We can then infer that there is an `a'` such that `iter_rel g a a'` and `not_wf_from (iter_rel g) a'`. Combining these facts with the `iter_unfold` rule, we can conclude that `iter g a ≈ iter g a'`, soundly shift our goal to `iter g a' ≈ spin`, and conclude by applying our coinductive hypothesis.

In the remaining case, we effectively have an evaluation derivation in the proof of our `wf_from (iter_rel g) a` proof term. We can induct on this term. In the `base` case, we know that `a` has no sucessors with respect to the `rg` relation. This means that either `g a ≈ ret (inr b)` for some `b` or `g a ≈ spin`. Combined with the `iter_unfold` rule, this tells us that either `iter g a ≈ ret b` or `iter g a ≈ spin`. In the first case, the loop converges to `ret b` and we use the `establishment` and `postcondition` clauses to prove `p (ret b)`. In the second case, we already know `diverges spin`, so we can finish.

In the `step` case we get to assume either `p (iter g a')` or `diverges (iter g a')` for any `a'` where `iter_rel g a a'`. If there is no such `a'` the proof proceeds as it did in the previous step. Otherwise, we know that `g a ≈ ret (inl a')` and can, from that, infer that `iter g a ≈ iter g a'`. At this point, we can conclude the proof by applying the inductive hypothesis. □

5.4 Using Rooted Well-Foundedness

Beyond their usefulness in the above proof, `wf_from` and `not_wf_from` are useful for reasoning about specific programs. However, neither is particularly easy to directly use. Proving `not_wf_from` requires coinduction, an often frustrating task even for experienced computer scientists, and proving `wf_from` requires often non-obvious selection of the evidence to induct on. For this reason, we have provided simple reasoning principles to construct these proofs. Such a proof effectively takes an element and builds an infinite stream where each pair of elements is justified by the relation. To prove `not_wf_from r a`, it is sufficient to: (1) provide a predicate `inv` on `A` such that `inv` is preserved under `r`; (2) provide a function `f : A → A` that preserves `r` when given an element that satisfies `inv`; (3) prove that `a` satisfies `inv`.

```
Lemma intro_not_wf : ∀(r : A → A → Prop) (inv : A → Prop) (f : A → A) (a : A),
    inv a → (∀ a1 a2, inv a1 → r a1 a2 → inv a2 ) → (∀ a, inv a → r a (f a)) →
    not_wf_from r a.
```

For a concrete example, consider the less-than relation. To prove that this relation is not well-founded given an arbitrary natural number, all one needs to do is provide the successor function and prove that every number is less than its successor.

One might think that proving the `wf_from` predicate would be significantly easier because induction is often easier to handle than coinduction. However, the inductive structure of the `wf_from` proof can often have little connection to the structures involved in the relation, which may itself be coinductive. To avoid these difficulties, we allow proofs to be built by injecting the type into the natural numbers and to prove `wf_from r a`, it is sufficient to: (1) provide a predicate `inv` on `A` such that `inv` is preserved under `r`; (2) provide a function `f : A → nat` such that given any elements `a1`, `a2` that are related under `r` and satisfy `inv`, `f a1 > f a2`; (3) prove that `a` satisfies `inv`.

```
Lemma wf_intro_gt : ∀(r : A → A → Prop) (f : A → nat) (inv : A → Prop) (a : A),
    (∀ a1 a2, inv a1 → r a1 a2 → inv a2) →
    (∀ a1 a2, inv a1 → r a1 a2 → f a1 > f a2) →
```

```
Definition reassoc {A B : Type} (t : Delay (S * (A + B) ) ) : Delay ((S * A) + (S * B) ) :=
  bind t (fun '(s,ab) ⇒
          match ab with
          | inl a ⇒ ret (inl (s , a))
          | inr b ⇒ ret (inr (s , b))
          end).

Definition state_reassoc {A B : Type} (f : A → StateDelay (A + B) ) :
  (S * A) → Delay ((S * A) + (S * B)) :=
    fun '(s,a) ⇒ reassoc (f a s).

Lemma loop_invar_state: ∀(A B : Type) (g : A → StateDelay (A + B)) (a : A) (s : S)
            (p : Delay ( S * B) → Prop) (q : Delay ((S * A) + (S * B)) → Prop)
            (establishment : q (reassoc (g a s) ))
            (preservation : ∀t, q t → q (bind t (iter_lift ( state_reassoc g))))
            (postcondition : ∀t, q (fmap inr t) → p t),
      (p (iter g a s) ∨ diverges (iter g a s)) .
```

Fig. 5. StateDelay Monad Iteration Loop Invariant Lemma

```
      inv a → wf_from r a.
```

5.5 Stateful Loop Invariants

We now have a verified loop invariant inference rule for the `Delay` monad. In practice, the programs we work with often do not have semantics directly in the `Delay` monad. However, monads built on top of `Delay` have iteration structures based on that of `Delay`. Recall the definition of `StateDelay S A`, which is `S → Delay (S * A)`. Given `g : A → StateDelay (A + B)`, `a : A`, and `s : S`, `iter` over `StateDelay` keeps applying `g` to the elements `S * A` produced by the previous iteration. It achieves this possibly nonterminating behavior by using the coinductively defined `iter` of the `Delay` monad. Consequently, we can use our `iter` loop invariant inference rule for the `Delay` monad to create a verified loop invariant inference rule for the `StateDelay` monad. This rule is stated in Figure 5. The proof of this inference rule follows from unfolding the definitions of the `StateDelay` `iter` combinator until we can apply our original loop invariant inference rule.

6 HOARE LOGIC EXTENSION

Now we will turn our attention specifically to proving specifications on stateful, possibly diverging programs. We can construct the `StateDelay` and `StateDelaySpec` monads by applying the state transformation to our base `Delay` and `DelaySpec` monads. We also get the stateful weakest precondition observation, `StateDelaySpecObs` for free from our approach. We can define a function that takes a stateful computation and a stateful specification and produce a verification condition that the computation satisfies the specification. Recall the definition on `DijkstraVerify` from Section Figure 3.

```
 Definition verify_cond A := DijkstraVerify StateDelay StateDelaySpec StateDelayObs A.
```

6.1 Embedding Pre and Post Conditions

The types of specifications we have been directly working with are not how we typically think of specifications. For example, we often want to specify precondition that, when enforced on input, leads to output that satisfies a postcondition. As shown by Maillard *et. al.*, the backwards predicate transformer type is expressive enough to encode such predicates. Consider the following encoding function for state preconditions and postconditions.

```
Definition encode {A : Type} (pre : S → Prop) (post : Delay (S * A) → Prop) : StateSpec A :=
 fun s p ⇒ pre s ∧ (∀ r, post r → p r).
```

It may be unclear why this predicate requires the precodition to hold on any s. One might expect that the precondition holding on s would be a hypothesis in some implication. The design decision becomes clear once one looks at the proof goal constructed when trying to prove that the observation of a computation m is contained in this specification. Suppose we have a program m, a post condition post, and a precondition pre. The proposition that m satisfies this pair of conditions is
`verify_cond (encode (pre,post) ) m`
After some reduction, we receive the following goal.

```
Goal ∀(s : S) (p : Delay (S * A) → Prop ), pre s ∧ (∀ r, post r → p r) → p (m s).
```

With a minimal amount of manipulation, this goal reduces to being able to assume the precondition and then being required to prove the postcondition, exactly as desired. We also can write a direct definition of what it means for a program to satisfy a precondition and post condition, `∀ s, pre s → post (m s)`. This condition is satisfied exactly when
`verify_cond (encode (pre,post)) m` holds.

To inch even closer to actual practice, we often have a list of precondition postcondition pairs where satisfication of the specification means that for each precondition the input satisfies, we have a guarantee that the associated postcondition is satisfied. We can encode that as well.

```
Definition encode_list {A : Type} (ppl : list (PrePost A)) : StateSpec A :=
 fun s (p : Delay (S * A) → Prop ) ⇒
   List.Exists (fun pp : PrePost A ⇒ let (post,pre) := pp in pre s ∧ ∀r, post r → p r) ppl.
```

This breaks down to proving that each precondition entails that the postcondition holds on all respective program executions.

6.2 Generalized Correctness

Classic Hoare logic deals only with partial correctness properties, guarantees that a program satisfies a specification if it ever terminates. Later work built on Hoare logic dealt with total correctness properties, guarantees that a program satisfies a specification and that it terminates. Ideally, we want specifications to be able to reason more flexibly about convergence. Some programs, like operating systems, are intended to diverge. Other programs might be expected to diverge under certain error conditions. For instance, a programmer may want to verify that a simple numerical program loops forever when the precondition is violated in a specific way. Understanding specific error behavior can be useful. The type of our specifications is rich enough to express convergence and divergence as predicates, subsuming both partial and total correctness. Consider the example program introduced in the introduction.

```
Definition nat_sqrt : com :=
  i ::= 0;;;
  WHILE ~(i * i = n) DO i ::= i + 1 END.
```

We want to prove that if the initial value of n is a perfect square then the program halts in a state where i maps to the square root of n, and if the initial value of n is not a perfect square then the program diverges. We can formalize this specification with two pairs of preconditions and post conditions. First we need to provide formal definitions of these conditions.

```
Definition pre1 := fun s ⇒ is_square (get n s).
Definition post1 := fun (t : Delay (env *unit)) ⇒
                    ∃ s, t ≈ Ret (s,tt) ∧ get i s * get i s = get n s.

Definition pre2 := fun s ⇒ ~(is_square (get n s)).

Definition nat_sqrt_spec := encode_list [(post1, pre1); (diverges, pre2) ].
```

With these in place, we need to prove the following lemma.

```
Lemma nat_sqrt_sat_spec : verify_cond (interp_imp nat_sqrt) nat_sqrt_spec.
```

Proof. Our reasoning principles are over the state transformed `Delay` monad, not over IMP syntax, so we need to look directly at the denotation of this program. However, because interpretation is defined as the homomorphic extension of the given event handler, this does not give us a reasonable term. Consider a simpler example program `x ::= x + x`. This program will be denoted as the following term.

```
fun s ⇒  (x0 ← handle (Get x);; x1 ← handle (Get x);; Ret (put x (x0 s + x1 s) s), tt).
```

Notice that this term is flattened, hard to read, and does not make it clear that `x0` and `x1` are equivalent. For less trivial programs like our actual example, the monadic term is far less readable. This makes it difficult to directly reason about these programs even when they are very simple. Luckily we can prove that these denotations are equivalent to more readable terms and use Coq's setoid rewriting to replace the denotations during proofs. Nothing within an expression can ever mutate the state, so we can rewrite the denotation for any expression to a term without any usage of `bind`. For instance, the above term can be rewritten into the following, much simpler term.

```
fun s ⇒  Ret (put x (get x s + get x s) s, tt)
```

Given a `compute_bexp` that denote boolean expressions as functions from state to booleans, and the `inc_var` function that takes a variable, a state, and returns the state with the given variable incremented by 1, we can express the `nat_sqrt` program very succinctly as a computation.

```
(fun s ⇒ bind (Ret (set i 0 s, tt)) (iter (fun (_ :unit) (s : env) ⇒
                                          if (compute_bexp (~ i * i = n) s)
                                          then Ret (inc_var i s, inl tt)
                                          else Ret (s, inr tt)) )
```

Proving our specification comes down to proving three main goals: (1) The program diverges if `n` is not a perfect square; (2) The program converges if `n` is a perfect square; (3) The program is partially correct with respect to the first spec.

Recall from Section 5 that we have convenient lemmas for proving the well-foundedness or not-well-foundedness of relations. Also, note that the loop body of this program converges in all cases. This means that in each case we can prove the convergence or divergence of the program by proving the well-foundedness or not-well-foundedness of the relations induced by the loop bodies. First, consider the case where `n` is a perfect square. Let `m` be the square root of `n`. Let `inv` be the set of states where `i ≤ m` and `n` retains its initial value and let `f` take a state to the value `m-i` evaluated on that state. Since `i` starts at 0, the initial state is in this set and it is easy to see that `f` decreases in this set. This is sufficient to prove that this relation is well-founded, and that the program converges.

Now consider the case where `n` is not a perfect square. In this case `compute_bexp ~(i * i = n)` always evaluates to `false`. Therefore the function that takes a state `s` and returns `inc_var i s` respects the relation from any starting point. This is sufficient to show that the relation is not-well-founded, which in turn lets us conclude that the program diverges.

$$\frac{}{\{P\}\texttt{SKIP}\{P\}} \qquad \frac{\{P\}c_1\{Q\} \qquad \{Q\}c_2\{R\}}{\{P\}c_1;;;c_2\{Q\}} \qquad \frac{\{P \wedge b\}c_1\{Q\} \qquad \{P \wedge \neg b\}c_2\{Q\}}{\{P\}\texttt{IF}\ b\ \texttt{THEN}\ c_1\ \texttt{ELSE}\ c_2\ \texttt{FI}\{Q\}}$$

$$\frac{}{\{P[x \mapsto a]\}x{::=}a\{P\}} \qquad \frac{\{P \wedge b\}c\{P\}}{\{P\}\texttt{WHILE}\ b\ \texttt{DO}\ c\ \texttt{DONE}\{P \wedge \neg b\}}$$

Fig. 6. (Standard) Hoare-logic rules.

To prove the partial correctness property, we need to use the `iter` loop invariant. Recall that this invariant must account for the sum type in the body of `iter` term. As we can see in the denoted term, the body returns a `StateDelay (unit + unit)` computation. With this distinction, left returns are associated with the loop guard evaluating to true and running the body and right returns are associated with the loop guard evaluating to false. For these less formal statements, let `n0` refer to the initial value of `n` and let `n` refer to the current value. A valid loop invariant maps left states to the requirement that `i * i ≤ n ∧ n0 = n` and right states to `i * i = n ∧ n = n0`. Stated more formally,

```
Definition loop_invar_iter := (fun (t : Delay ((env * unit) + (env * unit)) ) ⇒ ∃s0,
    ((t ≈ ret (inl (s0,tt)) ∧ get i s0 * get i s0 ≤ n0 ) ∨
    (t ≈ ret (inr (s0,tt)) ∧ get i s0 * get i s0 = n0)) ∧ get n s0 = n0 )
```

Establishment, preservation, and postcondition for `loop_invar_iter` follow from straightforward applications of ITree reasoning principles, along with the denotation shortcut lemmas for IMP programs and expressions. □

6.3 Recovering Hoare Logic

Note how our proof about the example program dealt with partial correctness seperately. This shows that our approach still leaves room for partial correctness logics. Using our ITree IMP semantics, we can define Hoare Triples as follows:

```
Definition hoare_triple (P Q : env → Prop) (c : com) :=
  ∀ (s s' : env), P s → interp_imp c s ≈ Ret (s',tt) → Q s'.
```

In addition we can verify all of the standard Hoare rules shown in Figure 6. With the exception of the `WHILE` rule, all follow directly from the definitions. The `WHILE` rule can be proven by an application of our `loop_invar` reasoning principle. As always, proofs derived directly using these rules only reason about partial correctness. One, often convenient, way to prove total correctness results is to prove partial correctness results, identify the right decreasing metrics, and prove convergence. That would have been an alternative way to prove the first precondition, postcondition pair that holds on `nat_sqrt`.

The command sequencing and if rule both follow from equational reasoning principles from the ITree library. The weakening rule is simply equivalent to the transitivity of predicate implication. Both the assignment and while rules required some more significant development. Both require a few steps of interpretation of the ITree denotation of the program before they can be reasoned about.

It is worth mentioning that in order to conclude that after a while loop terminates, its loop guard evaluates to false, we need to include that information in our `iter` loop proof. With these rules, we can revisit our example. A valid loop invariant for that approach would be `(i * i ≤ n) ∧ (n0 = n)`. The rest of the Hoare Style proof is trivial.

```
Definition euttEv {E1 E2 R S} (REv : ∀(A B : Type), E1 A → E2 B → Prop)
         (RAns : ∀(A B : Type), E1 A → A → E2 B → B → Prop) (RR : R → S → Prop) :
   itree E1 R → itree E2 S → Prop := (* omitted *)

Lemma euttEv_ret : RR r s → euttEv REv RAns RR (Ret r) (Ret s).
Lemma euttEv_taulr : euttEv REv RAns RR t1 t2 → euttEv REv RAns RR (Tau t1) (Tau t2).
Lemma euttEv_taul : euttEv REv RAns RR t1 t2 → euttEv REv RAns RR (Tau t1) t2.
Lemma euttEv_taur : euttEv REv RAns RR t1 t2 → euttEv REv RAns RR t1 (Tau t2).
Lemma euttEv_vis : REv e1 e2 → (∀ a b, RAns e1 a e2 b → euttEv REv RAns RR (k1 a) (k2 b)) →
     euttEv REv RAns RR (Vis e1 k1) (Vis e2 k2).
```

Fig. 7. EuttEv Properties

7 INTERACTION TREE TRACES

As previously discussed, not all programming languages can be easily modelled by a sequence of monad transformers applied to the `Delay` monad. This motivates us to go beyond transformations of `DelaySpec` and develop a Dijkstra monad for `itree E` where `E` is any arbitrary event type family.

7.1 First Attempt at Specification Monad

A straightforward generalization of the `DelaySpec` monad does not work. Consider the following flawed monadic structure.

```
Definition ITreeSpec (A : Type) := (itree E A → Prop) → Prop.
Definition ret_itreespec {A : Type} (a : A) : ITreeSpec A := fun p ⇒ p (Ret a).

Definition bind_itreespec {A B : Type} (w : ITreeSpec A) (g : A → ITreeSpec B) :=
  fun p ⇒ w (fun t ⇒ (∃ a : A, (converges a t ∧ g a p)) ∨ (diverges t ∧ p (div_cast t) ) ).
```

This is the direct generalization of `DelaySpec`. The `ret` case contains identical code. In defining the `bind` for arbitrary event ITrees, we use the `div_cast` function on ITrees that are guaranteed to diverge. We also use the `converges` predicate to extract return values from trees. However, this definition fails to satisfy the monad laws. Consider the following predicate and specification.

```
Definition p_counter : itree E unit → Prop := fun t ⇒ t ≈ Vis e (fun _ ⇒ Ret tt).
Definition w_counter : ITreeSpec unit := fun p ⇒ p (Vis e (fun _ ⇒ Ret tt) ).
Lemma bind_ret_counter : w_counter p_counter ∧ ~(bind_itreespec w_counter (fun a ⇒
     ret_itreespec a) ) p_counter.
```

This is a counterexample to the second monad law. The `bind` function ends up obscuring information about the events and nondeterminism in ITrees, causing `bind w_counter (fun a ⇒ ret a)` to reject the predicate `p_counter`, while `w_counter` accepts it. Intuitively, the problem is that in designing both our `ret` and `bind` functions, we need to choose between distinguishing trees based on their return values and distinguishing them based on the `eutt` relation. Neither choice gives us the expressiveness we need to have a useful specification monad.

7.2 Interaction Traces

We visualize ITrees as trees that branch at every `Vis` node into a potentially infinite number of subtrees. To develop a Dijkstra Monad for ITrees with uninterpreted events that avoids the issues we just discussed, we first develop a notion of potentially infinite traces over ITrees. We refer to these traces as ITraces. Intuitively, ITraces are determinized ITrees, each of which represents a single path through the tree, recording the "answers" given by the environment interpreting each event. To give an alternative intuition, ITraces are ITrees that package with each event the environment's response to the event and a continuation specialized to that specific response. We implement ITraces as ITrees with the event type `EvAns`.

```
Variant EvAns (E : Type → Type) : Type → Type :=
  | evans : ∀(A : Type) (ev : E A) (ans : A), EvAns E unit
  | evempty : ∀(A : Type) (ev : E A) (Hempty : A → void), EvAns E void.
```

Note that the parameter type of an `EvAns` constructed instance is either `unit` or `void`, which means that the continuation is forced to be either empty or a determinized tail. Now ITraces can be defined as an instantiation of ITrees.

```
Definition itrace (E : Type → Type) (R : Type) := itree (EvAns E) R.
```

Implementing ITraces as an instance of ITrees gives us the ITrees monadic and equational theory for free. ITraces are a useful intermediate abstraction because they provide expressiveness between the `Delay` monad and arbitrary ITrees. They can encode events and therefore interactions with the environment; however, unlike ITrees, they are still deterministic—an ITrace can converge to at most one value. This makes case analysis on the convergence or divergence of an ITrace very useful.

7.3 ITrace Refinement

Now, we will develop new machinery in order to build a notion of an ITrace "refining" an ITree. We build this relation using the `euttEv` relation shown in Figure 7. Like `eutt` it is parameterized by a relation on return values `RR`. It extends `eutt` by adding two additional relations `REv` and `RAns`. As you can see from the provided lemmas, this relation works the same as `eutt` when it comes to handling `Ret` and `Tau`. It only differs in the case where both the ITrace and the ITree have a top level `Vis` constructor. To understand this case, consider two trees `Vis e1 k1`, `Vis e2 k2` where `e1` and `e2` come from different event type families `E1` and `E2`. The `REv` relations determines if the events `e1` and `e2` are allowed to be related. The `RAns` relations takes `e1` and `e2` and determines which values `a` and `b` need to be considered when passing arguments to the continuations `k1` and `k2` to coinductively check.

By using equality for `REv` and allowing all values to be passed to the continuations, we get the original `eutt` relation. We use `euttEv` to construct the `trace_refine` relation using `eq` for `RR` and using specially designed `REv` and `RAns` relations.

```
Variant event_refine {E : Type → Type} : ∀(A B : Type), EvAns E A → E B → Prop :=
  | rer {A : Type} (e : E A) (a : A) : event_refine A (evans e a) e
  | ree {A : Type} (e : E A) (Hempty : A → void) : event_refine A (evempty e Hempty) e
.

Variant event_refine_cont {E : Type → Type} : ∀(A B : Type), EvAns E A → A → E B → B → Prop
    :=
  | rar {A : Type} (e : E A) (a : A) : event_refine_cont (evans e a) tt e a.

Definition trace_refine {E R} := @euttEv (EvAns E) E R R event_refine event_refine_cont eq.

Notation "b ⊑ t" := (trace_refine b t) (at level 90).
```

For inhabited types, `event_refine` pairs events `e : E A` with event answers `evans e a` for any possible answer `a`. For uninhabited types, `event_refine` pairs events `e : E A` with evidence that `A` is empty. `event_refine_cont` enforces that the only continuations that can be paired with events `e` and `evans e a` is `tt` and `a`. For example, suppose we had two trees `Vis (evans (GetE x) 3) ktt` and `Vis (GetE x) ka`, and wanted to prove the `trace_refine` relation accepts them. `event_refine` allow `GetE x` to be paired with `evans (GetE x) 3`, and `event_refine_cont` would enforce that `trace_refine` would only be called on `ktt tt` and `ka 3`.

This notion of traces and trace refinement yields a new notion of ITree equivalence.

```
Definition trace_equiv {E R} (t1 t2 : itree E R) :=
  ∀ b, b ⊑ t1 ↔ b ⊑ t2.
```

This notion of trace equivalence is actually equivalent to the `eutt eq` relation.

```
Lemma trace_set_complete : ∀E R (t1 t2 : itree E R),
      trace_equiv t1 t2 ↔ t1 ≈ t2.
```

Proof. (Sketch) By straightforward, if rather painstaking, coinduction. □

Knowing that these relations are equivalent gives us some more confidence that ITraces capture all of the interesting semantics of the ITrees they refine.

Later in the paper, we will define an effect observation from `itree E` into a monad based on the ITrace type. In order to prove that effect observation is a monad morphism, we will need some machinery to reason about `bind` over ITraces. When specialized to the ITrace case, `bind` is similar to the stream append function. It passes by each event, looking for a `Ret r` constructor so it can pass `r` to the continuation. Thus all of the events that occur in the original ITrace come before any produced by the continuation.

Given a trace `b`, a trace continuation `g`, a tree `t`, and a tree continuation `f`, we have two main ways to prove that `bind b g` ⊑ `bind t f`.First, if `b` converges to a value `r`, it suffices to prove that `g r` ⊑ `f r`.

```
Lemma converge_bind_refine : ∀E R S (b : itrace E R) (t : itree E R) (g : R → itrace E S)
(f : R → itree E S) (r : R),
  converges r b → b ⊑ t → (g r ⊑ f r) → (bind b g) ⊑ (bind t f).
```

This corresponds to the fact that `bind b g` is essentially `b` followed by `g r` which is guaranteed to follow a path along `bind t f`. This follows by induction on the evidence `converges r b`.

Second, if `b` diverges, then that is enough to conclude `bind b g` ⊑ `bind t f`.

```
Lemma diverge_bind_refine : ∀E R S (b : itrace E R) (t : itree E R) (g : R → itrace E S)
(f : R → itree E S),
    diverges b → b ⊑ t → (bind b g) ⊑ (bind t f).
```

This follows from our knowledge about divergent traces.

We can also partition a trace `b` that refines `bind t f` in the following way.

```
Lemma bind_refine_inv : ∀E R S (b : itrace E S) (t : itree E R) (f : R → itree E S),
    b ⊑ bind t f → ∃b', ∃g', b ≈ (bind b' g') ∧ (b' ⊑ t).
```

The proof of this lemma is more technical than the others. First, we introduce a relation `branch_prefix` on ITraces. We then show that given any prefix `b'` of a trace `b`, there exists a continuation `g'` that can be bound to `b'` to produce `b`.

```
Lemma trace_prefix_bind : ∀E R S (b : itrace E S) (b' : itrace E R),
    trace_prefix b' b → ∃g', b ≈ (bind b' g').
```

Finally, we provide a construction `peel : itrace E R → itrace E S → itrace E R` that satisfies the following condition.

```
Lemma peel_lemma : ∀E R S (b : itrace E R) (t : itree E S) (f : S → itree E R),
    (b ⊑ bind t f) → trace_prefix (peel b t) b ∧ (peel b t ⊑ t).
```

Essentially the `peel` function takes a trace `b` and a tree `t`, and follows along `b` until `b` extends further than `t`. Put together, this allows us to prove the `bind_refine_inv` lemma.

8 ITREE TRACE SPECIFICATIONS

With the trace model for ITraces constructed and investigated, we create a specification monad based on the IO specification monad presented in Dijkstra Monads For All [Maillard et al. 2019]. First we introduce the `ev_list E` type for any type family `E : Type → Type`. This is the type of logs that track all of the emitted events along with the answers that the environment provides. We then introduce the following type for inductive trace specifications

```
Definition TraceSpecInd (A : Type) :=
  ((ev_list E) → A → Prop) → ev_list E → Prop.
```

Intuitively, it is the type of predicate transformers from postconditions, over final traces with return values, to preconditions, over initial partial traces. We can give it the following monadic structure

```
Definition ret_ts_ind {A : Type} (a : A) : TraceSpecInd A := fun p log ⇒ p log a.
Definition bind_ts_ind {A B : Type} (w : TraceSpecInd A) (g : A → TraceSpecInd B) :
     TraceSpecInd B :=
  fun p log ⇒ w (fun log' a ⇒ g a p log') log .
```

`ret a` relates predicates to initial logs where the predicate accepts results where no further events are added to the log and `a` is returned as a final value.

From this specification monad, we need to figure out how to generalize it from traces represented by finite lists to potentially divergent traces. Here it is useful to use ITraces as our model for traces. ITraces give us an equational theory, a monadic structure, and also capture the notion that an infinite trace has no return value. We define the append operation, ++, over event lists and ITraces using `bind`, and use it in later definitions. The type of coinductive trace specifications is as follows

```
Definition TraceSpec (A : Type) :=
  (itrace E A → Prop) → ev_list E → Prop.
```

Note that all we did is change the type of postconditions from `ev_list E → A → Prop` to `itrace E A → Prop`[5]. The type of initial traces is inductive because it is not meaningful to ask questions like, after this infinite stream of events terminates, we see this other stream of events. Given that infinite initial logs are not meaningful, the choice to represent them inductively allows us to turn several potentially more laborious coinductive proofs into inductive proofs. We can then give this type the following monadic structure

```
Definition ret_ts {A : Type} (a : A) : TraceSpec A := fun p log ⇒ p (log ++ Ret a).
Definition bind_ts {A B : Type} (w : TraceSpec A) (g : A → TraceSpec B) : TraceSpec B :=
  fun p log ⇒ w (fun b : itrace E A ⇒ (∃ log', ∃a, b ≈ (log' ++ Ret a) ∧ g a p log') ∨
                              (diverges b ∧ (p (div_cast b)))) log .
```

We also provide an effect observation mapping ITrees into an equivalent `TraceSpec` elements. We first use the trace refinement relation to map ITrees into sets of ITraces and accept predicates `p` and initial logs `log` such that `p` accepts `log` appended to each refining trace `b`.

```
Definition obs_ts {A : Type} (t : itree E A ) : TraceSpec A := fun p log ⇒ ∀b, b ⊑ t → p (log
     ++ b) .
```

The proof of the monad laws for this structure is straightforward and provided in the supplementary materials. The proof that `obs_ts` forms a monad morphism between ITrees and `TraceSpec` elements is more involved and has more interesting components. The proof also makes heavy use

[5]Technically we only consider specifications that are monotonic in a sense analogous to the one described in Section 4 over predicates that respect `eutt`. These details are elided for clarity.

of the bind related properties we proved about ITraces. Recall that being a monad morphism means two things.

```
obs (ret a) ≈ ret a      and      obs (bind w f) ≈ bind (obs w) (fun a ⇒ obs (f a))
```

Unfolding that definition for the computational monad of ITrees, the specification monad TraceSpec, and the effect observation obs_ts, the ret clause is easy to prove, we obtain the following goal for the bind clause.

```
Lemma monad_morph_unfold:
      ∀ (A B : Type) (m : itree E A) (f : A → itree E B) (log : ev_list E)
           (p : itrace E B → Prop),
        ((∀ b : itrace E B, b ⊑ (bind m f) → p (log ++ b)) ↔
         ∀ b : itrace E A,
          b ⊑ m →
          (∃ (a : A) (log' : ev_list E),
              log ++ b ≈ log' ++ Ret a ∧ (∀ b0 : itrace E B, b0 ⊑ f a → p (log' ++ b0))) ∨
          diverges (log ++ b) ∧ p ( bind (log ++ b) (fun _ : A ⇒ spin))).
```

First we consider the forward case. When introducing the trace b : itrace E A, we perform case analysis on whether or not it diverges.

```
Lemma monad_morph_forward_conv:
  ∀ A B (m : itree E A) (f : A → itree E B) (log : ev_list E)
    (p : itrace E B → Prop),
    (∀ b : itrace E B, b ⊑ bind m f → p (log ++ b )) →
    ∀ (b : itrace E A) (a : A) (log0 : ev_list E), b ⊑ m →
      (log0 ++ Ret a ≈ b) →
      (∃ (a : A) (log' : ev_list E),
          log ++ b ≈ log' ++ Ret a ∧ (∀ b0 : itrace E B, b0 ⊑ f a → p (log' ++ b0)) ∨
      diverges (log ++ b) ∧ p ( bind (log ++ b) (fun _ : A ⇒ spin) ) ).
```

Proof. Here, we consider when b consists of a list of events log0 and a return value a. Choose the left disjunct and introduce a as the return value and log ++ log0 as the event list. We provide a proof that log ++ log0 ++ Ret a ≈ (log ++ log0) ++ Ret a in the ITraces library. That leaves us with an arbitrary bf that refines f a where we need to prove p (log ++ log0 ++ bf). Since b ≈ log0 ++ Ret a, we know log0 ++ bf ≈ bind b (fun x ⇒ bf) and perform that substitution. Then we can apply our hypothesis ∀ b, b ⊑ (bind m f) → p (log ++ b), and be left to prove that bind b (fun x ⇒ bf) ⊑ bind m f. Since b ⊑ m and bf ⊑ f a, we can conclude by calling the converge_bind_refine lemma. □

Then we consider the case where b diverges.

```
Lemma monad_morph_forward_div: ∀A B (m : itree E A) (f : A → itree E B) (log : ev_list E)
                              (p : itrace E B → Prop),
    (∀ b : itrace E B, b ⊑ (bind m f) → p (log ++ b)) →
    ∀ b : itrace E A, b ⊑ m → diverges b →
      (∃ (a : A) (log' : ev_list E),
          log ++ b ≈ log' ++ Ret a ∧ (∀ b0 : itrace E B, b0 ⊑ f a → p (log' ++ b0))) ∨
      diverges (log ++ b) ∧ p ( bind (log ++ b) (fun _ : A ⇒ spin) ).
```

Proof. We choose the right disjunct and prove that p accepts the casting of log ++ b to itrace E B. Using the associativity law of bind, we can apply our hypothesis ∀ b, b ⊑ (bind m f) → p (log ++ b) and be left with the goal that bind b (fun x ⇒ spin) ⊑ bind m f. Since diverges b and b ⊑ m, we can apply diverge_bind_refine to complete this case.

Now, we need to prove the backwards case of our double implication goal. One of our current hypotheses is b ⊑ bind m f. We can apply the bind_refine_inv to decompose b into b' and g'.

Then we can apply the left side of the double implication, available to us as a hypothesis, using the evidence that `b' ⊑ m`. We can now do case analysis on whether this bind predicate takes the convergent or divergent disjunct. In the convergent disjunct case, in addition to assuming that `log ++ b' ≈ log' ++ Ret a`, we can derive that `∃ log'', b' ≈ log'' ++ Ret a`. This sets up the following useful lemma. □

```
Lemma monad_morph_backwards_conv:
        ∀ A B (m : itree E A) (f : A → itree E B) (log log' log'': ev_list E)
              (p : itrace E B → Prop )
              (b : itrace E B) (b' : itrace E A) (g' : A → itrace E B) (a : A),
          (bind b' g' ≈ b) → (b' ⊑ m) →
          (log ++ b' ≈ log' ++ Ret a) →
          (∀ b : itrace E B, b ⊑ f a → p (log' ++ b)) →
          (log'' ++ Ret a ≈ b') →
          ∀ b : itrace E B, b ⊑ (bind m f) → p (log ++ b).
```

Proof. By substituting `log'' ++ Ret a` for `b'` in our assumption `bind b' g' ≈ b`, and doing some manipulation using the monad laws, we can derive that `log'' ++ g a ≈ b`. We can also derive that `log' ≈ log ++ log''` from simple substitutions. This allows us to restate our goal as `p (log' ++ g a)`. From here we can apply our assumption that `∀ b, b ⊑ f a → p (log' ++ b)` and be left to prove `g a ⊑ f a`. Because `b'` converges to `a`, `b' ⊑ m` and `bind b' g' ⊑ bind m f`, we can prove this goal and wrap up this case. □

We are now left with the final case, corresponding to `b'` diverging.

```
Lemma monad_morph_backwards_div :
    ∀ (A B : Type) (log : ev_list E) (p : itrace E B → Prop) (b : itrace E B)
      (b' : itree (EvAns E) A) (g' : A → itree (EvAns E) B),
      (bind b' g' ≈ b) →
      diverges (log ++ b') →
      p (bind (log ++ b') (fun _ : A ⇒ spin)) →
      p (log ++ b).
```

Proof. First we substitute `bind b' g'` in for `b` in the goal. Using the monad laws, we can further change the goal to `p (bind (log ++ b') g')`. Since `log ++ b'` diverges, the actual continuation used in the bind doesn't matter, and `bind (log ++ b') g' ≈ bind (log ++ b') (fun x ⇒ spin)`. This allows us to rewrite our goal into an assumption, and conclude the proof. □

9 TRACE SPECIFICATION EXAMPLES

This section presents two examples of using trace specifications for programs with non-trivial interactions with their environments.

9.1 Example with IO

We first demonstrate how to use the `TraceSpec` monad on a simple, yet illustrative example. Recall the example presented in the introduction, which we here render directly as an ITree.

```
Definition queryUntilFalse : itree NonDet unit :=
  iter (fun _ ⇒ bind (trigger Decide)
                (fun b : bool ⇒ if b then Ret (inl tt) else Ret (inr tt) )) tt.
```

This represents a program with a while loop that keeps making a nondeterministic choice to either run the loop again, or terminate the loop and end the program. It is natural to view this program as a set of possible traces encoded by ITraces. If the program diverges, then each nondeterministic choice must evaluate to true. If the program converges, all except the final choice must be true. With a few relatively simple technical additions, we can encode those intuitions as formal specifications and prove that the program satisfies them.

9.2 Predicates over ITraces

ITraces have very similar structures to streams. This means that many predicates that are useful to define over streams, which are in turn often adaptations of predicates defined over lists, are useful to define over ITraces. Consider the following type of a `Forall` predicate.

```
Definition traceForall E R (PE : ∀A, E A → Prop) (PR : R → Prop) : itrace E R → Prop :=
  (* omitted *)
```

`traceForall E R PE PR b` if all of the events in `b` satisfy `PE`, and if `b` returns a value `r` then `r` satisfies `PR`. This differs from a stream `Forall` predicate because it enforces properties on elements of `E A` for various `A`, rather than enforcing a property of elements of a fixed `A`. Additionally, it enforces some property on any value `b` returns. `traceForall` is defined coinductively. This means that infinite traces whose elements all satisfy `PE` are accepted. The `traceForall` predicate allows us to define what we require to be true of any divergent traces, namely that all nondeterministic choices must evaluate to `true`.

To encode the specification for the convergent case, we need a predicate that takes one property to enforce on all but the last event, one property to enforce on the last event, and another property to enforce on the final return value.

```
Definition front_and_last E R (PEF : ∀A, E A → Prop) (PEL : ∀A, E A → Prop) (PR : R → Prop) :
     itrace E R → Prop := (* omitted *)
```

The `front_and_last` predicate is defined inductively, and can only accept traces that converge. For our example, the front predicate, `PEF`, ensures that all choices except the final one must evaluate to `true` and the last predicate, `PEL` ensures that the final choice must evaluate to `false`. The returns predicate, `PR`, always holds.

9.3 Encoding Simple Specifications

Much like the `Delay` case, the specifications that we want to write over arbitrary event ITrees are rarely in the form of backwards predicate transformers. In order to write useful specifications, we want encodings from standard specification styles. Because the `TraceSpec` monad is so similar to the state transformed `DelaySpec` monad, the encoding functions can be reused.

```
Definition encode_ts {A : Type} (pre : ev_list E → Prop) (post : itrace E A → Prop)
  : TraceSpec A :=
  fun p log ⇒ pre log ∧ ∀b, post (log ++ b) → p (log ++ b).
```

With these tools in place, we can define the specification of our example. The specification is a precondition, postcondition specification. The properties we want to verify about the traces only actually hold on the whole trace when given specific initial logs. For this specification, we choose to enforce that the initial log is empty. Then we can encode the rest of the specification in the postcondition.

```
Definition queryUntilFalse_pre : ev_list NonDet → Prop := fun log ⇒ log = nil.
Definition fal_queryUntilFalse : itrace NonDet unit → Prop :=
  front_and_last (is_bool true) (is_bool false) (fun _ ⇒ True).
Definition queryUntilFalse_post : itrace NonDet unit → Prop :=
  fun b ⇒ (must_diverge b → traceForall (is_bool true) (fun _ ⇒ True) b) ∧
       (can_converge tt b → fal_queryUntilFalse b).
Definition queryUntilFalse_spec : TraceSpec NonDet unit :=
  encode NonDet queryUntilFalse_post queryUntilFalse_pre
```

The proof that this program satisfies its specification is straightforward. For the divergent case, it proceeds by simple coinduction. For the convergent case, it proceeds by induction on the evidence of convergence.

9.4 Example with IO and State

Just like in the DelaySpec case, we can automatically create a specification monad and an effect observation for the state transformation of ITrees with some uninterpreted event type. We use this structure to write a specification for a program with IO and state events.

Consider a program that begins by prompting the user to provide a natural number n, and then enters an infinite loop to print all of the multiples of n starting with 0. Let us assume we have an IMP-like programming language with added Input and Output constructs (on the left below). Let us also assume we have corresponding events to represent this program as an ITree (on the right). We omit the definition of these events types.

```
X := Input;;;
WHILE true DO
  Output Y;;;
  Y := X + Y
DONE
```

```
x  ← Input;; Store X x;;
iter (fun _ ⇒
  y  ← Load Y;; Output y;;
  x  ← Load X;; Store Y (x + y) )
tt
```

We can once again develop a notion of pre- and postconditions for this monad, and encode the pairs as elements of the specification monad. Both the pre- and postcondition types and the encoding function are nearly identical to the ones introduced earlier. The post condition we want to prove that this program satisfies is that the final trace gets an Input event that evaluates to some number n, and then has a stream of Output events that print the multiples of n in order. It is straightforward to define this trace coinductively.

```
CoFixpoint mults_of_n_from_m (n m : nat) : itrace IO unit :=
  Vis (evans (Output m) tt) (fun _ ⇒ mults_of_n_from_m n (n + m) ).
Definition mults_of_n (n : nat) : itrace IO unit := mults_of_n_from_m n 0.
```

From there it is straightforward to define the pre- and postconditions.

```
Definition pre_mult := fun log s ⇒ log = nil ∧ get Y s = 0.
Definition post_mult := fun tr ⇒ ∃n k, tr ≈ Vis (evans Input n) k ∧ k tt ≈ mults_of_n n.
```

With all of this in place, we automatically generate a verification condition and prove the specification holds using coinduction.

10 RELATED WORK

Work on creating logics to verify program specifications dates back to the 1960's. Foundational works like [Hoare 1969], [Floyd 1967] and [Dijkstra 1975] provided interpretations of programs that map postconditions to preconditions. These were originally external proof techniques for pen and paper proofs about the behavior of algorithms.

The work we build on most directly is that found in the Dijkstra Monad literature. This line of research has its roots in Hoare Type Theory [Nanevski et al. 2006], which presented a dependently typed functional programming language with mutable state and a novel Hoare type. A Hoare Type consists of some base type A, a precondition P on the state, and a postcondition Q on the state; it is inhabited by a computation producing an A that changes the state in a way that satisfies the postcondition given the precondition. This formulation is equivalent to specifying stateful computations using the state transform of the DelaySpec monad as the specification monad. Because Hoare Type Theory works with partial correctness, it is less expressive than how one could specify programs in our framework.

As we discussed earlier in the paper the Dijkstra Monad [Swamy et al. 2013], [Maillard et al. 2019] framework extends the ideas of Hoare Type Theory, adding support for algebraic effects like exceptions and IO, as well as providing a general framework for adding a new effect and a specification monad to handle it. This is used as an underlying technology for F*'s verification of effectful programs with respect to specifications that can describe their effects and not just their return values [Swamy et al. 2011]. In contrast to our `DelaySpec` and `TraceSpec` monads, previous Dijkstra Monads work has not directly addressed non-termination. We extend this work by adding the ability to reason fully about divergence in specifications, while retaining the ability to reason about interactions with the environment.

Our work uses ITrees [Xia et al. 2020] as its semantic model. Previous ITrees work focuses on the specific problems of program equivalences and the validity of program transformations, while this work focuses on the problem of general specifications. Using ITrees for semantics has the advantage of being able to leverage previous work on program equivalences while retaining extractable, and thus testable, reference executable implementations.

There is also a vast body of prior work on Coq-based Proof Frameworks for program correctness. Systems like YNot [Malecha et al. 2011], based on Hoare Type Theory, Iris [Jung et al. 2016], VST [Appel 2014], and FCSL [Sergey et al. 2015], all based on concurrent separation logic, and CertiKOS [Gu et al. 2016] [Gu et al. 2019], which uses certified abstraction layers, have had major success in the field of large scale program verification. Those models typically rely on small-step, relationally-specified operational semantics, and are especially useful for reasoning about concurrent programs—a domain that is still being explored for ITrees. To date most of these frameworks have emphasized partial correctness properties and many have limited facilities for reasoning about I/O, both of which are notable differences from our approach. This paper is a step towards creating a framework for large-scale program verification based on the Interaction Trees semantics.

11 CONCLUSION

This paper introduces two new base specification monads, `DelaySpec` and `TraceSpec`, and effect observations from `Delay` computations and arbitrary event ITrees, respectively. The construction of the `TraceSpec` monad required the development of a trace semantics for ITrees. The paper also investigates proving properties encoded in this framework, including verifying a loop invariant inference rule for the `Delay` monad's `iter` combinator.

In future work, we plan to develop reasoning principles for the `iter` combinator for arbitrary event ITrees. Because `TraceSpec` elements inspect entire traces instead of just the end result, a simple generalization of our current loop invariant principles will very often be insufficient. We also plan to investigate creating `iter` functions over specification monads. This function would allow us to write interpreters from ITrees directly into specification monads, allowing more abstract definitions of events that could be satisfied by classes of interpreters.

ACKNOWLEDGMENTS

This work was funded by the National Science Foundation's Expedition in Computing *The Science of Deep Specification* under the award 1521539 (Weirich, Zdancewic, Pierce) with additional support by ONR grant *REVOLVER* award N00014-17-1-2930. We are grateful to all the members of Penn's PL Club and the DeepSpec project for their collaboration and feedback, and we greatly appreciate the the reviewers' comments and suggestions.

REFERENCES

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

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

Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. 2017. Automated resource analysis with Coq proof objects. In *International Conference on Computer Aided Verification.* Springer, 64–85.

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457. https://doi.org/10.1145/360933.360975

Robert W. Floyd. 1967. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics* 19 (1967), 19–32. http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf

Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. 2019. Building Certified Concurrent OS Kernels. *Commun. ACM* 62, 10 (Sept. 2019), 89–99. https://doi.org/10.1145/3356903

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

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. *SIGPLAN Not.* 48, 1 (Jan. 2013), 193–206. https://doi.org/10.1145/2480359.2429093

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, LArs Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 256–269. https://doi.org/10.1145/2951913.2951943

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

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

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

Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. 2011. Trace-based Verification of Imperative Programs with I/O. *J. Symb. Comput.* 46, 2 (Feb. 2011), 95–118. https://doi.org/10.1016/j.jsc.2010.08.004

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

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2006. Polymorphism and Separation in Hoare Type Theory. *SIGPLAN Not.* 41, 9 (Sept. 2006), 62–73. https://doi.org/10.1145/1160074.1159812

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

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.

Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Beguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *IEEE Symposium on Security and Privacy.* IEEE. https://www.microsoft.com/en-us/research/publication/evercrypt-a-fast-verified-cross-platform-cryptographic-provider/

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-Grained Concurrent Programs. *SIGPLAN Not.* 50, 6 (June 2015), 77–87. https://doi.org/10.1145/2813885.2737964

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011.* 266–278. https://doi.org/10.1145/2034773.2034811

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 387–398. https://doi.org/10.1145/2491956.2491978

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

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