Abstract

This paper presents a specification framework for monadic, recursive, interactive programs that supports auto-active verification, an approach that combines user-provided guidance with automatic verification techniques. This verification tool is designed to have the flexibility of a manual approach to verification along with the usability benefits of automatic approaches. We accomplish this by augmenting Interaction Trees, a Coq datastructure for representing effectful computations, with logical quantifier events. We show that this yields a language of specifications that are easy to understand, automatable, and are powerful enough to handle properties that involve non-termination. Our framework is implemented as a library in Coq. We demonstrate the effectiveness of this framework by verifying real, low-level code.

2012 ACM Subject Classification Theory of computation → Denotational semantics; Theory of computation → Programming logic; Theory of computation → Separation logic

Keywords and phrases coinduction, specification, verification, monads

Digital Object Identifier 10.4230/LIPIcs...

Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations that Supports Auto-active Verification

Lucas Silver
University of Pennsylvania, USA

Eddy Westbrook
Galois, Inc.

Matthew Yacavone
Galois, Inc.

Ryan Scott
Galois, Inc.

1 Introduction

Formal verification is starting to see adoption in industry as a tool for ensuring the security and correctness of software. For instance, the formally verified seL4 microkernel [13] has established a foundation that is seeing investment from a wide variety of industrial partners. Block-chain companies are using formal verification to ensure the security of cryptocurrency [15]. Amazon has even incorporated formal verification into the CI/CD process of their s2n cryptographic library [7].

Unfortunately, formal verification still remains expensive, not just in terms of time and effort but also in terms of the expertise required to formally verify a system. A number of powerful frameworks have been developed for manual formal verification, including Iris [12], VST [2], and FCSL [24]. These frameworks can specify a wide array of behaviors on a wide array of languages, but they require an expert to be used effectively. Other powerful frameworks have been developed for automatic verification, including approaches such as bounded model-checking [4] and property-directed reachability [5]. While these approaches can be operated by non-experts, they are limited in their expressiveness, leaving important properties unverified.

It is particularly difficult to reason about low-level code that contains complicated manipulations of pointer structures on the heap, as is common in languages like C, C++, and LLVM. Recently, researchers have tackled this problem using the observation that programs that are well-typed in a memory-safe, Rust-like type system are basically functional programs [3, 9, 10, 17, 18]. That is, there exists a program in a functional language whose behavior is equivalent to the original, heap-manipulating program. We call this functional program a *functional specification*. While many projects rely only implicitly on the functional specification, some, like the Heapster project [9], reify functional specifications into concrete code. Engineers can then verify properties about the derived functional code, and ensure those properties hold on the original program.

The Heapster tool consists of two components: a memory-safe type system for LLVM code, and a translation tool that produces an equivalent functional program from any well-typed LLVM program. Heapster uses these components to break verification of heap manipulating programs into two phases: a memory-safe type-checking phase that generates a monadic, recursive, interactive program that is equivalent to the original program; and a behavior-verification phase that ensures that the generated program has the correct behavior. Previous work has left open major questions about the behavior verification phase, namely, what should the language of specifications be and how do we actually prove that the programs satisfy the specifications.

This work answers these questions by developing a logic well-suited to reasoning about the programs output by Heapster, as well as tools to work with these logical formulae. Taken together, the Heapster tool and this work form a two-step pipeline for verifying low-level, heap manipulating programs. Heapster transforms low-level, heap manipulating programs into equivalent functional programs. The techniques in this paper enable proof engineers to write and prove specifications over the resulting functional programs.

In this work, we present *interaction tree specifications*, or ITree specifications. ITree specifications are an *auto-active verification framework* for *monadic, recursive, interactive programs* based on *interaction trees* [29], or ITrees. Auto-active verification is a verification technique that merges user input and automated reasoning to leverage the benefits of each. Monadic, recursive, interactive programs have the ability to diverge, can interact with their environment, but otherwise act as pure functional programs. Interactions with the environment can include making a system call, sending a message from a server, and throwing an error. ITrees are a model for monadic, recursive, interactive programs formalized in Coq. ITree specifications are designed to be able to write and verify specifications about the output programs of the Heapster translation tool, which are written in terms of ITrees.

The main body of work that takes on the task of verifying monadic programs is the Dijkstra monad literature [1, 16, 27, 28]. However, most of the Dijkstra monad literature cannot handle the kinds of termination sensitive specifications that we need. These papers either assume a strongly normalizing language, or handle only partial specifications. The exception to this is the work of Silver and Zdancewic [25]. However, while that work does have a rich enough specification language for our goals, it has two significant shortcomings. First, the work provides no reasoning principles for arbitrary recursive specifications. Second, the work does not attempt to automate the verification of these specifications. Our work accomplishes both of these goals.

This work is based on the idea of augmenting ITrees with operations for logical quantifiers. We show that this idea leads to a language of specifications that is:

- easy to read, because the specifications are simply programs annotated with logical quantifiers,
- capable of encoding recursive specifications, because the underlying computational language has a powerful recursion operator, and
- amenable to auto-active verification, because specifications are syntactic constructs enabling syntax-directed inference rules.

ITrees represent computations as potentially infinite trees whose nodes are labelled with *events*. Events are syntactic representations of computational effects, like raising an error, or sending data from a server. ITrees can be used to represent the semantics of recursive, monadic, interactive programs. ITree specifications are ITrees enriched with events for logical quantifiers. This language of specifications has the capability to express purely executable computations, fully abstract specifications, and combinations of both. For example, consider the following executable specification `server_impl` for a simple server program that sorts lists which are sent to it:

```
Definition server_impl : unit → itree_spec E void :=
  rec_fix_spec (fun rec _ ⇒
                  l  ← trigger rcvE;;
                  ls  ← sort l;;
                  trigger (sendE ls);;
                  rec tt
               ).
```

```
Class EncodingType (E:Type) : Type :=
  response_type : E → Type.
```

Figure 1 EncodingType typeclass definition

This specification is defined with `rec_fix_spec`, a recursion operator (defined in Section 4) where applications of the `rec` argument correspond to recursive calls. The body of the recursive function first calls `trigger rcvE`, which triggers the use of the receive event `rcvE`, causing the program to wait to receive data. The list `l` that is received is then passed to the `sort` function, defined in Section 6, which is a recursive implementation of the merge sort algorithm. Finally, the sorted list returned by `sort` is sent as a response with `trigger (sendE ls)`, and the server program loops back to the beginning by calling `rec`.

Now, consider the following specification of the behavior of our server using a combination of executable and abstract features:

```
Definition server_spec : unit → itree_spec E void :=
  rec_fix_spec (fun rec _ ⇒
                 l  ← trigger rcvE;;
                 ls  ← exists_spec (list nat);;
                 assert_spec (Permutation l ls);;
                 assert_spec (sorted ls);;
                 trigger (sendE ls);;
                 rec tt).
```

This function acts mostly like `server_impl` but, instead of computing a sorted list, it uses the existential quantification operation `exists_spec` to introduce the list value `ls`, which it then asserts is a sorted permutation of the initial list. By leaving this part of the specification abstract, it allows the user to express that it is unimportant how the list is sorted, as long as the response is a sorted permutation of the input list. The send and receive events, however, are left concrete, allowing the user to specify what monadic events should be triggered in what order. This specification implicitly defines a liveness property of the server, it will reject any program that fails to eventually perform the next send or receive. By using a single language for programs and specifications, our approach provides a natural way for users to control how concrete or abstract the various portions of their specifications are. Our approach then provides auto-active tools for proving that programs refine these specifications.

Necessary background explaining ITrees and Heapster is given in Section 2 and Section 3. The contributions of this paper are as follows:

- ITree specifications, a data structure for representing specifications over monadic, recursive, interactive programs, presented in Section 4
- a specification refinement relation over ITree specifications, along with collection of verified, syntax-directed proof rules for refinement also presented in Section 4,
- tools for encoding and proving refinements involving total correctness specifications in ITree specifications presented in Section 5,
- an auto-active verification technique briefly discussed in Section 6
- an evaluation of the presented techniques in the form of verifying a collection of realistic C functions using ITree specifications and Heapster presented in Section 6.

2 Background

ITrees are a formalization for denotational semantics implemented as a coinductive variant of the free monad in Coq. ITrees represent programs as potentially infinite trees. The nodes of these trees are labelled with *events*. Events can, depending on the context, either represent algebraic effects or recursive function calls. The ITree type is parameterized by a return type `R` and a type family `E`, where `E` has an instance of the `EncodingType` type class defined in Figure 1. The `EncodingType` type class consists of function, named `response_type`, from `E` to `Type`. A value of type `itree E R` is a potentially infinite tree whose internal nodes are each labelled with an *event* `e` of type `E`, with one branch for each element of the `response_type e` whose leaves are labelled with an element of type `R`. Such a tree represents an effectful computation, where the leaves represent termination of the computation with a return value in `R` while the nodes represent uses of monadic effects. The event `e` of type `E` that labels a node represents a monadic effect that returns a value of type `response_type e`, and the children of that node represent the possible continuations of that computation depending on the return value of the effect. This is formalized in the following Coq code[1].

```
CoInductive itree (E : Type) `{EncodingType E} (R : Type) :=
  | Ret (r : R)
  | Tau (t : itree E R)
  | Vis (e : E) (k : response_type e → itree E R).
```

The ITree datatype has three constructors. The `Ret` constructor represents a pure computation that simply returns a value. The `Ret` constructor forms the leaves of an ITree. The `Tau` constructor represents one step of silent internal computation followed by another ITree. Finally, the `Vis` constructor contains an event `e` along with a continuation function `k` which defines all the branches of this `Vis` node.

Because ITrees are defined coinductively, we can construct ITrees with infinitely long branches. Such ITrees represent divergent computations. For example, the following code describes an ITree that consists of an infinite stream of `Tau` constructors with no events.

```
CoFixpoint spin : itree E R := Tau spin.
```

In practice, ITrees often end up using an event type family `E` that is a composition of several smaller type families combined in a large sum. This can easily clutter and complicate the notation. To avoid this burden, the ITrees library introduces the `ReSum` typeclass defined in Figure 2. An instance of `ReSum E1 E2`, written `E1 -< E2`, contains two functions: the `resum` function that injects an element of `E1` into `E2`, and the `resum_ret` function that maps elements from the response type of `resum e` to the response type of `e`. It can be thought of as a kind of subevent typeclass. The `ReSum` typeclass allows for the definition of the `trigger` function in Figure 2. The `trigger` function takes an event `e : E1` and injects it into `itree E (response_type e)` by injecting `e` into `E2`, placing that in a `Vis` node, and applying the `resum_ret` function to the response.

2.1 Equivalence up to Tau

One of the major advantages of the ITrees library is its rich equational theory. The primary notion of equivalence used for ITrees is called `eutt` or *equivalence up to tau*. Xia et al. [29] defines `eutt` as a bisimulation relation that quotients out finite differences in the number of `Tau` constructors. We use this relation because `Tau` constructors are supposed to indicate *silent* steps of computation. Ignoring finite numbers of `Tau` constructors lets us equate two ITrees that vary only in the number of silent computation steps.

[1] In the actual formalization, we use a negative coinductive types presentation of this data structure.

```
Class ReSum (E1 : Type) (E2 : Type) `{EncodingType E1} `{EncodingType E2} :=
{
  resum : E1 → E2;
  resum_ret : forall {e : E1}, response_type (resum e) → response_type e;
}.

Notation "E1 -< E2" := (ReSum E1 E2) (at level 10).

Definition trigger {E1 E2} `{EncodingType E1} `{EncodingType E2} `{E1 -< E2} :forall (e1
     : E1), (itree E2 (response_type e1)) :=
  fun e ⇒ Vis (resum e) (fun x ⇒ Ret (resum_ret x)).
```

Figure 2 ReSum Definition

```
Example spin ≈ spin.                    Example Tau (Ret 0) ≈ Ret 0.

Example ∼(spin ≈ Ret 0).
```

Figure 3 `eutt` Examples

The `eutt` relation is parameterized by a relation `RR` over return values. If the relation `RR` is *heterogeneous*, relating values over distinct types `R1` and `R2`, then `eutt RR` is also a heterogeneous relation over `itree E R1` and `itree E R2`. Intuitively, if `eutt RR t1 t2`, then the `Vis` nodes of `t1` precisely match those of `t2`, and if equivalent paths in `t1` and `t2` lead to the leaves `Ret r1` and `Ret r2` then the values `r1` and `r2` are related by `RR`. Often, we are interested in `eutt eq` and denote this relation with the symbol ≈.

The `eutt` relation is implemented in Coq using both *inductive* and *coinductive* techniques. Observe the following definition of `eutt`:

```
Inductive euttF (RR : R1 → R2 → Prop) (sim : itree E R1 → itree E R2 → Prop) :
    itree E R1 → itree E R2 → Prop :=
  | eutt_Ret (r1 : R1) (r2 : R2) : euttF RR sim (Ret r1) (Ret r2)
  | eutt_Tau (t1 : itree E R1) (t2 : itree E R2) :
    sim t1 t2 → euttF RR sim (Tau t1) (Tau t2)
  | eutt_Vis (e : E) (k1 : response_type e → itree E R1)
    (k2 : response_type e → itree E R2) :
    (forall a, sim (k1 a) (k2 a)) → euttF RR sim (Vis e k1) (Vis e k2)
  | eutt_TauL (t1 : itree E R1) (t2 : itree E R2) :
    euttF RR sim t1 t2 → euttF RR sim (Tau t1) t2
  | eutt_TauR (t1 : itree E R1) (t2 : itree E R2) :
    euttF RR sim t1 t2 → euttF RR sim t1 (Tau t2).
```

```
Definition eutt (RR : R1 → R2 → Prop) := gfp (euttF RR).
```

The `euttF` relation is an inductively defined relation, defined in terms of the `sim` argument. The `eutt` relation is then defined as the greatest fixpoint of `euttF`. In this paper, all greatest fixpoints are defined using the `paco` library[11] for coinductive proofs. Calls to the `sim` argument in the definition of `euttF` correspond to coinductive calls of `eutt`. Recursive calls to `euttF` correspond to inductive calls of `eutt`. This method of defining `eutt` allows the coinductive constructors to be called infinitely often in sequence, while only a finite number of calls to inductive constructors can be called without an intervening call to a coinductive constructor. Specifically, only finitely many `eutt_TauL` and `eutt_TauR` steps, that remove a `Tau` from only one side, are allowed before one of the remaining rules must be used to relate the same constructor on both sides.

This definition allows us to achieve our goal of ignoring any finite difference in numbers of `Tau` constructors. In particular the equations and inequalities presented in Figure 3 hold.

ITrees form a monad. Monads are type families with a `ret` combinator that denotes a pure value, and a `bind` combinator that sequentially composes two monadic computations into one. The `ret` combinator is implemented with the `Ret` constructor, while the `bind t k` combinator is implemented as a coinductive function that traverses the ITree `t` and replaces each leaf `Ret r` with the new subtree `k r`. This is implemented in the following Coq code:

```
CoFixpoint bind (t : itree E R) (k : R → itree E S) :=
  match t with
  | Ret r ⇒ k r
  | Tau t ⇒ Tau (bind t k)
  | Vis e kvis ⇒ Vis e (fun x ⇒ bind (kvis x) k)
  end.
```

2.2 Mutually Recursive Computations

This section explains the recursion operator introduced by Xia et al. [29]. That work demonstrated how to use events as a piece of syntax for writing collections of mutually recursive functions over ITrees. Specifically, it introduced the `mrec` combinator, which lifts a collection of function bodies that syntactically reference one another to a collection of actually recursive functions. A similar recursion combinator is used extensively in Section 4 and Section 6.

When using the `mrec` combinator, you must first choose an event type `D`, with an `EncodingType` instance, to serve as the type of recursive calls. An element `d : D` packages together the choice of the function being called along with the arguments being supplied to that function. The return type of the function call `d` is `response_type d`. In this context, an ITree with the type `itree (D + E) R` represents the body of a mutually recursive function viewing the recursive calls as inert `D` events. This ITree defines a recursive function in terms of *syntactic* recursive calls. In order to resolve these syntactic recursive calls, we need a mapping from recursive calls to a single layer of unfolding of the recursive function. This is represented as a function of type `bodies : forall (d:D), itree (D + E) (response_type d)`. The variable name `bodies` refers to the fact that this term represents the body of each function in this collection of mutually recursive functions. We can then take this ITree, corecursively replace each `d : D` event with the unfolded function body `bodies d`, and then repeat the process with the resulting ITree. This is formalized in the following `interp_mrec` function.

```
CoFixpoint interp_mrec {R : Type}
  (bodies : forall (d:D), itree (D + E) (response_type d))
  (t : itree (D + E) R) : itree E R :=
  match t with
  | Ret r ⇒ Ret r
  | Tau t ⇒ Tau (interp_mrec bodies t)
  | Vis (inr e) k ⇒ Vis e (fun x ⇒ interp_mrec bodies (k x))
  | Vis (inl d) k ⇒ Tau (interp_mrec bodies (bind (bodies d) k))
  end.
```

Given this function that can resolve the recursive calls in an ITree, we can define the `mrec` function that takes an initial recursive call `init : D` and computes its result.

```
Definition mrec (bodies : forall (d:D), itree (D + E) (response_type d)) (init : D)
    :=
  interp_mrec bodies (bodies init).
```

Figure 4 provides an example of a mutually recursive function defined with `mrec`. The `evenoddE` type represents calls to compute the parity of a natural number. The `evenodd` function computes either the `even` or the `odd` function depending on the initial recursive call

```
Variant evenoddE : Type:=
  | even (n : nat) : evenoddE
  | odd (n : nat) : evenoddE.
Instance EncodingType_evenoddE : EncodingType evenoddE := fun _ ⇒ bool.

Definition evenodd_body : forall eo : evenoddE, (itree (evenoddE + voidE)) (
     response_type eo) :=
  fun eo ⇒
    match eo with
    | even n ⇒ if Nat.eqb n 0
               then Ret true
               else trigger (odd (n -1))
    | odd n ⇒ if Nat.eqb n 0
               then Ret false
               else trigger (even (n -1))
    end.
Definition evenodd : evenoddE → itree voidE bool :=
  mrec evenodd_body.
```

Figure 4 `evenodd` Definition

```
Definition Rel (A B : Type) : Type := A → B → Prop.
Definition PostRel (D1 D2 : Type) `{EncodingType D1} `{EncodingType D2} : Type :=
     forall (d1 : D1) (d2 : D2), response_type d1 → response_type d2 → Prop.


Inductive RComposePostRel
  (R1 : Rel D1 D2) (R2 : Rel D2 D3) (PR1 : PostRel D1 D2) (PR2 : PostRel D2 D3) :
  PostRel D1 D3 :=
  | RComposePostRel_intros (d1 : D1) (d3 : D3) (a : response_type d1) (c :
       response_type d3) :
    (forall (d2 : D2), R1 d1 d2 → R2 d2 d3 →
     exists b, PR1 d1 d2 a b ∧ PR2 d2 d3 b c) →
    RComposePostRel R1 R2 PR1 PR2 d1 d3 a c.
```

Figure 5 Heterogeneous Event Relation Types

event that it is given. The `evenodd` function defines these computations mutually recursively using the `mrec` function.

This section briefly introduces the classes of relations that we will need in order to reason about specification refinement in the presence of mutually recursive computations. The definition of `eutt` is parameterized by a return relation, making it easy to define a relation for ITrees that have identical tree structures up to Taus, with identical event nodes, but allows freedom to choose what conditions to enforce on return values. It is natural to consider generalizing `eutt` to allow variation not only in the return values but also in the event nodes. This kind of generalization is explored in Silver and Zdancewic [25][2]. The generalized relation analyzes uninterpreted events, typically those representing recursive function calls, with respect to pre-conditions and post-conditions. We want to relate `Vis` nodes whose events satisfy the pre-condition and whose continuations are related given any inputs that satisfy the post-condition. This corresponds to assuming that two function calls return related outputs as long as they are given related inputs.

Definitions of pre-condition and post-condition types are presented in Figure 5. Pre-conditions, `Rel`, are encoded as two-argument, heterogeneous relations, i.e. functions of type `D→E→Prop`, and utilize standard relational combinators like relational sums, `sum_rel`, and

[2] In Silver and Zdancewic [25] this relation is referred to as `euttEv`. It has since been renamed to `rutt` in release branches of the Interaction Trees library.

relational composition, `rcompose`. Post-conditions, `PostRel`, are encoded as four-argument, dependent relations. In particular, `forall (d:D) (e:E), encoded_by d → encoded_by e → Prop`, where both `D` and `E` have an `EncodingType` instance. Intuitively, post-conditions are a function from events to relations over their response types. These post-conditions admit a standard definition of relational sums. For relational composition, in addition to requiring two `PostRel` relations, it also requires two standard relations, called *coordinating relations*. The full definition is presented in Figure 5.

To relate four values `d1:D1`, `d3:D3`, `a:encoded_by d1`, `c:encoded_by d3`, we require that given any `d2:D2` that is related by the coordinating relations to `d1` and `d3`, there exists a `b:encoded_by d2` such that both `PR3 d1 d2 a b` and `PR4 d2 d3 b c`.

Later in the paper, we recover an `eutt`-like definition of specification refinement by specializing the event relations to be an appropriate form of equality. For `Rel`, this is precisely the equality relation. For `PostRel`, we define an inductive datatype that enforces equality on response values.

```
Variant PostRelEq : PostRel E E :=
  PostRelEq_intro e a : PostRelEq e e a a.
```

3 Specification Extraction with Heapster

This section introduces the Heapster tool for specification extraction. We present Heapster in order to provide context for the evaluation of this work in Section 6. In the evaluation, we demonstrate how effective ITree specifications can be when paired with a tool like Heapster. We start with a collection of low-level, heap manipulating C programs, use Heapster to produce equivalent functional programs, and finally use ITree specifications to specify and verify the output programs.

There is a growing body of work [3, 9, 17, 18] based on the idea that programs that satisfy memory-safe type systems like Rust can be represented with equivalent functional programs. Rust's pointer discipline, which ensures that all pointers in a program are either shared read or exclusive write, allows us to reason about the effects of pointer updates purely locally. This locality property can be used to define a pure functional model, referred to as a *functional specification*, of the behaviors of a program, which can in turn be used to verify properties of that program.

Whereas some work uses this notion of a functional model implicitly, *specification extraction* is the idea that the functional model can be extracted automatically as an artifact that can be used for verification. Specification extraction separates verification into two phases: a type-checking phase, where the functions in a program are type-checked against user-specified memory-safe types; and a behavior verification phase, where the user verifies the specifications that are extracted from this type-checking process. The Heapster tool[9] is an implementation of the idea of specification extraction. Heapster provides a memory-safe, Rust-like type system for LLVM, along with a typechecker. Heapster also provides a translation from well-typed LLVM programs to monadic, recursive, interactive programs, modeled with ITrees, that describe a behavioral model of the original program. This translation is inspired by the Curry-Howard isomorphism. Heapster types are essentially a form of logical propositions regarding the heap, so, by the Curry-Howard isomorphism, it is natural to view typing derivations, a form of proof, as a program. We give a brief overview of the Heapster type system and its specification extraction process in this section and illustrate it with an example.

$$
\begin{array}{llll}
\text{Value Types} & T & ::= & \mathsf{bv}\ n \mid \mathsf{llvmptr}\ n \mid \cdots \\
\text{Expressions} & e & ::= & n \mid \mathsf{llvmword}\ e \mid \cdots \\
\text{RW Modality} & rw & ::= & \mathsf{W} \mid \mathsf{R} \\
\text{Permissions} & \tau & ::= & \mathsf{ptr}((rw, e) \mapsto \tau) \mid \tau_1 * \tau_2 \mid \tau_1 \vee \tau_2 \mid \exists x{:}T.\tau \mid \mathsf{eq}(e) \mid \mu X.\tau \mid X \mid \cdots
\end{array}
$$

Figure 6 An Abbreviated Grammar of the Heapster Type System

The Heapster type system is a permission type system. Typing assertions of the form $x : \tau$ mean that the current function holds permissions to perform actions allowed by τ on the value contained in variable x. The central permission construct of Heapster is the permission to read or write a pointer value. Like Rust, Heapster is an affine type system, meaning that the permissions held by a function can change at different points in the function. In particular, a command can consume a permission, preventing further commands from using that permission again. Also like Rust, Heapster allows read-only permissions to be duplicated, allowing multiple read-only pointers to the same address, but does not allow write permissions to be duplicated. This enforces the invariant that all pointers are either shared read or exclusive write, a powerful property for proving memory-safety.

Figure 6 gives an abbreviated grammar for the Heapster type system. The value types T are inhabited by pieces of first order data. In particular, they contain the type $\mathsf{bv}\ n$ of n-bit bitvectors (i.e., n-bit binary values) and the type $\mathsf{llvmptr}\ n$ of n-bit LLVM values, among other value types not discussed here. Heapster uses the CompCert memory model [14], where LLVM values are either a word value or a pointer value represented as a pair of a memory region plus an offset in that region. The expressions e include numeric literals n and applications of the $\mathsf{llvmword}$ constructor of the LLVM value type to build an LLVM value from a word value.

The first permission type in Figure 6, $\mathsf{ptr}((rw, e) \mapsto \tau)$, represents a permission to read or write (depending on rw) a pointer at offset e. Write permission always includes read permission. This permission also gives permission τ to whatever value is currently pointed to by the pointer with this permission. Permission type $\tau_1 * \tau_2$ is the separating conjunction of τ_1 and τ_2, giving all of the permissions granted by τ_1 or τ_2, where τ_1 and τ_2 contain no overlapping permissions. Permission type $\tau_1 \vee \tau_2$ is the disjunction of τ_1 and τ_2, which either grants permissions τ_1 or τ_2. The existential permission $\exists x{:}T.\tau$ gives permission τ for some value x of value type T. The equality permission $\mathsf{eq}(e)$ states that a value is known to be equal to an expression e. This can be viewed as a permission to assume the given value equals e. Finally, $\mu X.\tau$ is the least fixed-point permission, where permission variable X is bound in τ. This satisfies the fixed-point property, that $\mu X.\tau$ is equivalent to $[\mu X.\tau/X]\tau$.

As a simple example, the user can define the Heapster type

$$\mathsf{int64} = \exists x{:}\mathsf{bv}\ 64.\mathsf{eq}(\mathsf{llvmword}\ x)$$

This Heapster type describes an LLVM word value, i.e., an LLVM value that equals $\mathsf{llvmword}\ x$ for some bitvector x.

As a slightly more involved example, consider the following definition of a linked list structure in C:

```
typedef struct list64_t { int64_t data;
                          struct list64_t *next; } list64_t;
```

```
int64_t is_elem (int64_t x, list64_t *l) {
  x:int64, l:list64⟨R⟩
  x:int64, l:eq(llvmword 0)  OR  x:int64, l:ptr((R,0) ↦ int64) * ptr((R,8) ↦ list64⟨R⟩)
  if (l == NULL) {
    x:int64, l:eq(llvmword 0)
    return 0;
  } else {
    x:int64, l:ptr((R,0) ↦ int64) * ptr((R,8) ↦ list64⟨R⟩)
    if (l->data == x) { return 1; }
    else {
      list64_t *l2 = l->next;
      x:int64, l:ptr((R,0) ↦ int64) * ptr((R,8) ↦ eq(l2)), l2:list64⟨R⟩
      return is_elem (x, l2);
}}}
```

Figure 7 Type-checking the `is_elem` Function Against Type $\texttt{x}:\mathsf{int64}, \texttt{l}:\mathsf{list64}\langle R\rangle \multimap r:\mathsf{int64}$

A C value of type `list64_t*` represents a list, where a NULL pointer represents the empty list and a non-NULL pointer to a `list64_t` struct represents a list whose head is the 64-integer contained in the `data` field and whose tail is given by the `next` field.

The following Heapster type describes this linked list structure:

$$\mathsf{list64}\langle rw\rangle = \mu X.\mathsf{eq}(\mathsf{llvmword}\ 0) \vee (\mathsf{ptr}((rw, 0) \mapsto \mathsf{int64}) * \mathsf{ptr}((rw, 8) \mapsto X))$$

The $\mathsf{list64}\langle rw\rangle$ type is parameterized by a read-write modality rw, which says whether it describes a read-only or read-write pointer to a linked list. The permission states that the value it applies to either equals the NULL pointer, represented as $\mathsf{llvmword}\ 0$, or points at offset 0 to a 64-bit integer and at offset 8[3] to an LLVM value that itself recursively satisfies the $\mathsf{list64}\langle rw\rangle$ permission. Note that the fact that it is a least fixed-point implicitly requires the list to be loop-free.

Figure 7 illustrates the process of Heapster type-checking on a simple function `is_elem` that checks if 64-bit integer `x` is in the linked list `l`. Note that Heapster in fact operates on the LLVM code that results from compiling this C code, but the type-checking is easier to visualize on the C code rather than looking at its corresponding LLVM. Ignoring the Heapster types for the moment, which are displayed with a grey background in the figure, `is_elem` first checks if `l` is `NULL`, and if so returns `0` to indicate that the check has failed. If not, it checks if the head of the list in `l->data` equals `x`, and if so, returns `1`. Otherwise, it recurses on the tail `l->next`.

The Heapster permissions for this function are

$$\texttt{x}:\mathsf{int64}, \texttt{l}:\mathsf{list64}\langle R\rangle \multimap r:\mathsf{int64}$$

The lollipop symbol, $\multimap$, is used to write Heapster function types. This type means that input `x` is a 64-bit integer and `l` is a read-only linked list pointer and the return value r is a 64-bit integer value.

To type-check `is_elem`, Heapster starts by assuming the input types for the arguments. This is displayed in the first grey box of Figure 7. In order to type-check the `NULL` comparison

[3] We assume a 64-bit architecture, so offset 8 references the second value of a C struct.

on `l`, Heapster must first unfold the recursive permission on `l` and then eliminate the resulting disjunctive permission. This latter step results in Heapster type-checking the remaining code twice, once for each branch of the disjunct. More specifically, the remaining code is type-checked once under the assumption that `l` equals `NULL` and once under the assumption that it points to a valid `list64_t` struct. In the first case, the NULL check is guaranteed to succeed, and so the `if` branch is taken with those permissions, while in the second, the NULL check is guaranteed to fail, so the `else` branch is taken.

In the `if` branch, the value `0` is returned. Heapster determines that this value satisfies the required output permission int64. In the `else` branch, `l->data` is read, by dereferencing `l` at offset 0. This is allowed by the permissions on `l` at this point in the code. If the resulting value equals `x`, then `1` is returned, which also satisfies the output permission int64. Otherwise, `l->next` is read, by dereferencing `l` at offset 0, and the result is assigned to local variable `l2`. This assigns $\mathsf{list64}\langle R\rangle$ permission to `l2`. The permission on offset 8 of `l` is updated to indicate that the value currently stored there equals `l2`. The $\mathsf{list64}\langle R\rangle$ permission on `l2` is then used to type-check the subsequent recursive call to `is_elem`.

Once a function is type-checked, Heapster performs specification extraction to extract a pure functional specification of the function's behavior. Specification extraction translates permission types to Coq types and typing derivations to Coq programs. The type translation is defined as follows:

$$\begin{array}{rcl@{\qquad}rcl}
[\![\mathsf{ptr}((rw,e)\mapsto\tau)]\!] &=& [\![\tau]\!] & [\![\tau_1 * \tau_2]\!] &=& [\![\tau_1]\!] * [\![\tau_2]\!] \\
[\![\tau_1 \vee \tau_2]\!] &=& [\![\tau_1]\!] + [\![\tau_2]\!] & [\![\exists x{:}T.\tau]\!] &=& \{x : [\![T]\!] \;\&\; [\![\tau]\!]\} \\
[\![\mathsf{eq}(e)]\!] &=& \mathsf{unit} & [\![\mu X.\tau]\!] &=& \text{user-specified type } A \\
&&&&& \text{isomorphic to } [\![[\mu X.\tau/X]\tau]\!]
\end{array}$$

Pointer permissions $\mathsf{ptr}((rw,e)\mapsto\tau)$ are translated to the result of translating the permission τ of the value that is pointed to. This means that specification extraction erases pointer types, which are no longer needed in the resulting functional code. Conjuctive permissions are translated to pairs, disjunctive permissions are translated to sums, and existential permissions are translated to dependent pairs (using a straightforward translation $[\![T]\!]$ of value types that we omit here). The equality type $\mathsf{eq}(e)$ is translated to the Coq unit type unit, meaning that they contain no data in the extracted specifications. We already proved the equality in the typechecking phase, and we have no use for the particular equality proof the typechecker provided. To translate a least fixed-point type $\mu X.\tau$, the user specifies a type that satisfies the fixed-point equation, meaning a pair of functions

$$\mathsf{fold} : [\![[\mu X.\tau/X]\tau]\!] \to [\![\mu X.\tau]\!] \qquad \mathsf{unfold} : [\![\mu X.\tau]\!] \to [\![[\mu X.\tau/X]\tau]\!]$$

that form an isomorphism.

As an example, the translation of int64 is the Coq sigma type `{x:bitvector 64 & unit}`. Note that Heapster will in fact optimize away the unnecessary `unit` type, yielding the type `bitvector 64`. As a slightly more complex example, in order to translate the $\mathsf{list64}\langle rw\rangle$ described above, the user must provide a type `T` that is isomorphic to the type

```
unit + (bitvector 64 * T)
```

The simplest choice for `T` is the type `list (bitvector 64)`. In this way, the imperative linked list data structure defined above in C is translated to the pure functional list type.

Rather than defining the translation of Heapster typing derivations into Coq programs here, we illustrate the high-level concepts with our example and refer the interested reader to He et al. [9] for more detail. The translation of `is_elem` is given as a Coq specification

```
Definition is_elem_spec : bitvector 64 * list (bitvector 64) →
                          itree_spec E (bitvector 64) :=
  rec_fix_spec (fun rec '(x,l) ⇒
                either
                  unit (bitvector 64 * list (bitvector 64)) (* input types *)
                  (itree_spec _ (bitvector 64))            (* output type *)
                  (fun _ ⇒ Ret (intToBv 64 0))             (* nil case *)
                  (fun '(hd,tl) ⇒                          (* cons case *)
                     if bvEq 64 hd x then Ret (intToBv 64 1) (* return 1 *)
                     else rec (x,tl))                        (* recursive call *)
                  (unfoldList l)).                           (* unfolded argument *)
```

Figure 8 Extracted Specification for `is_elem`

`is_elem_spec` in Figure 8. At the top level, this specification uses `rec_fix_spec` to define a recursive function to match the recursive definition of `is_elem`. This binds a local variable `rec` to be used for recursive calls to the specification.

To understand the rest of the specification, we step through the Heapster type-checking depicted in Figure 7. The first step of that type assignment unfolds the permission type $\mathsf{list64}\langle W\rangle$ on `l`. The corresponding portion of the specification is the call to `unfoldList`, which unfolds the input list `l` to a sum of a unit or the head and tail of the list. The next step of the Heapster type-checking is to eliminate the resulting disjunctive permission on `l`. The corresponding portion of the specification is a call to the `either` sum elimination function. In the left-hand case of the disjunctive elimination, the NULL test of the C program succeeds, and `0` is returned. Similarly, in the Coq specification, the `nil` case returns the 0 bitvector value.

In the right-hand case of the disjunctive elimination of the Heapster type-checking, the NULL test fails, and so `l` is a valid pointer to a C struct with `data` and `next` fields. This is represented by the pattern-match on the cons case in the Coq specification, yielding variables `hd` and `tl` for the head and tail of the list. The body of this case then tests whether the head equals the input variable `x`, corresponding to the `x==l->data` expression in the C program. If so, then the bitvector value 1 is returned. Otherwise, the specification performs a recursive call, passing the same value for `x` and the tail of the input list for `l`.

4 ITree Specifications and Refinement

In this paper, we introduce a specialization of the ITree data type that encodes specifications over ITrees. To do this, we take some base event type family `E`, and extend it with constructors for universal and existential quantification. This is formalized in the following definition for `SpecEvent`.

```
Inductive SpecEvent (E : Type) `{EncodingType E} : Type :=
  | Spec_vis (e : E) : SpecEvent E
  | Spec_forall (A : type) : SpecEvent E
  | Spec_exists (A : type) : SpecEvent E
.
```

The `Spec_vis` constructor allows you to embed a base event `e : E` into the type `SpecEvent E`. The `Spec_forall` constructor signifies universal quantification, and the `Spec_exists` constructor signifies existential quantification. For the purposes of specifying Heapster programs, we only need to quantify over a fixed grammar of first order types[4]. This includes natural numbers, bit vectors, functions, products, logical propositions, and sums. We have omitted the definition of the particular fixed grammar of types used in this work for space.

We define *ITree specifications* as the type of ITrees with a `SpecEvent` as the event type.

```
Definition itree_spec (E : Type) `{EncodingType E} (R : Type) :=
   itree (SpecEvent E) R.
```

Because ITree specifications are actually a special kind of ITree, they inherit all the useful metatheory and code defined for ITrees. In particular, we can reason about them equationally with `eutt`, and apply the monad functions to them.

4.1 ITree Specification Refinement

The notion that a program adheres to a specification is defined with the notion of refinement. Refinement is the main judgment involved in using ITree specifications, and is for instance the primary form of proof goal proved by the provided automation tool. Intuitively, the logical quantifier events mean that an ITree specification represents a set of computations. A fully concrete ITree specification, with no logical quantifier events, represents a singleton set, while a more abstract specification might represent a larger set. The refinement relation is then defined such that, if one ITree specification refines another, then the former represents a subset of the latter. So, for instance, if we prove that a concrete specification refines a more abstract specification, then we have shown that the singleton program in the set represented by the concrete specification satisfies the specification. Note that refinement is actually a coarser relation than subset; this is discussed later in Section 4.4.

The ITree specification refinement relation is based on the idea of refinement of logical formulae with the `eutt` relation. As in a sequent calculus, we can eliminate quantifiers in our specification logic using quantifiers in the base logic, in this case Coq. Quantifiers on the right of a refinement get eliminated to the corresponding Coq quantifiers, while quantifiers on the left get eliminated to the dual of the corresponding Coq quantifier. This means that both a `Spec_forall` on the right and a `Spec_exists` on the left get eliminated to a Coq `forall`. And both a `Spec_exists` on the right and a `Spec_forall` on the left get eliminated to a Coq `exists`. ITree specifications form a lattice with refinement serving as the preorder, `Spec_forall` acting as the complete meet, and `Spec_exists` acting as the complete join. The portions of ITree specifications with computational content, including the `Ret` leaves, `Spec_vis` nodes, and silent `Tau` nodes, get compared as they do in the `eutt` relation.

The ITree specification refinement relation shares many mechanical details with the `eutt` relation. Both are defined by taking the greatest fixed point of an inductively defined relation to get a mixture of inductive and coinductive properties. Both behave identically on `Tau` and `Ret` nodes. The refinement relation differs in its inductive rules for eliminating logical quantifiers, and in its usage of heterogeneous event relations to enforce pre- and post-conditions on `Spec_vis` events. These pre- and post- conditions are necessary in order to give the refinement relation the flexibility needed to state the reasoning principle for `mrec`. The initial inductively defined relation, `refinesF`, contains the following header code.

```
Inductive refinesF
          (RPre : Rel E1 E2) (RPost : PostRel E1 E2) (RR : Rel R1 R2)
          (sim : itree_spec E1 R1 → itree_spec E2 R2 → Prop)
   : itree_spec E1 R1 → itree_spec E2 R2 → Prop :=
```

[4] While we could quantify over `Type` in these definitions, this introduces universe level constraints that we prefer to avoid

Much like in the definition of `euttF`, the `sim` argument represents corecursive calls of the `refines` relation, and the `RR` argument is the relation used for return. Unlike in `euttF`, `refinesF` takes in arguments for a `PreRel` and a `PostRel`. These arguments are included in order to represent pre- and post- conditions on mutually recursive function bodies.

The `refinesF` relation has several constructors that work precisely the same as the corresponding `euttF` constructors. These constructors define the relation's behavior on `Ret` and `Tau` nodes.

```
| refines_Ret (r1 : R1) (r2 : R2) : RR r1 r2 → refinesF RPre RPost RR sim (Ret r1)
     (Ret r2)
| refines_Tau (phi1 : itree_spec E1 R1) (phi2 : itree_spec E2 R2) : sim phi1 phi2
   →
                                    refinesF RPre RPost RR sim (Tau phi1) (Tau phi2)
| refines_TauL (t1 : itree_spec E1 R1) (t2 : itree_spec E2 R2) :
  refinesF RPre RPost RR sim t1 t2 → refinesF RPre RPost RR sim (Tau t1) t2
| refines_TauR (t1 : itree_spec E1 R1) (t2 : itree_spec E2 R2) :
  refinesF RPre RPost RR sim t1 t2 → refinesF RPre RPost RR sim t1 (Tau t2)
```

The constructor dealing with `Spec_vis` nodes generalizes the constructor dealing with `Vis` nodes in `euttF`. This constructor relates `Spec_vis` nodes as long as two conditions hold on the events, `e1` and `e2`, and the continuations, `k1` and `k2`. The ITree specifications must satisfy the precondition, by having `e1` and `e2` satisfy `RPre`. And the ITree specifications must satisfy the post condition by having `k1 a` refine `k2 b`, whenever `a` and `b` are related by `RPost e1 e2`.

```
| refines_Spec_vis (e1 : E1) (e2 : E2)
                  (k1 : response_type e1 → itree_spec E1 R1) (k2 : response_type e2
                        → itree_spec E2 R2) :
  RPre e1 e2 → (forall a b, RPost e1 e2 a b → sim (k1 a) (k2 b)) →
  refinesF RPre RPost RR sim (Vis (Spec_vis e1) k1) (Vis (Spec_vis e2) k2)
```

The added complications of this rule allow us to reason about mutually recursive functions. It ensures that related function outputs assume that function calls with arguments related by the precondition return values related by the post condition when analyzing mutually recursive functions.

Finally, we need constructors dealing with quantifier events. This definition uses only inductive constructors to eliminate quantifier events. We made this choice to avoid certain peculiar issues related to ITree specifications that consist of infinite trees of only quantifiers. Given coinductive constructors for quantifier events, we would be able to prove that such ITree specifications both refine and are refined by any other arbitrary ITree specification. That choice would cause certain ITree specifications to serve as both the top and bottom elements of the refinement order. This would serve as a counterexample to the transitivity of refinement, a desired property. So we chose to only use inductive constructors for quantifier events. This means that ITree specifications that consist of infinite trees of only quantifiers cannot be related by refinement to any other ITree specifications.

Quantifiers on the right get directly translated into Coq level quantifiers.

```
| refines_forallR (t : itree_spec E1 R1) (A:type) (k : response_type A →
     itree_spec E2 R2) :
  (forall a, refinesF RPre RPost RR sim t (k a)) →
  refinesF RPre RPost RR sim t (Vis (Spec_forall A) k)
| refines_existsR (t : itree_spec E1 R1) (A : type) (k : response_type A →
     itree_spec E2 R2) :
  (exists a, refinesF RPre RPost RR sim t (k a)) →
  refinesF RPre RPost RR sim t (Vis (Spec_exists A) k)
```

Quantifiers on the left get translated into their dual quantifier at the Coq level. Eliminating a `Spec_forall` on the left gives you an `exists`. Eliminating a `Spec_exists` on the left gives you an `forall`.

```
Class CoveredType (A : Type) := {
    encoding : type;   surjection : response_type encoding → A;
    surjection_correct : forall a : A, exists x, surjection x = a; }.
```

```
Definition forall_spec {E}
      `{EncodingType E}
       (A:Type) `{CoveredType A} :
  itree_spec E A :=
  Vis (Spec_forall encoding)
      (fun x ⇒ Ret (surjection x)).
```

```
Definition exists_spec {E}
      `{EncodingType E}
       (A:Type) `{CoveredType A} :
  itree_spec E A :=
  Vis (Spec_exists encoding)
      (fun x ⇒ Ret (surjection x)).
```

```
Definition assume_spec {E}
  `{EncodingType E} (P : Prop) :
  itree_spec E unit :=
  forall_spec P;; Ret tt.
```

```
Definition assert_spec {E}
  `{EncodingType E} (P : Prop) :
  itree_spec E unit :=
  exists_spec P;; Ret tt.
```

Figure 9 Basic Specifications

```
      | refines_forallL (A : type) (k : response_type (Spec_forall A) → itree_spec E1 R1)
            (t : itree_spec E2 R2) :
        (exists a, refinesF RPre RPost RR sim (k a) t) →
        refinesF RPre RPost RR sim (Vis (Spec_forall A) k) t
      | refines_existsL (A : type) (k : response_type (Spec_exists A) → itree_spec E1 R1)
            (t : itree_spec E2 R2) :
        (forall a, refinesF RPre RPost RR sim (k a) t) →
        refinesF RPre RPost RR sim (Vis (Spec_exists A) k) t
```

This `refinesF` relation is used to define the `refines` relation as follows.

```
Definition refines RPre RPost RR := gfp (refinesF RPre RPost RR).
```

4.2 Padded ITrees

Useful refinement relations should respect the `eutt` relation. When using ITrees as a denotational semantics, `eutt` is the basis of any program equivalence relation. Equivalent programs and specifications should not be observationally different according to the refinement relation. However, the `refines` relation does not respect `eutt`

We can easily demonstrate this with the following three ITree specifications.

```
CoFixpoint spin : itree_spec E R := Tau spin.
CoFixpoint phi1 : itree_spec E R := Vis (Spec_forall t) (fun _ ⇒ Tau (phi1)).
CoFixpoint phi2 : itree_spec E R := Vis (Spec_forall t) (fun _ ⇒ phi2).
```

The `spin` specification represents a silently diverging computation. The `phi1` specification is an infinite stream that alternates between `Spec_forall` nodes and `Tau` constructors. The `phi2` specification is a similar ITree to `phi1` that just lacks the `Tau` nodes. As these ITree specifications all diverge along all paths and lack any `Spec_vis` nodes, the `RPre`, `RPost`, and `RR` relations that we choose do not matter. Given any choice for those relations, `spin` refines `phi1` as we can use the inductive `refines_forallL` rule to get rid of the `Spec_forall` nodes, allowing us to match `Tau` nodes on both trees and apply the coinductive `refines_Tau` rule. This process can be extended coinductively allowing us to construct the refinement proof. The `phi1` ITree specification is `eutt` to `phi2`, as the only difference between the specifications is a single `Tau` node after every `Vis_forall` node. However, `spin` does not refine `phi2`, as there is no coinductive constructor that we can apply in order to write a proof for these divergent

```
CoFixpoint interp_mrec_spec {R : Type}
  (bodies : forall (d:D), (itree_spec (D + E)) (response_type d)) (t : itree_spec (D + E
     ) R) : itree_spec E R :=
  match t with
  | Ret r ⇒ Ret r
  | Tau t ⇒ Tau (interp_mrec_spec bodies t)
  | Vis (Spec_forall A) k ⇒ Vis (@Spec_forall E _ A) (fun x : response_type (Spec_forall
      A) ⇒ interp_mrec_spec bodies (k x))
  | Vis (Spec_exists A) k ⇒ Vis (@Spec_exists E _ A) (fun x ⇒ interp_mrec_spec bodies (
     k x))
  | Vis (Spec_vis (inr e)) k ⇒ Vis (Spec_vis e) (fun x ⇒ interp_mrec_spec bodies (k x))
  | Vis (Spec_vis (inl d)) k ⇒ Tau (interp_mrec_spec bodies (bind (bodies d) k))
  end.

Definition mrec_spec (bodies : forall (d:D), (itree_spec (D + E)) (response_type d)) (
    init : D) :=
  interp_mrec_spec bodies (bodies init).
```

Figure 10 `mrec_spec` Definition

ITree specifications. Problems like this arise with any ITree specifications that consist of infinitely many quantifier nodes with nothing between them.

To fix this problem, we restrict our focus to a subset of ITrees that does not include ones like `phi2`. This is the set of *padded* ITrees, in which every `Vis` node must be immediately followed by a `Tau`. We formalize this with the coinductive `padded` predicate, whose definition has been omitted to save space. The refinement relation does not distinguish between different ITree specifications that are `eutt` to one another as long as they are padded. This means that can rewrite one ITree specification into another under a refinement according to `eutt` as long as both are padded.

Furthermore, it is easy to take an arbitrary ITree, and turn it into a padded ITree. That is implemented by the `pad` function, which corecursively adds a `Tau` after every `Vis` node. From here, we can focus primarily on the following definition of `padded_refines` which pads out all ITree specifications before passing them to the `refines` relation.

```
Definition padded_refines RPre RPost RR phi1 phi2 :=
  refines RPre RPost RR (pad phi1) (pad phi2).
```

In Figure 9, we introduce several simple ITree specifications that implement quantification over some types, and assumption and assertion of propositions. The `forall_spec` and `exists_spec` specifications rely on the `CoveredType` type class. A `CoveredType` instance for a type `A` contains an element of the restricted type grammar, `encoding`, whose interpretation corresponds to `A`. It also contains a valid surjection from the interpreted type `response_type encoding` to the original type `A`. In practice, we always instantiate this surjection with the identity function, but this type class formalization gives us the tools that we need without needing to do too much dependently typed programming. We can use `forall_spec` and `exists_spec` to define assumption and assertion, respectively, as `Prop` is part of the restricted grammar of types that `SpecEvent` can quantify over.

4.3 Padded Refinement Meta Theory

This subsection introduces some of the useful, verified metatheory we provide for ITree specifications in terms of `padded_refines` relation.

We prove that we can compose refinement results with the monadic `bind` operator.

```
Theorem padded_refines_bind (phi1 : itree_spec E1 R1) (phi2 : itree_spec E2 R2)
                            (kphi1 : R1 → itree_spec E1 S1)
                            (kphi2: R2 → itree_spec E2 S2) :
  padded_refines RPre RPost RR phi1 phi2 →
  (forall r1 r2, RR r1 r2 → padded_refines RPre RPost RS (kphi1 r1) (kphi2 r2)) →
  padded_refines RPre RPost RS (bind phi1 kphi1) (bind phi2 kphi2).
```

We prove that the padded_refines relation is transitive. To state the transitivity result in full generality, we need to use the composition relation introduced in Figure 5.

```
Theorem padded_refines_trans : forall (phi1 : itree_spec E1 R1) (phi2 : itree_spec E2
      R2) (phi3 : itree_spec E3 R3),
  padded_refines RPre1 RPost1 RR1 phi1 phi2 →
  padded_refines RPre2 RPost2 RR2 phi2 phi3 →
  padded_refines (RCompose RPre1 RPre2)
    (RComposePostRel RPre1 RPre2 RPost1 RPost2) (RCompose RR1 RR2) phi1 phi3.
```

We prove a reasoning principle for mutually recursive specifications as well. To do this, we first provide a slightly different definition of mutual recursion that handles the quantifier events correctly, defined in Figure 10. The key to proving refinements between mrec_spec specifications is to use the PreRel and PostRel relations to establish pre- and post-conditions on recursive calls. This involves choosing a PreRel over recursive call events, RPreInv, and a PostRel over recursive call events, RPostInv. Just like any form of invariants in formal verification, correctly choosing RPreInv and RPostInv requires striking a careful balance between choosing preconditions that are weak enough to hold, but strong enough to imply post conditions. The rule is expressed in the following code.

```
Theorem padded_refines_mrec : forall (init1 : D1) (init2 : D2),
    RPreInv init1 init2 →
    (forall d1 d2, RPreInv d1 d2 →
                padded_refines (SumRel RPreInv RPre)
                               (SumPostRel RPostInv RPost)
                               (RPostInv d1 d2)
                               (bodies1 d1) (bodies2 d2)) →
    padded_refines RPre RPost (RPostInv init1 init2)
                  (mrec_spec bodies1 init1)
                  (mrec_spec bodies2 init2).
```

The hypotheses in this theorem state that the initial recursive calls, init1 and init2, are in the precondition RPreInv, and that given any two recursive calls related by the precondition, d1 and d2, the recursive function bodies refine one another, where recursive calls are related by RPreInv and RPostInv and any other events are related by RPre and RPost. These reasoning principles allow us to prove complicated propositions involving the coinductively defined refinement relation without needing to perform direct coinduction.

While we include several parameter relations with the definition of padded_refines, at the top level, we are typically interested in the case where all relations are set to equality. We call this relation *strict refinement*, and refer to it with the ≤ symbol.

```
Notation "phi1 '≤' phi2" :=
  (padded_refines eq PostRelEq eq phi1 phi2).
```

Strict refinement is a transitive relation, and is strong enough to allow rewrites under the context of any other application of padded_refines.

4.4 ITree specification Incompleteness

One way to interpret ITree specifications is as sets of ITrees. The following code defines *concrete* ITree specifications, which correspond to executable ITrees.

```
Variant concreteF {E R} `{EncodingType E} (F : itree_spec E R → Prop) : itree_spec E
     R → Prop :=
  | concreteRet (r : R) : concreteF F (Ret r)
  | concreteTau (t : itree_spec E R) : F t → concreteF F (Tau t)
  | concreteVis (e : E) (k : response_type e → itree_spec E R) :
    (forall a, F (k a)) → concreteF F (Vis (Spec_vis e) k).
Definition concrete {E R} `{EncodingType E} : itree_spec E R → Prop := gfp concreteF.
```

A concrete ITree specification contains no quantifiers along any of its branches. We can map each ITree specification to the set of `concrete` ITree specifications that refine it.

However, ITree specifications are not complete with respect to this interpretation. In particular, there are pairs of ITree specifications that represent equivalent sets of concrete ITree specifications, but do not refine one another. To see why, consider the following two ITree specification over an empty event signature `voidE`.

```
Definition top1 : itree_spec voidE unit :=
  forall_spec void;; Ret tt.
```

```
Definition top2 : itree_spec voidE unit :=
  or_spec spin (Ret tt).
```

Both `top1` and `top2` are refined by all concrete ITree specifications of type `itree_spec voidE unit`. We can prove the refinement for `top1` by applying the right `forall` rule, and reducing to a trivially satisfied proposition. For `top2`, we know that every concrete ITree specification of this type is `eutt` to either `spin` or `Ret tt`[5]. In each case, apply the right `exists` rule and choose the corresponding branch. However, given any relations `RE`, `REAns`, `RR`, we cannot prove `padded_refines RE REAns RR top1 top2`. This is because the only way to eliminate the `Spec_forall` on the left is to provide an element of the `void` type, which does not exist. This, along with the transitivity theorem, demonstrates that `padded_refines` is strictly weaker than the subset relation on sets of refining concrete ITree specification.

5 Total Correctness Specifications

This section discusses how to encode and prove simple pre- and post- condition specifications using ITree specifications. We also discuss how these definitions relate to our syntax-directed proof automation.

Suppose we have a program that takes in values of type `A` and returns values of type `B`. Suppose we want to prove that if given an input that satisfies a precondition `Pre : A → Prop`, it will return a value that satisfies a postcondition `Post : A → B → Prop` without triggering any other events. The postcondition is a relation over `A` and `B` to allow the postcondition to depend on the initial provided value. We can encode these conditions in the following ITree specification.

```
Definition total_spec : A → itree_spec E B :=
  fun a ⇒ assume_spec (Pre a);;
       b ← exists_spec B;;
       assert_spec (Post a b);;
       Ret b.
```

The specification assumes that the input satisfies the precondition, existentially introduces an output value, asserts the post condition holds, and finally returns the output.

[5] Proving this fact requires a nonconstructive axiom like the Law of The Excluded Middle.

```
Definition call_spec (a : A) : itree_spec (callE A B + E) B := trigger (inl (Call a)).

Definition calling' {F} `{EncodingType F} : (A → itree F B) →
          (forall (c : callE A B) , itree F (response_type c)) :=
            fun f c ⇒ f (unCall c).
Definition rec_spec (body : A → itree_spec (callE A B + E) B) (a : A) :
  itree_spec E B :=
 mrec_spec (calling' body) (Call a).
Definition rec_fix_spec
          (body : (A → itree_spec (callE A B + E) B) → A →
           itree_spec (callE A B + E) B) :
  A → itree_spec E B :=
  rec_spec (body call_spec).
```

Figure 11 `rec_fix_spec` Definition

The `total_spec` specification can be effectively used compositionally. Consider a merge sort implementation, named `sort`, built on top of two recursively defined helper functions, one for splitting a list in half, named `halve`, and one for merging sorted lists, named `merge`. If we have already proven specializations of `total_spec` for these sub functions, it becomes easier to prove a specification for `sort`. Immediately we can replace these sub functions with their total correctness specification. Now consider how this total correctness specification will behave on the left side of a refinement. First, we can eliminate `assume_spec (Pre a)` as long as we can prove `Pre a`. Once we have done that, we get to universally introduce the output `b`, along with a proof that it satisfies the post condition. We are finally left with only `Ret b` with the assumption `Post a b`. This is a much simpler specification than our initial executable specification, which relied on several control flow operators including a recursive one.

However, this easy to use specification is not easy to directly prove. The `padded_refines_mrec` rule gives us a sound reasoning principle for proving that a recursively defined function refines another recursively defined function, but it does not give any direct insight into how to prove any refinement that does not match that syntactic structure. To address this, we introduce a recursively defined version of `total_spec_fix` that we can apply our recursive reasoning principle on.

First, we introduce a specialization of the `mrec_spec` combinator called `rec_fix_spec`, defined in Figure 11. The `rec_fix_spec` function has a type similar to that of a standard fixpoint operator. The first argument, `body`, is a function that takes in a type of recursive calls `A →  itree_spec (callE A B + E) B` and an initial argument of type `A` and produces a result in terms of an ITree specification. It relies on the `calling'` function to transform this value into a value of type `forall (c:callE A B), itree_spec (callE A B + E) B` which the `mrec_spec` function requires. From there it relies on the `call_spec` and `rec_spec` functions to wrap values of type `A` into `Call` events and `trigger` them.

Given this recursion operator, we introduce an equivalent version of the total correctness specification, `total_spec_fix`.

```
Definition total_spec_fix : A → itree_spec E B :=
  rec_fix_spec (fun rec a ⇒
                assume_spec (Pre a);;
                n ← exists_spec nat;;
                trepeat n (
                        a' ← exists_spec A;;
                        assert_spec (Pre a' ∧ Rdec a' a);;
                        rec a'
                      );;
                b ← exists_spec B;;
                assert_spec (Post a b);;
                Ret b).
```

This specification is reliant on the `trepeat n t` function, with simply binds an ITree, `t`, onto the end of itself `n` times. Note that `total_spec_fix` is defined recursively, and contains the elements of `total_spec` inside the recursive body. This makes it easier to relate to recursively defined functions. It begins by assuming the precondition and ends by introducing an output, asserting it satisfies the post condition, and returning the output. What comes between these familiar parts requires more explanation. Recall the discussion of the `padded_refines_mrec` rule. This reasoning principle lets you prove refinement between two recursively defined ITree specifications when a single layer of unfolding of each specification match up one to one with recursive calls.

This means that to have a useful, general, and recursively defined version of total correctness specification we need to allow our recursive definition for total correctness specification to choose the number of recursive calls the function requires. For this reason, `total_spec_fix` existentially introduces a number `n` that specifies how many recursive calls are needed for one level of unfolding of the recursive function starting at `a`. The specification then includes `n` copies of a specification that existentially chooses a new argument `a'`, asserts a predicate holds on it, and then recursively calls the specification on this new argument. This asserted predicate contains two parts. First, we assert the precondition. A correct recursively defined function should not call itself on an invalid input if given a valid input. Second, we assert that `a'` is *less than* `a` according to the relation `Rdec`. In order for `total_spec_fix` to actually be equivalent to `total_spec`, we need to assume that `Rdec` is well-founded[6]. The fact that `Rdec` is well-founded ensures that this specification contains no infinite chains of recursive calls. This allows us to prove that `total_spec_fix` refines `total_spec` as long as `Rdec` is well-founded.

```
Theorem total_spec_fix_correct :
  well_founded Rdec → forall (a : A), total_spec_fix a ≤ total_spec a.
```

This theorem allows us to initially prove refinement specifications for recursive functions using the `padded_refines_mrec` rule with `total_spec_fix` and then replace it with the easier to work with `total_spec`.

Both `total_spec` and `total_spec_fix` do not accept any ITree specifications that trigger any events. As a result, these total correctness specifications do not allow any exceptions to be raised, as you would expect with total correctness specifications.

5.1 Demonstration

To demonstrate how to work with `total_spec`, we describe how to verify the `merge` function, a key component of the merge sort algorithm. The `merge` function takes two sorted lists

[6] We use the Coq standard library's definition of well-foundedness for this.

```
Definition merge : (list nat * list nat)
    →
           itree_spec E (list nat) :=
  rec_fix_spec (fun rec '(l1,l2) ⇒
               b1 ← is_nil l1;;
               b2 ← is_nil l2;;
               if b1 : bool then
                 Ret l2
               else if b2 : bool then
                 Ret l1
               else
                 x ← head l1;;
                 tx ← tail l1;;
                 y ← head l2;;
                 ty ← tail l2;;
                 if Nat.leb x y then
                   l ← rec (tx, y::ty);;
                   Ret (x :: l)
                 else
                   l ← rec (x::tx, ty);;
                   Ret (y::l)).
```

```
Definition merge_pre p :=
  let '(l1,l2) := p in
  sorted l1 ∧ sorted l2.
Definition merge_post '(l1,l2) l :=
  sorted l ∧ Permutation l (l1 ++ l2).


Definition rdec_merge '(l1,l2) '(l3,l4) :=
  length l1 < length l3 ∧
    length l2 = length l4 ∨
  length l1 = length l3 ∧
    length l2 < length l4.


Theorem merge_correct : forall l1 l2,
    merge (l1,l2) ≤ total_spec merge_pre
        merge_post (l1,l2).
```

Figure 12 Merge implementation

and combines them into one larger sorted list which contains all the original elements. In Figure 12, we present a recursively defined implementation of `merge` along with relevant relations and the correctness theorem. The `merge` function is based on the standard list manipulating functions `is_nil`, `head`, and `tail`. We assume that the event type `E` contains some kind of error event which is emitted if `head` or `tail` is called on an empty list.[7]

The `merge` function relies on its arguments being sorted and guarantees that its output is a single, sorted list that is a permutation of the concatenation of the original lists. We formalize these conditions in `merge_pre` and `merge_post`. To prove that `merge` is correct, we want to show that it refines the total specification built from its pre- and post- conditions. To accomplish this, it suffices to choose a well founded relation and prove that `merge` satisfies the resulting `total_spec_fix` specification. For this function, we use `rdec_merge` which ensures that the pairs of lists that we recursively call `merge` on either both decrease in length, or one decreases in length and the other has the same length.

This leaves us with a refinement goal between two recursively defined specifications. We can then apply the `padded_refines_mrec_spec` theorem. For the relational precondition, we require that each pair of `Call` events is equal, and that `Pre` holds on the value contained within the call. For the relational postcondition, we require that equal `Call` events return equal values and that `Post` holds on them. Finally, we can prove that the body `merge` refines the body of `total_spec_fix` given these relation pre- and postconditions. We accomplish this by setting the existential variables on the right to make a single recursive call and give it the same argument as the recursive call that the body of `merge` makes.

With this technique, we can verify the simple server introduced in Section 1. Recall that the `server_impl` program executes an infinite loop of receiving a list of numbers, sorting it, and sending it back as a message. To verify `server_impl`, we first verify `halve`, the remaining sub function of sort, using the same technique we used to prove the correctness of `merge`. We can then use these facts to prove the correctness of `sort`, and use the correctness of `sort` to

[7] We manage this assumption with a Coq type class called `ReSum`. For more information please read the original ITrees paper[30] or inspect the associated artifact.

Function Name	Description	C LoC	Proof LoC
mbox_free_chain	Deallocate an mbox chain	11	18
mbox_len	Compute the length in bytes of an mbox chain	9	40
mbox_concat	Concatenates an mbox chain after a single mbox	5	18
mbox_concat_chains	Concatenates two mbox chains	14	24
mbox_split_at	Split an mbox chain into two chains	25	147
mbox_copy	Copy a single mbox	13	74
mbox_copy_chain	Copy an mbox chain	18	173
mbox_detach	Detach the first mbox from a chain	18	18
mbox_detach_from_end	Detach the first N bytes from an mbox chain	3	50
mbox_randomize	Randomize the contents of an mbox	9	121
mbox_drop	Remove bytes from the start of an mbox	12	23

Figure 13 Verified `mbox` functions

prove the correctness of `server_impl`.

```
Theorem server_correct :
  (server_impl tt) ≤ (server_spec tt).
```

6 Automation and Evaluation

6.1 Auto-active Verification

A key goal of this work is to provide auto-active automation for ITree specifications refinement. To this effect, the current section presents an automated Coq tactic for proving refinement goals called `prove_refinement`. The `prove_refinement` tactic is designed to reduce proof goals about refinement of programs to proof goals about the data and assertions used in those programs. In the spirit of auto-active verification, this is done mostly automatically, but with the user guiding the automation in places where human insight is needed.

The `prove_refinement` tactic defers to the user in two specific places. The first is in defining invariants for uses of the `mrec` recursive function combinator. The tool defers to the user to provide these invariants because inferring such invariants is undecidable. The second place where `prove_refinement` defers to the user is in proving non-refinement goals regarding first order data. The user can then apply other automated and/or manual proof techniques for the theories of the resulting proof goals.

The `prove_refinement` tactic is defined using a collection of syntax-directed inference rules for proving refinement goals. The tactic proves refinement goals by iteratively choosing and applying a rule that matches the current goal and then proceeding to prove the antecedents. The `prove_refinement` tactic implements this strategy using the Coq hint database mechanism, which is already a user-extensible mechanism for proof automation using syntax-directed rules.

We omit further implementation details both for space and because we do not claim the implementation of the `prove_refinement` tactic is novel or interesting. What is novel and interesting is that ITree specifications are designed in such a way that the straightforward implementation is able to achieve impressive results.

6.2 Evaluation

He et al. [9] discussed using Heapster to verify the interface of `mbox`, a key datastructure in the implementation of the Encapsulating Security Payload (ESP) protocol of IPSec. The `mbox` datastructure represents a data packet as a linked list of fixed length arrays. He et al. [9] type checked and extracted functional specifications for several functions that manipulate `mbox`. Using ITree specifications, we specified and verified the behavior of these functional specifications using our auto-active verification tool. These functions are nontrivial, combining loops, recursion, and pointer manipulations. We present the list of verified functions in Figure 13.

For each function, we include the function's name, a description of its behavior, the number of lines of C code in its definition, and the number of lines of Coq code required to verify it. Lines of code are, of course, a very coarse metric for judging the complexity of code and proofs. However, these metrics do demonstrate the viability of this verification approach, showing that the remaining proof burden after the automation is of a reasonable size. The primary advantage this approach has over others is that the system reduces the verification down to facts about first order data. In this case, the data is a variant of the `mbox` datastructure written in Coq.

7 Related Work

The most closely related work is the work on Dijkstra monads [1, 16, 27, 28]. Dijkstra monads are a framework for writing specifications over arbitrary monads. This framework is the basis for verifying programs with effects in $F^\star$ [26], a programming language specifically designed for verification. Dijkstra monads arise from the interaction of three structures, a *monad* `M`, a *specification monad* `W`, and an *effect observation* function `obs`. The monad `M` represents computations to be verified, while the specification monad `W` is a monad for writing specifications about those computations. The effect observation function `obs` is a monad homomorphism that embeds computations in `M` to the most precise specification in `W` that they satisfy. The specification monad is also equipped with a refinement relation that expresses when one specification implies or is contained in another. As an example, Dijkstra monads arose out of generalizing the notion of weakest precondition computations, by viewing the weakest precondition transformer of a computation as itself being a stateful computation from postconditions to preconditions. The mapping from a computation to its weakest precondition transformer is then a monad homomorphism from the computation monad to the weakest precondition monad.

ITree specifications in fact form a Dijkstra monad, where the type `itree_spec E R` acts as the specification monad and the corresponding ITree monad `itree E R` without logical quantifier events forms the computation monad. The effect observation homomorphism is then the natural embedding from the ITree type without quantifiers to the type with quantifiers. Most Dijkstra monads are specialized to act as either partial specification logics, which always accept any nonterminating computations, or total specification logics, which always reject any nonterminating computations. This means that most existing Dijkstra monads cannot reason about termination-sensitive properties like liveness. ITree specifications have the advantage of admitting specifications that accept particular divergent computations and not others. For example, an ITree specification could accept any computation that produces an infinite pattern of messages and responses from a server, and reject any computation that silently diverges.

A notable exception is the work of Silver and Zdancewic [25], who also provided a Dijkstra monad for ITrees. Much like ITree specifications it was capable of expressing specifications that allow for specifying infinite behavior. However, it did not provide reasoning principles for general recursion. The fact that ITree specifications represent specifications as syntax rather than semantics, as an ITree rather than some function relating ITrees to `Prop`, enabled us to write reasoning principles for general recursion and to build automation around the refinement rules.

A lot of work on verifying monadic computations has been based on notions of equational reasoning. This was in fact a key part of Moggi's original work [19]. Pitts [21] and Moggi [20] extend this approach be building general theories of an evaluation predicate for reasoning about return values of computations. This approach provides no explicit means to reason about the effects, however, and also has no direct way of handling non-termination in specifications such as the specifications needed for a server process. Plotkin and Pretnar [22] further extend this approach with a general-purpose logic for algebraic effects, allowing it to reason about the effects themselves and not just return values. This approach cannot handle general Hoare logic assertions, however, and although there is a high-level discussion about handling recursion, it is not clear how well it works for those sorts of specifications. Rauch et al. [23] extends monads with native exceptions and non-termination and provides a logic for these monads. Much like in our work, monads in Rauch et al. [23] can be annotated with assertions. However, it restricts the language of assertions, and does not provide assumptions, or general universal or existential quantification. It also handles only tail recursive programs, and not general, mutual recursion.

One particularly effective approach in the space of equational reasoning was that of Gibbons and Hinze [8]. This work showed how to use the specialized monad laws of each sort of effect in a computation to define rewrite rules for simplifying and reasoning about effectful computations, and then demonstrated that this approach is both straightforward to use and powerful enough to verify a number of small but interesting programs.

The ultimate goal of this work is to provide techniques for auto-active verification of imperative code. Therefore, it is natural to compare this work to semi-automated separation logic tools like VST-Floyd[2] and CFML[6]. We argue this approach has two major advantages over these related techniques. First, while VST-Floyd is specialized to C and CFML is specialized to Caml, ITree specifications can be used to specify any programs with an ITrees based semantics. When paired with Heapster techniques, ITree specifications can be used to specify a wide array of imperative, heap-manipulating languages with a memory-safe type system. In particular, the Heapster type system is closely related to the Rust type system, meaning these techniques should be adaptable to specify and verify Rust code. Second, the Heapster types are able to perform all the separation logic specific reasoning, freeing the verifier to focus on the underlying mathematical structures.

8 Conclusion

This paper introduces ITree specifications along with verified metatheory and proof automation for reasoning about them. ITree specifications are a specialization of ITrees with a general notion of specification refinement. Unlike previous work developing specifications for ITrees, this paper provides techniques for working with the general recursion operator provided by the ITrees library. Finally, this paper demonstrates the effectiveness of its techniques by applying them on a collection of realistic C functions.

References

1 Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martinez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.

2 Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, USA, 2014. ISBN 110704801X.

3 Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2019.

4 Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999.

5 Aaron R. Bradley. Sat-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011.

6 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 418–430, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308656. doi: 10.1145/2034773.2034828. URL `https://doi.org/10.1145/2034773.2034828`.

7 Andrey Chudnov, Nathan Collins, Byron Cook, Joey Dodds, Brian Huffman, Colm MacCárthaigh, Stephen Magill, Eric Mertens, Eric Mullen, Serdar Tasiran, Aaron Tomb, and Eddy Westbrook. Continuous formal verification of amazon s2n. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, 2018.

8 Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming (ICFP)*, 2011.

9 Paul He, Edwin Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Stefanescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. A type system for extracting functional specifications from memory-safe imperative programs. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2021.

10 Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022. doi: 10.1145/3547647. URL `https://doi.org/10.1145/3547647`.

11 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013. doi: 10.1145/2429069.2429093.

12 Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 256–269. ACM, 2016. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951943. URL `http://doi.acm.org/10.1145/2951913.2951943`.

13 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596. URL `http://doi.acm.org/10.1145/1629575.1629596`.

14 Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *J. Autom. Reason.*, 41(1):1–31, jul 2008. ISSN 0168-7433. doi: 10.1007/s10817-008-9099-0. URL `https://doi.org/10.1007/s10817-008-9099-0`.

15 Giuliano Losa and Mike Dodds. On the Formal Verification of the Stellar Consensus Protocol. In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, 2020.

16 Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi: 10.1145/3341708. URL `https://doi.org/10.1145/3341708`.

17 Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. Rusthorn: Chc-based verification for rust programs. In *Proceedings of the 29th European Symposium on Programming (ESOP)*, 2020.

18 Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. Rusthornbelt: A semantic foundation for functional verification of rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2022.

19 Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS)*, 1989.

20 Eugenio Moggi. A semantics for evaluation logic. *Fundamenta Informaticae*, 22(1), 1989.

21 Andrew M. Pitts. Evaluation logic. In *Proceedings of the IV Higher Order Workshop*, 1990.

22 Gordon Plotkin and Matija Pretnar. A logic for algebraic effects. In *Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2008.

23 Christoph Rauch, Sergey Goncharov, and Lutz Schröder. Generic hoare logic for order-enriched effects with exceptions. In Phillip James and Markus Roggenbach, editors, *Recent Trends in Algebraic Development Techniques*, pages 208–222, Cham, 2017. Springer International Publishing. ISBN 978-3-319-72044-9.

24 Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 77–87, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737964. URL `https://doi.org/10.1145/2737924.2737964`.

25 Lucas Silver and Steve Zdancewic. Dijkstra monads forever: Termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: 10.1145/3434307. URL `https://doi.org/10.1145/3434307`.

26 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 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*, pages 266–278, 2011. doi: 10.1145/2034773.2034811. URL `http://doi.acm.org/10.1145/2034773.2034811`.

27 Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 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, pages 387–398, 2013. doi: 10.1145/2491956.2491978. URL `http://doi.acm.org/10.1145/2491956.2491978`.

28 Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 256–270, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335492. doi: 10.1145/2837614.2837655. URL `https://doi.org/10.1145/2837614.2837655`.

29 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371119. URL `https://doi.org/10.1145/3371119`.

30 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL), January 2020. doi: 10.1145/3371119.