INTERACTION TREES AND FORMAL SPECIFICATIONS

Lucas Silver

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2023

Supervisor of Dissertation

Stephan Zdancewic, Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Stephanie Weirich, Professor of Computer and Information Science, Committee Chair
Benjamin C. Pierce, Professor of Computer and Information Science
Val Tannen, Professor of Computer and Information Science
Eddy Westbrook, Principal Researcher, Galois Inc.

ACKNOWLEDGEMENT

I would like to thank all of my colleagues and collaborators. I would particularly like to thank my fellow students in PL Club and my advisor, Steve.

ABSTRACT

INTERACTION TREES AND FORMAL SPECIFICATIONS

Lucas Silver

Stephan Zdancewic

Interaction Trees are a recently developed form of denotational semantics for effectful programs that is executable and compositional. This dissertation uses Interaction Trees to develop reusable, language-independent tools for different classes of specifications. First, it demonstrates how to apply the Dijkstra monads (Swamy et al., 2013; Maillard et al., 2019) approach to Interaction Trees. Second, it demonstrates how to analyze the information flow properties of Interaction Trees, enabling security analysis for any programs with Interaction Tree denotations. Finally, it presents the Interaction Tree Specification framework, a program logic for Interaction Trees that enables efficient, syntactic automated proofs of properties of Interaction Trees.

TABLE OF CONTENTS

CHAPTER 1

Introduction

1.1. Motivation

Formal verification of software is on the rise, combining a steady stream of theoretical advances from academic research with a growing interest in verification from industry. Notable academic examples include: the CompCert compiler (Leroy, 2009; Kästner et al., 2018), a formally verified, optimizing compiler from C to machine code; the VST project (Appel, 2011, 2014), a separation logic for reasoning about C programs; and the Iris logic (Jung et al., 2015), a language independent separation logic for higher-order, effectful programming. Notable examples from industry include: the sel4 verified microkernel (Klein et al., 2009); Amazon's verification of the s2n cryptography library (Chudnov et al., 2018); and the Heapster project (He et al., 2021) for reasoning about low-level heap manipulating programs.

Formal verification can provide intrinsically stronger guarantees of the correctness and safety of software than traditionally dominant methods like testing and code reviews (Appel et al., 2017). Even the most sophisticated testing tools can test only a finite amount of a program's possible inputs, while formal verification can give guarantees about all possible inputs. The utility of formal methods was also empirically demonstrated by the CSmith project. CSmith used randomized testing techniques to identify bugs in a collection of C compilers and failed to find any bugs in the verified components of CompCert (Yang et al., 2011). For all of these reasons, formal verification is a promising avenue of research.

One limitation of formal verification, however, is the lack of multilanguage verification tools. For example, VST (Appel, 2011) and CFML (Charguéraud, 2011) are both semi-automated separation logics, but are incompatible because VST is for C and CFML is for Caml. A major exception to this is Iris (Jung et al., 2015), a language independent separation logic for higher-order, effectful programming. The Iris tool chain supports defining new programming languages while automatically instantiating powerful logics for formal verification. However, Iris is a very heavy-duty tool, and much of its weight comes from its ability to reason about higher-order program features, like recursive, higher-order functions. This leaves a gap in the literature for simpler tools that deal with simpler, lower-order programming languages. This dissertation explores one path in the direction of addressing this gap.

1.2. Formal Specifications and Formal Semantics

Formal verification relies heavily on two key, related technologies, formal semantics and formal specifications. Formal semantics assign a *mathematically precise* meaning to programs. By formulating the meaning of programs in terms of mathematics, we can apply the tools of mathematical proofs to obtain high assurance guarantees about the behavior of programs. This can include guarantees about an individual program, e.g., ensuring it computes the correct result; guarantees across an entire programming language, e.g., ensuring well-typed programs have no undefined behavior; and guarantees across different programming languages, e.g., ensuring that compiled programs refine the behavior of their source programs.

Formal specifications provide a language to describe the behavior of a program or collection of programs. A classic example of a language of formal specifications is Hoare logic (Hoare, 1969). Hoare logic operates over Hoare triples. Hoare triples are tuples which contiain a program, a precondition, and a postcondition. A Hoare triple is valid if, given any initial state that satisfies the precondition, the program produces an output state that satisfies the postcondition. A closely related example is separation logic (Reynolds, 2002; O'Hearn, 2007; Brookes, 2007). A separation logic enriches the language of Hoare logic with the capability with the *separating conjunction* operator. The separating conjunction operator enables reasoning *separately* about disjoint sections of the heap. Separation logics have proven immensely useful for reasoning about memory safety and concurrency, among many other applications (Jung et al., 2015; Cao et al., 2018; O'Hearn, 2007).

Besides program correctness, another example specification is noninterference, a condition from the security literature. Suppose a program manipulates data with different levels of privilege, i.e., public and private. Also suppose that observers have no way of directly reading or manipulating data that they do not have access to. This assumption is a key component of the threat model for this security property. A noninterfering program respects these privilege levels, preventing observers from gaining information about data they lack the permission to access. Such programs prevent public observers from learning private information by manipulating and observing public information.

Each of the kinds of specification mentioned above constrains the behavior of a program, but leaves significant leeway for programmers to make different implementation choices. This allows the specifications to cover a wide range of solutions without compromising on correctness. Formal specifications typically come equipped with reasoning principles to help users verify the correctness of programs. In the case of program logics, these include inference rules involving the different syntactic constructs of the language. In the case of noninterference, this can include a type system that ensures that programs do not leak any private information. These reasoning principles are justified with respect to the formal semantics of the language.

1.3. Interaction Trees

Interaction Trees, or ITrees (Xia et al., 2020), are a new form of formal semantics for representing interactive, effectful, and potentially nonterminating computations. Most mechanized proof of properties of interactive, effectful, and potentially nonterminating computations rely either on operational semantics or on trace models (Focardi et al., 2002; Malecha et al., 2011; Gu et al., 2016). Such representations have been instrumental in mechanized proofs of programs in diverse settings. However, they each have significant drawbacks as well. Small-step operational semantics are noncompositional, and require auxiliary information (like program counters, evaluation contexts, and stateful stores) in addition to the syntax in order to specify their behavior. Both operational semantics and trace models rely on noncomputable predicates, rendering them unable to be run either for testing purposes or as a reference implementation.

ITrees, in contrast, are an *executable*, *compositional*, and *denotational* semantics for programming languages. This unique combination of features provides many advantages for program verification. Executability enables the formal semantics of a programming language to be tested, just like a compiler or an interpreter. This opens up the possibility of using all of the available tools for testing programs to check properties, reducing the possibility that a large amount of effort is spent trying to prove a property that is actually false. Compositionality gives us the ability to separate reasoning about different language effects. Traditional semantics tend to model all effects in an interconnected way, making it impossible to reason about the exception behavior of a language separately from its stateful behavior. Denotational semantics map programs to mathematical objects in the underlying metatheory. This allows us to use more of the tools of our metalogic. For example, loops are defined in terms of a Coq function, which is reusable across different program semantics, rather than in terms of inference rules in an inductive step relation specific to a particular language semantics This in turn allows proof engineers to reuse a single reasoning principle about this underlying loop function. They also admit simpler equational theories than operational semantics.

A growing number of projects seek to profit from these advantages by using ITrees. This includes Vellvm (Zakowski et al., 2021a), an ITrees based semantics for the LLVM programming language. The Vellvm project makes extensive use of the compositionality of effects in ITrees, separately defining several categories of effects required in LLVM. This ITrees based semantics automatically yields a reliable reference interpreter, an invaluable tool for testing production implementations. Vellvm has also been used to justify compiler optimizations, heightening trust in LLVM compiler passes. It also includes the Heapster project (He et al., 2021). Heapster provides a memory safe type system to LLVM code, along with a program that transforms well typed LLVM code into equivalent programs, known as *functional specifications*, written directly with the ITrees datastructure[1]. ITrees have also been used in the DeepSpec web server to formalize a server's specification (Koh et al., 2019), in the Conditional Contextual Refinement framework for reasoning about code with independently defined modules (Song et al., 2023), and in the DimSum framework for reasoning programs involving multiple languages (Sammler et al., 2023).

These successes provide good reasons to consider an ITrees semantics when developing the formal semantics of a programming language. Another reason to consider an ITrees semantics is the advantage of standardization. A problem that arises in the formalization of one language can lead to tools that directly apply to the formalization of another. If we create tools for reasoning about

[1] Heapster is discussed at length in Chapter 5.

formal specifications over ITrees, the tools can be imported to different languages as libraries of verified code. However, because ITrees are a new technology, relatively few such tools exists yet. **In this dissertation, I develop reusable, language-independent tools for different classes of specifications over programs with ITree semantics.**

1.4. Contributions

In particular, I introduce three additions to the ITrees literature that each investigate how we might define certain kinds of specifications with respect to ITrees semantics.

- In Chapter 3, I show how to apply an array of algebraic-effect-aware specification types to ITrees semantics using an extension of Dijkstra Monads (Maillard et al., 2019). The work in this chapter is formalized in the artifact provided in Silver and Zdancewic (2020).

- In Chapter 4, I develop an information flow aware bisimulation relation for ITrees, enabling the specification of information flow properties like noninterference on ITrees. The work in this chapter is formalized in the artifact provided in Silver et al. (2023a).

- In Chapter 5, I augment ITrees with universal and existential quantification operators, show how to use it as a language of specifications for ITrees, and demonstrate its effectiveness in verifying real C programs when used in concert with the Heapster tool (He et al., 2021). The work in this chapter is formalized in the artifact provided in Silver et al. (2023d).

The definitions and theorems in this dissertation have all been formalized in the Coq proof assistant.

1.5. Attribution

Most of the work presented in this dissertation was adapted from my previously published papers. I performed the majority of the technical work, particularly with respect to the Coq developments, and wrote the papers in collaboration with several coauthors. Chapter 3 is an adaptation of *Dijkstra Monads Forever: Termination-sensitive specifications for interaction trees* (Silver and Zdancewic, 2021), which I wrote with my advisor Steve Zdancewic. Chapter 4 is an adaptation of *Semantics for Noninterference with Interaction Trees* (Silver et al., 2023b), which I wrote with Paul He, Ethan

Cecchetti, Andrew Hirsch, and Steve Zdancewic. Chapter 5 is an adaptation of *Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations that Supports Auto-active Verification* (Silver et al., 2023c), which I wrote with Eddy Westbrook, Matthew Yaccavone and Ryan Scott. Both *Semantics for Noninterference with Interaction Trees* and *Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations that Supports Auto-active Verification* have been accepted for publication at ECOOP 2023. The work in each chapter is heavily reliant on Interaction Trees (Xia et al., 2020). In particular, Chapter 2 is primarily a repackaging of information from Xia et al. (2020), with the exception of Section 2.7 which repackages information originally produced for *Dijkstra Monads Forever* (Silver and Zdancewic, 2021) and *Interaction Tree Specifications.*

CHAPTER 2

Interaction Trees

2.1. Definition

Interaction Trees (ITrees) are a data structure for denotational semantics implemented as a coinductive variant of the free monad in Coq. The use of coinduction enables ITrees to represent possibly divergent computation. The monadic structure of ITrees provides a natural notion of sequential composition. And *free* monads provide an interface for flexibly representing various effects, as we will see below.

Intuitively, ITrees represent effectful programs as potentially infinite trees. Concretely, the ITree type is parameterized by a return type `R` and a type family `E` with sort `Type` → `Type`. The definition is presented in Figure 2.1[2]. The `Ret` constructor forms the leaves of these trees, which carry inhabitants of the `R` type. These leaves represent pure computations with no effects. Given any pure value `r : R`, `Ret r` represents the program that does nothing except return the value `r`.

The `Tau` constructor represents one step of silent internal computation inside an ITree. `Tau` nodes are key for representing programs that diverge without performing any other effect. An example of such a program is `while (true) do {skip}`. Because ITrees are defined as a coinductive type, an infinite number of `Tau` nodes can be chained together to form an ITree consisting only of silent internal steps of computation. This is implemented in the following code.

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

The `Vis` constructor forms the branching nodes of ITrees. Each `Vis` node is labelled with an *event* of type `E A`. Given an event `e : E A`, `A` is called the *answer type* of `e`. Events are inert values that are primarily used to represent algebraic effects (Plotkin and Pretnar, 2013; Bauer and Pretnar, 2015; Plotkin and Power, 2001). The answer type of an event is the type of the answer the corresponding effect would evaluate to if interpreted by an environment. For example, consider the type family

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

```
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 answer in A *)
```

Figure 2.1: Interaction Trees definition

implemented by the following code.

```
Inductive stateE : Type → Type :=
  | Get : stateE nat
  | Put : nat → stateE unit.
```

The `Get` event in `stateE` represents an access of the state cell. When the environment interprets this event, it will provide an answer in the form of a natural number. A `Put n` event in `stateE` represents a mutation to the state cell, replacing its current contents with `n`. When the environment interprets this event, it will provide an answer without any computational information, represented by a unit value `tt`. This unit value represents a signal from the environment indicating that the `Put` event has finished; it gives no further information about how it may have affected the environment . While events are often used to represent algebraic effects such as state, Section 2.6 demonstrates how to use them to define recursive computations in ITrees, and Chapter 5 demonstrates how to use them to define logical quantifiers.

The continuation, of type `A →  itree E R`, contained in the `Vis` node determines how the rest of the program executes after the environment interprets the event. This continuation defines a branch of the ITree for each element of the answer type `A`.

For a simple example, consider the following program.

```
Definition access : itree stateE nat :=
  Vis Get (fun n ⇒ Ret n).
```

The `access` ITree consists of a single `Vis` node with a `Get` event and defines each branch off of the node as a pure computation that returns the answer provided by the state access. It defines all of these branches with the single function `fun x ⇒  Ret x`. As another example, consider the following more complicated program.

```
Definition increment : itree stateE unit :=
   Vis Get (fun n ⇒ Vis (Put (1 + n)) (fun _ ⇒ Ret tt)).
```

The increment program accesses the current value of the state cell, places the successor of that value into the state cell, and returns the unit value to indicate termination.

2.2. Equivalence Up To Tau (eutt)

One of the major advantages of the ITrees datatype is the rich equational theory provided for it in Xia et al. (2020). The primary notion of equivalence used for ITrees is called eutt or *equivalence up to tau*. Xia et al. (2019) 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.

Consider the following ITree,

```
Definition increment_with_taus : itree stateE unit :=
   Tau (Vis Get (fun n ⇒ Vis (Put (1 + n)) (fun _ ⇒ Tau (Ret tt)))).
```

It has the same visible events and return values as increment, but has an extra Tau node at the head and right before the leaves. Because Tau nodes represent silent steps of computation, we want to equate the increment and increment_with_taus ITrees. The eutt relation is designed to contain this equation.

In Section 2.8, we provide semantics for a simple imperative language in terms of ITrees. In this semantics, the number of loop iterations affects the number of Tau nodes in the resulting ITree. This means we need to ignore finite numbers of Tau nodes in order to equate two programs with identical input/output behavior that differ in the number of loop iterations.

The eutt relation is parameterized by a relation RR over return values. In general, the relation RR is *heterogeneous*, relating values over distinct types R1 and R2. The eutt RR relation is heterogeneous in general as well, relating values 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 *homogeneous* relations, `RR : R → R → Prop`, that relate ITrees with the same return type. In particular, we are interested in `eutt eq` and denote this relation with the symbol $\approx$.

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

```
Inductive euttF {E R1 R2} (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 A (e : E A) (k1 : A → itree E R1) (k2 : A → itree E R2) :
    (∀ 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 {E R1 R2} (RR : R1 → R2 → Prop) : itree E R1 → itree E 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, `gfp`, of `euttF`[3]. Calls to the `sim` argument in the definition of `euttF` correspond to coinductive calls to `eutt`. Recursive calls to `euttF` correspond to inductive calls to `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 chained without an intervening call to a coinductive constructor. Specifically, only finitely many `eutt_TauL` and `eutt_TauR` steps, which remove a `Tau` from only one side, are allowed before one of the remaining rules is 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`

[3] In this document, all greatest fixpoints are defined using the `paco` library (Hur et al., 2013)

constructors. In particular, we can prove that spin is equivalent to itself, that Tau (Ret 0) is equivalent to Ret 0, that increment is equivalent to increment_with_taus, and that spin is *not* equivalent to Ret 0.

It is important to note that the eutt relation does not have any information about the semantics of the algebraic effects that an event represents. It reasons exclusively about the tree structure. For example consider the following two programs.

```
Definition access : itree stateE nat :=
  Vis Get (fun n ⇒ Ret n).

Definition access2 : itree stateE nat :=
  Vis Get (fun n1 ⇒ Vis Get (fun n2 ⇒ Ret n2)).
```

The first program is the same access example presented in Section 2.1. This program accesses the value in the state cell and returns it as output. The second program makes two accesses to the state cell and then returns the answer to the second access as the result. Intuitively, we may want to identify these programs because they compute the same return value and output state when given the same input state. However, they are *not* related by eutt. This is both because they have different numbers of events, and because the eutt relation lacks the knowledge that state accesses don't change the state. In Section 2.4, we solve this problem by providing a way to give semantics to the uninterpreted events in an ITree. With the proper interpretation, these two programs are equated.

2.3. Monad and Iteration Structure

ITrees form a monad. Monads are a mathematical structure for representing computations with a notion of sequential composition. For the purposes of this document, monads are defined with the type class presented in Figure 2.2. A monad is a type family, M : Type → Type, with corresponding ret and bind functions. An element of M A is a computation that returns a value of type A. The ret combinator of a monad wraps up a pure value a into a monadic computation. And the bind combinator sequentially composes two, potentially effectful, computations. That is, bind m k is the computation that consists of executing m : M A and feeding any result of type A into the continuation

```
Class Monad (M : Type → Type) :=
  {
    ret : ∀(A : Type), A → M A;
    bind : ∀(A B : Type), M A → (A → M B) → M B }.

Class MonadIter (M : Type → Type) :=
  {
    monad : Monad M;
    iter : ∀(A B : Type), (A → M (A + B)) → A → M B }.

Class MonadIterLaws (M : Type → Type) `{MonadIter M} :=
  {
    bind_ret_l : ∀a k, bind (ret a) k ≈ k a;
    bind_ret_r : ∀m, bind m ret ≈ m;
    bind_bind : ∀m k1 k2, bind (bind m k1) k2 ≈ bind m (∀ x ⇒ bind (k1 x) k2);
    iter_bind : ∀body a, iter body a ≈ bind (body a) (case (iter body) ret )
    ... }.
```

Figure 2.2: Monad and Iteration Typeclasses

`k : A → M B` to determine the rest of the computation.

Figure 2.2 also presents the monad iterator type class, `MonadIter`. Instances of `MonadIter`, in addition to implementing the functions required by `Monad`, must implement a generalization of a do-while loop called `iter`. The `iter` function extends do-while loops to include input and output values. The `iter` function defines a loop in terms of the a loop body, `body : A → M (A + B)`. The type `A` represents inputs to the loop as well as signals to continue running the loop. In the execution of the loop, continuation signals are fed into the loop body as input to compute the next iteration. The type `B` represents final outputs to the loop. Given an initial input, `init : A`, the result of `iter body init`, of type `M B`, is obtained by computing `body init`, and binding a continuation that terminates the loop if it is passed a `B` value, and reruns the loop if given an `A` value. We call a type family that satisfies the `MonadIter` typeclass, an *iterable monad.* ITrees are an example of an iterable monad.

Concrete implementations of these typeclasses for ITrees can be founded in Figure 2.3. The `ret` combinator is implemented by the `Ret` constructor. The `bind` combinator is implemented by grafting the continuation `k` onto each of the leaves of the ITree `m`. It is implemented as a cofixpoint in Coq. The `iter` combinator can be implemented in terms of `bind`, again using a cofixpoint.

ITrees satisfy a collection of equations, including the well-known monad laws, the less well-known

```
Definition ret_itree {E R} (r : R) : itree E R := Ret r.

CoFixpoint bind_itree {E A B} (m : itree E A) (k : A → itree E B) : itree E B :=
  match m with
  | Ret a ⇒ k a
  | Tau t ⇒ Tau (bind_itree t k)
  | Vis e kvis ⇒ Vis e (fun x ⇒ bind_itree (kvis x) k )
  end.

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

Class MonadIter {E} (itree E) := {|
  ret := ret_itree;
  bind := bind_itree;
  iter := iter_itree |}.
```

Figure 2.3: ITree Typeclass Instance Definitions

```
Definition interp_body {E M : Type → Type} `{MonadIter M} {R : Type}
           (handler : ∀A, E A → M A)
           (t : itree E R) : M (itree E R + R) :=
          match t with
          | Ret r   ⇒ ret (inr r)
          | Tau t   ⇒ ret (inl t)
          | Vis e k ⇒ bind (handler _ e) (fun a ⇒ ret (inl (k a)))
          end
Definition interp {E M : Type → Type} `{MonadIter M} {R : Type}
           (handler : ∀A, E A → M A) : itree E R → M R :=
  iter interp_body.
```

Figure 2.4: Interpretation Definition

iteration laws, and a collection of ITree specific equations regarding `Vis` and `Tau` nodes. For example, the iteration laws contain an equation, named `iter_bind`, which expresses that `iter` loops are equivalent to their unfoldings. A selected subset of these equations are presented in Figure 2.2. These equations, using the `eutt` relation as the notion of equivalence, form the primary interface for reasoning about semantics written with ITrees. Repeated rewrites using these equations allow us reason about the execution of programs by simplifying their denotations.

2.4. Interpretation

The inert event nodes in an ITree are typically used to represent algebraic effects. Recall the `stateE` event type family, presented again in Figure 2.5. ITrees with events in `stateE` represent simple

```
Inductive stateE : Type → Type :=
  | Get : stateE nat
  | Put : nat → stateE unit.

Definition stateITree (A : Type) : Type :=
  nat → itree voidE (nat * A).

Definition stateE_handler : ∀A, stateE A → stateITree A :=
  fun _ e ⇒
    match e with
    | Get ⇒ fun s ⇒ Ret (s,s)
    | Put s ⇒ fun _ ⇒ Ret (s, tt)
    end.

Definition interp_state : ∀A, itree stateE A → stateITree A) :=
  interp stateE_handler.
```

Figure 2.5: Example State Event Signature

stateful programs, which can read and write to a single state cell which contains a natural number value. However, as discussed at the end of Section 2.2, this representation is missing information about how stateful programs actually run.

In order to obtain the desired equational theory, we need to provide semantics for the inert events that maps the events to the algebraic effects that they represent. This mapping requires the representation of effectful computations to have an iterable monad structure. For example, figure 2.5 provides the `stateITree` type family which adapts the standard state monad for this purpose. The `stateItree A` type contains functions from natural numbers, the type of values in the state cell, to ITrees that return a value of `A` along with another natural number, the updated state cell. This type is an iterable monad represents possibly divergent stateful computations, and it can be assigned a lawful monad iterator structure based closely on the iterable monad structure of ITrees.

We assign semantics to an event signature using a *handler*. Given an event signature `E` and an iterable monad `M`, a handler is a parametric function from `E` to `M`, with type `∀ A, E A → M A`. Figure 2.5 provides the handler for `stateE`. The `stateE_handler` function maps `Get` events to stateful computations that return the current state as both the output state and the return value, modelling a state access. It maps `Put s` events to stateful computations that ignore their input state, return `s` as the new output state, and return the placeholder value `tt` as their output, modelling a state mutation.

ITrees support the definition of *interpreters*, functions from ITrees to computations modeled as an iterable monad. Given a handler, an interpreter traverses an ITree and use the handler to transform its inert events into effectful programs. That is, given a handler `h`, it transforms an ITree, `Vis e k` into the effectful computation that consists of `h e` sequentially composed with interpretation of `k` with the same handler. In order to create this interpreter, the type family of effectful computations needs three things: a way to represent pure computations to interpret `Ret` nodes; a way to sequentially compose effectful computations to insert handled events back into place; and a form of nonterminating computation in order to represent potentially infinite ITrees. These requirements correspond exactly to the constraints of the monad iterator typeclass which requires: a `ret` combinator for pure computations; a `bind` combinator for sequential composition; and an `iter` combinator loop.

Figure 2.4 presents the implementation of the `interp` function which maps handlers to interpreters. The `interp` function uses `iter` to create a loop of the `interp_body` function. Recall that `iter` loops rely on a continuation signal type as well as an output type. The `iter` loop used to define `interp` uses uninterpreted ITrees, `itree E R`, as the continuation signal. The loop body accomplishes one step of the tree traversal. If it encounters a `Ret r` node, it terminates the loop with `ret (inr r)`. If it encounters a `Tau t` node, it removes the `Tau` and returns the rest of the tree to be traversed. The `iter` combinator reruns the loop body on `inl` values, so this is accomplished by returning `ret (inl t)`. Finally, if it encounters a `Vis e k` node, it uses the handler to produce a computation `handler e`, and uses `bind` to sequentially compose the handled event with the continuation `k` which has been marked for further traversal. Intuitively, the intermediate values produced by `interp_body` can be thought of as computations which can return either return values or ITrees. Executing this loop with `iter` has the effect of traversing through all possible intermediate ITrees, and evaluating away their events.

With this machinery, we can revisit our simple state example. Recall the stateful computations,

```
Definition access : itree stateE nat :=
  Vis Get (fun n ⇒ Ret n).

Definition access2 : itree stateE nat :=
  Vis Get (fun n1 ⇒ Vis Get (fun n2 ⇒ Ret n2)).
```

```
Class ReSum (E1 E2 : Type → Type) :=
  resum : ∀A, E1 A → E2 A.

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

Definition trigger {E1 E2 A} `{E1 -< E2} (e : E1 A) : itree E2 A :=
  Vis (resum e) (fun x ⇒ Ret x).
```

Figure 2.6: ReSum Definition

Given the interpreter defined in Figure 2.5, the `Get` events will be given interpreted as actual state accesses. The resulting stateful computations take equal initial states to `eutt` result computations, which is an equivalence relation over stateful computations. Both of the resulting computations are equivalent to the following computation.

```
Definition access_interpreted : stateITree nat :=
    fun (n : nat) ⇒ Ret (n, n).
```

2.4.1. Subevents

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 2.6. An instance of `ReSum E1 E2`, written `E1 -< E2`, is a function that injects an element of `E1 A` into `E2 A`. It can be thought of as a kind of subevent typeclass. The `ReSum` typeclass allows for the definition of the `trigger` function in Figure 2.6. The `trigger` function takes an event `e1 : E1 A` and injects it into `itree E2 A` by injecting `e1` into `E2 A` and placing that in a `Vis` node.

2.5. Presenting Mixed Inductive CoInductive Relations

In order to reason about ITrees, we will need to introduce several other relations that mix inductive and coinductive reasoning priciples similarly to `eutt`. These relations are defined with a large number of different constructors which can be cumbersome present and reason about. For clarity, these definitions will be presented as a series of inference rules. For instance, figure 2.7 presents the definition of `eutt` in this format. As is standard for inference rules, the propositions above the line are assumptions, the proposition below the line is the conclusion, and all free variables are assumed to be universally quantified. Each inference rule corresponds to one of the constructors used to

$$[\textsc{EuttRet}]\ \frac{\texttt{RR r1 r2}}{\texttt{eutt RR (ret }r1\texttt{) (ret }r2\texttt{)}}$$

$$[\textsc{EuttTau}]\ \begin{array}{c}\texttt{eutt RR t1 t2}\\ \hline\hline \texttt{eutt RR (Tau t1) (Tau t2)}\end{array}$$

$$[\textsc{EuttVis}]\ \begin{array}{c}\forall\texttt{a, eutt RR (k1 a) (k2 a)}\\ \hline\hline \texttt{eutt RR (Vis e k1) (Vis e k2)}\end{array}$$

$$[\textsc{EuttTauL}]\ \frac{\texttt{eutt RR t1 t2}}{\texttt{eutt RR (Tau t1) t2}}$$

$$[\textsc{EuttTauR}]\ \frac{\texttt{eutt RR t1 t2}}{\texttt{eutt RR t1 (Tau t2)}}$$

Figure 2.7: Inference Rules for `eutt`

define `euttF`. Inference rules with a *single line* correspond to constructors that use only inductive self reference. Inference rules with a *double line* correspond to constructors that use only coinductive self reference. These coinductive inference rules perform the same tasks as references to the `sim` argument in the definition of `euttF`.

The primary weakness of presenting inductive/coinductive relations this way is that the implicit quantification of parameters in these rules removes an opportunity to explicitly specify the types of these parameters. We make up for this shortcoming by specifying the types in the declaration of the definition. For example, observe the following definition of `eutt`.

Definition 1 (Equivalence up to tau (`eutt`)). *Given:*

- *an event signature `E`;*
- *return types `R1` and `R2`;*
- *and a return relation over `R1` and `R2`, `RR`,*

equivalence up to taus with `RR`, a relation between `itree E R1` and `itree E R2`, is defined with the inference rules presented in Figure 2.7. We write this relation as `eutt RR t1 t2`.

Going forward in this document, relations defined with a mixture of induction and coinduction will be presented in this style.

```
CoFixpoint interp_mrec {D E R} (bodies : ∀A, D A → itree (D +' E) A) (t : itree (D +' E) R) :
    itree E R :=
  match t with
  | Ret r ⇒ Ret r
  | Tau t ⇒ Tau (interp_mrec bodies t)
  | Vis (inl1 d) k ⇒ Tau (interp_mrec bodies (bind (bodies _ d) k))
  | Vis (inr1 e) k ⇒ Vis e (fun x ⇒ interp_mrec bodies (k x))
  end.

Definition mrec {D E A} (bodies : ∀A, D A → itree (D +' E) A) (d : D A) :=
  interp_mrec bodies (trigger d).
```

Figure 2.8: `mrec` Definition

```
Inductive evenoddE : Type → Type :=
  | even (n : nat) : evenoddE bool
  | odd  (n : nat) : evenoddE bool.

Definition evenodd_body : ∀A, evenoddE A → itree (evenoddE +' E) A :=
  fun _ eo ⇒
    match eo with
    | even n ⇒ if n =? 0
              then Ret true
              else trigger (odd (n -1))
    | odd n ⇒ if n =? 0
             then Ret false
             else trigger (even (n -1))
    end.

Definition evenodd : evenoddE bool → itree E bool :=
  mrec evenodd_body.
```

Figure 2.9: `evenodd` Example

2.6. Mutual Recursion

The ITrees library also provides a mutual recursion operator, `mrec`. The `mrec` operator works by using events as a form of syntax indicating a recursive function call site. The `mrec` operator defines recursive computation in terms of one level of unfolding of the recursive calls. The definition of `mrec` is presented in Figure 2.8. It utilizes an event type family `D` which represents recursive function calls. An event `d : D A` represents a recursive function call that returns a value of type `A`. The event `d` packages together the choice of the function being called with the arguments being supplied to that function. The `bodies` function assigns every recursive function call event, `d : D A`, a corresponding ITree in `itree (D +' E) A`. This ITree represents the evaluation of the recursive function call in terms of further inert `D` events, acting as *syntactic* recursive calls. The `interp_mrec`, defined in Figure 2.8, function takes an ITree with syntactic recursive calls, replaces each call `d : D A` with the unfolded function body `bodies d`, and corecursively repeats this process with the resulting ITree. Given this function, `mrec` is defined by applying `interp_mrec` to a tree that consists of triggering a single event, which represents the initial function call.

For a concrete example, observe the definition of `evenodd` in Figure 2.9. This mutually defines both an `even` and `odd` function. To accomplish this, it first defines `evenoddE`, an event type that packages together the names of the function calls, `even` and `odd`, and their arguments, a natural number in each case. Both the `even` and `odd` events require a natural number argument. Both `even` and `odd` events have the type `evenoddE nat`, which means they both have natural numbers as their response types. This shared response type signifies that both functions return a boolean value. Then we must define the bodies of these functions with `evenodd_body`. The `evenodd_body` function first pattern matches on its argument to determine which function is being called. Given an element of `evenoddE` it returns an element of `itree (evenoddE +' voidE) bool`. This allows the returned ITree to contain events that represent more recursive calls. We compute `even` by computing triggering an `odd (n - 1)` event and compute `odd` by triggering an `even (n - 1)` event. Both functions have a trivial base case at 0. Finally, we apply the `mrec` function which intuitively takes an initial function call event, and keeps unfolding it by applying `evenodd_body` and unfolding any function call events `evenodd_body` produces.

2.7. Heterogeneous Equivalence Up To Taus (rutt)

While `eutt` is a very useful relation, its constraint to only relate ITrees with precisely equal events is too restrictive for the purposes of some work in this dissertation. Suppose we want to relate ITrees that have different event signatures. This is essential for developing a trace model of ITrees, presented in Chapter 3, as well as reasoning about mutually recursive functions. In this section, we introduce the `rutt` relation, which generalizes the `eutt` relation with the capability to choose what pairs of events to relate at the head of `Vis` nodes, as well as the ability to constrain the possible result types.

First, we will briefly discuss how we choose events and response types for events. Figure 2.10 presents

```
Definition PreRel (E1 E2 : Type → Type) := ∀A B, E1 A → E2 B → Prop.

Definition PostRel (E1 E2 : Type → Type) := ∀A B, E1 A → E2 B → A → B → Prop.

Inductive SumPreRel {D1 D2 E1 E2} (RPre1 : PreRel D1 E1) (RPre2 : PreRel D2 E2) : PreRel (D1 +' D2)
     (E1 +' E2) :=
  | sumprerel_inl1 A B (d : D1 A) (e : E1 B) : RPre1 A B d e →
                 SumPreRel RPre1 RPre2 (inl1 d) (inl1 e)
  | sumprerel_inr1 A B (d : D2 A) (e : E2 B) : RPre2 A B d e →
                 SumPreRel RPre1 RPre2 (inr1 d) (inr1 e).
Notation "RPre1 '⊕'' RPre2" := (SumPreRel RPre1 RPre2) (at level 10).

Inductive SumPostRel {D1 D2 E1 E2} (RPost1 : PostRel D1 E1) (RPost2 : PostRel D2 E2) : PostRel (D1
    +' D2) (E1 +' E2) :=
  | sumpostrel_inl1 A B (d : D1 A) (e : E1 B) (a : A) (b : B) : RPost1 A B d e a b →
                  SumPostRel RPost1 RPost2 (inl1 d) (inl1 e) a b
  | sumpostrel_inr1 A B (d : D2 A) (e : E2 B) (a : A) (b : B) : RPost2 A B d e a b →
                  SumPostRel RPost1 RPost2 (inr1 d) (inr1 e) a b.
Notation "RPost1 '⊕''' RPost2" := (SumPostRel RPost1 RPost2) (at level 10).
```

Figure 2.10: Heterogeneous event relations

the definitions of both `PreRel` and `PostRel`. Elements of `PreRel` can be thought of as relations over event types, quantifying over all possible response types. The `rutt` relation uses `PreRel` elements as a kind of precondition for relating events. Only events that satisfy a particular precondition are allowed to progress in an `rutt` simulation proof. Elements of `PostRel` can be thought of as functions that map events to relations over the response type. The `rutt` relation uses the result of this `PostRel` function applied to the events as a postcondition that can be assumed to hold on any responses to events that satisfy the precondition. Below these definitions, Figure 2.10 presents variants of the sum relation for both `PreRel` and `PostRel`.

Definition 2 (HeteRogeneous equivalence up to taus (`rutt`)). *Given:*

- *event signatures* `E1` *and* `E2`*; return types* `R1` *and* `R2`*;*
- *a precondition relation over* `E1` *and* `E2`*,* `RPre`*;*
- *a postcondition relation over* `E1` *and* `E2`*,* `RPost`*; and*
- *a return relation over* `R1` *and* `R2`*,* `RR`*,*

heterogeneous equivalence up to `RPre`*,* `RPost` *and* `RR`*, a relation between* `itree E1 R1` *and* `itree`

$$[\textsc{RuttRet}]\ \frac{\texttt{RR r1 r2}}{\texttt{RPre};\texttt{RPost} \vdash (\texttt{ret r1}) \approx_{\texttt{RR}} (\texttt{ret r2})}$$

$$[\textsc{RuttTau}]\ \frac{\texttt{RPre};\texttt{RPost} \vdash \texttt{t1} \approx_{\texttt{RR}} \texttt{t2}}{\texttt{RPre};\texttt{RPost} \vdash (\texttt{Tau t1}) \approx_{\texttt{RR}} (\texttt{Tau t2})} \qquad [\textsc{RuttTauL}]\ \frac{\texttt{RPre};\texttt{RPost} \vdash \texttt{t1} \approx_{\texttt{RR}} \texttt{t2}}{\texttt{RPre};\texttt{RPost} \vdash (\texttt{Tau t1}) \approx_{\texttt{RR}} \texttt{t2}}$$

$$[\textsc{RuttTauR}]\ \frac{\texttt{RPre};\texttt{RPost} \vdash \texttt{t1} \approx_{\texttt{RR}} \texttt{t2}}{\texttt{RPre};\texttt{RPost} \vdash \texttt{t1} \approx_{\texttt{RR}} (\texttt{Tau t2})}$$

$$[\textsc{RuttVis}]\ \frac{\texttt{RPre e1 e2} \qquad \forall\ \texttt{a b},\ \texttt{RPost e1 e2 a b} \rightarrow \texttt{RPre};\texttt{RPost} \vdash (\texttt{k1 a}) \approx_{\texttt{RR}} (\texttt{k2 b})}{\texttt{RPre};\texttt{RPost} \vdash \texttt{Vis e1 k1} \approx_{\texttt{RR}} \texttt{Vis e2 k2}}$$

Figure 2.11: rutt Definition

E2 R2, is defined with the inference rules presented in Figure 2.11. We write this relation as $\mathit{RPre}; \mathit{RPost} \vdash t1 \approx_{RR} t2$.

The rutt relation is defined very similarly to eutt, with the primary difference being the RUTTVIS rule. Intuitively, this rule says that if two events are in the precondition, we can assume that their evaluated answers satisfy the postcondition. Concretely, when relating two ITrees, Vis e1 k1 and Vis e2 k2, rutt first requires that the precondition, RPre, holds on e1 and e2. It further requires that given any possible responses, a and b, that satisfy RPost A B e1 e2, the continuations k1 a and k2 b must be related as well.

Most eutt theorems, including ones regarding monad iterator combinators and transitivity, can be generalized to rutt theorems. One use of rutt is to provide a good reasoning principle for mrec.

Theorem 1 (Mutual Recursion Respects Heterogeneous Equivalence)**.** *If recursive call events init1 and init2 are contained in the precondition invariant RPreInv, and given any recursive call events, d1 and d2, contained in RPreInv,* $RPreInv \oplus' RPre; RPostInv \oplus'' RPost \vdash bodies1\ d1 \approx_{RPd} bodies2\ d2$ *where RPd = RPostInv d1 d2, then*

$RPre; RPost \vdash mrec\_spec\ bodies1\ init1 \approx_{RPi} mrec\_spec\ bodies2\ init2$

where RPi = RPostInv init1 init2.

This theorem reduces proving an rutt bisimulation between two ITrees defined with mrec to proving an rutt bisimulation between the function bodies. This eliminates a layer of recursive reasoning, potentially encapsulating a coinductive proof. The bisimulation over the function bodies contains extra event pre- and postconditions. The event relations over the base event signature E, RPre and RPost, are present in both the hypothesis and conclusion. The event relations over the recursive call event signature D, RPreInv and RPostInv, are present only in the hypothesis. These relations enforce pre- and postconditions on recursive calls. Intuitively, they enforce that each recursive call on the left must be matched by a recursive call on the right, such that the pair satisfy the precondition, and enable the assumption that these recursive calls return results that satisfy the postcondition.

2.8. Example Language (Imp)

This section presents the denotational semantics for a simple stateful language using ITrees. This semantics serves as an example of a typical ITrees semantics. A typical path from a language syntax to an ITrees semantics consists of: a denotation function from programs to ITrees over an event type family, E1, with constructors for every effect in the language; and an interpreter from that ITree into a target monad that is a stack of effect monad transformers applied to itree E2 for some other event type family, E2. This second event type family, E2, contains constructors for effects that are represented well by uninterpreted events. Common choices for E2 include voidE, the empty event type family, and IOE, the event type family which represents input and output from the user.

Figure 2.12 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

$$\begin{array}{llcl} \text{Expressions} & e & ::= & x \mid n \mid e+e \mid e-e \mid e*e \\ \text{Commands} & c & ::= & \mathsf{skip} \mid x := e \mid c_1\,;c_2 \mid \mathsf{while}\ (e)\ \mathsf{do}\ \{c\} \\ & & \mid & \mathsf{if}\ (e)\ \mathsf{then}\ \{c_1\}\ \mathsf{else}\ \{c_2\} \end{array}$$

```
(* 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.
```

$$\begin{aligned} [\![\cdot]\!] &: \text{Command} \to \texttt{itree impE unit} \\ [\![\mathsf{skip}]\!] &= \texttt{ret tt} \\ [\![x := e]\!] &= v \leftarrow [\![e]\!]\,;;\ \texttt{trigger (SetE x v)} \\ [\![c_1\,;c_2]\!] &= [\![c_1]\!]\,;;\,[\![c_2]\!] \\ [\![\mathsf{while}\ (e)\ \mathsf{do}\ \{c\}]\!] &= \texttt{while}(v \leftarrow [\![e]\!]\,;; \\ &\qquad \mathsf{if}\ ([\![e]\!])\ \mathsf{then}\ \{[\![c_1]\!]\,;;\texttt{ret (inl tt)}\}\ \mathsf{else}\ \{\texttt{ret (inr tt)}\}) \\ [\![\mathsf{if}\ (e)\ \mathsf{then}\ \{c_1\}\ \mathsf{else}\ \{c_2\}]\!] &= \mathsf{if}\ ([\![e]\!])\ \mathsf{then}\ \{[\![c_1]\!]\}\ \mathsf{else}\ \{[\![c_2]\!]\} \end{aligned}$$

```
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.
```

Figure 2.12: IMP denotational semantics (excerpt).

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. The semantics of while is defined using while, which is a simple wrapper around the ITree's iter combinator.

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 → itree E (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.

CHAPTER 3

Dijkstra Monads Forever

This chapter was previously published as 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. I was the primary author and did most of the research.

3.1. Introduction

Chapter 1 discussed a variety of frameworks for formal verification. 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 Carbonneaux et al. (2017)'s work on verified resource analysis.) 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) because it relies on a deep embedding of a language. Such representation choices matter in practice.

This chapter 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 and, therefore, lacked nonterminating programs. In contrast, the monads we investigate in this chapter 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 chapter. 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;
  }.
```

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 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 a specification?—and for establishing metatheoretic properties—like soundness—of the reasoning principles. This is a second significant challenge that we address in this chapter 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 chapter makes the following contributions.

- In Section 3.3 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.

- Building on `DelaySpec`, Section 3.4 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 3.5 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 3.6 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.

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 aspects of that work as needed throughout the paper. Before diving into the details of `DelaySpec` and `TraceSpec`, we first give some background about Dijkstra monads (Section 3.2).

3.2. Dijkstra Monads

Dijkstra monads (Swamy et al., 2013; Maillard et al., 2019) 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:

- A monad M, called the computational monad;
- A monad W equipped with an ordering relation, called the specification monad;
- And a function,Θ, from M to W, of type $\forall A.\ M\ A \to W\ A$, called the effect observation.

The computational monad, `M`, is the type of programs which the specifications are reasoning about. The specification monad, `W`, is the type of the specifications, and the order models specification refinement. The effect observation is a map from programs to the most precise specification that they satisfy. The most precise specification is defined as the least specification according to the refinement ordering. Given a specification monad, W, and two specification w_1 and w_2 in $W\ A$, we write that (w_1, w_2) is in the ordering relation as $W \vdash w_1 \leq w_2$. And if $W \vdash w_1 \leq w_2$, then we say that w_1 refines w_2.

Section 3.2.1 presents and motivates the restrictions we place on valid specification monads and effect observations. But the currently presented material is enough to define Dijkstra monads. Intuitively, Dijkstra monads use the provided notions of specifications and mappings from computations to specifications to define when a computation satisfies a specification.

Definition 3 (Dijkstra monad)**.** *Given a computation monad, M, a specification monad, W, an effect observation from M to W, Θ, and a specification $w : W\ A$, the corresponding Dijkstra monad is the set of computations $m : M\ A$ such that $W \vdash \Theta\ m \leq w$. We write that m is in the Dijkstra monad defined by M, W, Θ, and w as $\Theta \vdash m \in w$.*

For a concrete example, we can provide a Dijkstra monad for pure computations. The computation monad is the identity monad, which maps any type A to itself.

To first approximation, the specification monad is the type family:

$$\texttt{IDSpec}\ A \ := \ (A \to \mathbb{P}) \to \mathbb{P}$$

The actual type family has further restrictions, namely monoticity, which are discussed in Section 3.3. The `IDSpec` A type is the type of sets of sets over A. It also is the type of continuations into propositions. The simplest elements of `IDSpec` are $\lambda p.\top$ and $\lambda p.\bot$, respectively the total and empty sets. Another simple example, in this case of `IDSpec` $\mathbb{N}$, is $\lambda p.\texttt{even} \subseteq p$. This is the set of all supersets of the set of even natural numbers. We present the monadic structure and specification refinement orders in Section 3.2.2 after introducing the full definitions for specification monads and effect observations in Section 3.2.1.

The effect observation from the identity monad to the ID specification is the function:

$$\Theta_{ID}\ a := \lambda p.(a \in p)$$

The effect observation maps a computation, a, to the set of sets that contain a.

3.2.1. Specification Monads and Effect Observations

A specification monad consists a type family W, which contains the actual specifications, and a notion of refinement over those specifications. Formally, refinement is modelled with a family of partial orders, $R \subseteq W\ A \times W\ A$ for each type A. This family of partial orders induces a family of equivalence relations.

Definition 4. *Given a type family W, two elements of $W\ A$, w_1 and w_2, are considered equivalent if $W \vdash w_1 \leq w_2$ and $W \vdash w_2 \leq w_1$. In this case we write $W \vdash w_1 \cong w_2$.*

For W to be a valid specification relation, it must provide definitions for `ret` and `bind` that respect the monad laws using this induced equivalence relation.

Furthermore, the refinement relation needs to monotonic with respect to the `bind` operator.

Definition 5 (Monotonicity with respect to `bind`). *Given a monad W with a family of partial orders $R_X \subseteq W\ X \times W\ X$, R_X is said to be monotonic with respect to* `bind` *if the following implication holds. Given any monad elements, w_1, w_2 in $W\ A$, and continuations, k_1, k_2 in $A \to W\ B$, such that $(w_1, w_2) \in R_A$ and given any a in A, $(k_1\ a, k_2\ a) \in R_B$, then $(\texttt{bind}\ w_1\ k_1, \texttt{bind}\ w_2\ k_2) \in R_B$.*

Stated informally, the refinement relation is monotonic with respect to `bind`, if we can prove refinement of sequentially composed specifications by separately comparing the head and tail specifications. This requirement ensures that we can build proofs of specification refinement out of refinements of smaller specifications. It plays a key role in ensuring that a generalization of the Hoare sequencing rule holds.

Definition 6 (Specification monad). *A type family W along with a family of partial orders $R_A \subseteq W\ A \times W\ A$ form a specification monad if W forms a monad according to the family of equivalence relations induced by R and if R is monotonic with respect to the corresponding* `bind` *function.*

An effect observation between a computational monad M and a specification monad W is a function $\Theta : \forall A, M\ A \to W\ A$. This function is required to be a *monad morphism*, meaning that it maps `ret` and `bind` in the computational monad to `ret` and `bind` in the specification monad. This forces the effect observation to preserve a relation between sequential composition of computations and sequential composition of specifications. In concert with the requirement that the refinement relation respects `bind`, this allows proofs that programs satisfy specifications to break down cleanly with sequential composition.

Definition 7 (Effect observation). *Given a monad M and a specification monad W, a function $\Theta : \forall A, M\ A \to W\ A$ is an effect observation if the following equations hold:*

- *for any a in A, $W \vdash \Theta\ (\texttt{ret}\ a) \cong \texttt{ret}\ a$;*
- *for any m in $M\ A$ and k in $A \to M\ B$, $W \vdash \Theta(\texttt{bind}\ m\ k) \cong \texttt{bind}\ (\Theta\ m)\ (\Theta \circ k)$.*

3.2.2. Base Specification Monads

Many commonly used monads can be expressed in terms of a stack of monad transformers. These monads include computations with multiple kinds of effects. For example, computations with both state and exceptions can be represented as the state transformer applied to the exception transformer applied to the identity monad. Maillard *et. al.* [2019] presents a general technique for producing specification monads for such computation monads. It relies on identifying an appropriate *base specification monad*, a specification monad for the identity monad, and applying the same sequence of monad transformers used to produce the computation monad to the base specification monad.

The primary base specification monad used in that work is the previously introduced `IDSpec` type family:

$$\texttt{IDSpec}\ A \ := \ (A \to \mathbb{P}) \to \mathbb{P}$$

As `IDSpec` is a monad, we need to provide definitions for `ret` and `bind`.

$$\texttt{ret}\ a := \lambda p.(a \in p)$$

$$\texttt{bind}\ w\ k := \lambda p.((\lambda a.(p \in k\ a)) \in w)$$

The `ret` a specification is the set of all sets that contain the element a. And the `bind` w k specification builds a set of sets over B using w : `IDSpec` A and $k : A \to$ `IDSpec` B. This set contains all predicates p such that w accepts the set of elements of $a : A$ where k a accepts p. This definition is far more complicated than that of `ret`, but we can be confident it is correct because it satisfies the constraints of specifications monads.

The effect observation from the identity monad to the ID specification type is the same as `ret`.

$$\Theta_{ID}\ a := \lambda p.(a \in p)$$

The refinement relation is defined as

$$\texttt{IDSpec} \vdash w_1 \leq w_2 := \forall p.(w_2 \subseteq w_1)$$

This definition makes sense when viewed in context of proving a computation satisfies a specification. For example, consider the proposition that the computation 2 satisfies the specification which contains all supersets of the set of even natural numbers, $\lambda p.\texttt{even} \subseteq p$. The following logical propositions are all logically equivalent, and this equivalence can be justified simply by unfolding definitions.

$$\begin{aligned} \Theta_{ID} \vdash 2 \in (\lambda p.(\texttt{even} \subseteq p)) \\ \texttt{IDSpec} \vdash \Theta_{ID}\ 2 \leq (\lambda p.(\texttt{even} \subseteq p)) \\ (\lambda p.(\texttt{even} \subseteq p)) \subseteq (\Theta_{ID}\ 2) \\ (\texttt{even} \subseteq p) \rightarrow (2 \in p) \end{aligned}$$

This demonstrates that the proposition $\Theta_{ID} \vdash 2 \in \lambda p.\texttt{even} \subseteq p$ is equivalent to the proposition that 2 is contained in every superset of the even natural numbers. This is obviously true because 2 is even.

Performing the same simplification for an abstract computation and specification can further illustrate how these definitions work together. Suppose there is a computation m and a specification w, and you want to prove that m satisfies w. The following equations are justified purely by unfolding definitions.

$$\begin{aligned} \Theta_{ID} \vdash m \in w \\ \texttt{IDSpec} \vdash \Theta_{ID}\ m \leq w \\ w \subseteq (\Theta_{ID}\ m) \\ \forall p.(p \in w \rightarrow m \in p) \end{aligned}$$

The proposition that m satisfies the specification w reduces to the proposition m is contained in all predicates contained in w. In order words, w contains no properties that exclude m.

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

$$\texttt{StateSpec S A} := S \to ((S \times A) \to \mathbb{P}) \to \mathbb{P}$$

This is the type of functions from states into `IDSpec`'s over state paired with the return type. Just as the instances of `ret` and `bind` can be automatically generated, we automatically generate a refinement relation and an effect observation from those defined with `IDSpec`.

$$\texttt{StateSpec}\ S \vdash w_1 \leq w_2 := \forall s, \texttt{IDSpec} \vdash w_1\ s \leq w_2\ s$$

$$\Theta_S\ m\ s := (m\ s) \in \Theta_{ID}$$

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.

3.3. Delay Specification Monad

Maillard *et. al.* [2019] uses the identity monad as the most basic form of computation. In a strongly normalizing base logic like Coq, all elements of the identity monad represent terminating computations. Representing possibly divergent computations requires a monad based on coinductive types. This section presents the `Delay` monad and uses that to model possibly divergent computations. This is sufficient for specifying programs with either no effects other than nontermination, or that can be modelled by a stack of monad transformers applied to the `Delay` monad. Section 3.6 provides tools to deal with programs with external, uninterpreted events like IO or specific system calls.

3.3.1. Computational and Specification Monad Definitions

The computational monad is implemented as a form of ITrees with no visible event nodes. This is the `Delay` monad, `itree voidE`.

To a first approximation, the specification monad is the type of sets of sets over `Delay` A,

$$(\texttt{Delay}\ A \to \mathbb{P}) \to \mathbb{P}$$

This is nearly identical to the `IDSpec` monad presented in Section 3.2, with `Delay` A replacing A. This is also the type of backwards predicate transformers over the `Delay` monad. The full, possibly divergent computation is the output, the set over those computations is the output predicate, and there is no notion of input so the output predicate is mapped to a proposition.

The actual specification monad type introduces two extra constraints. First, specifications only contain predicates that respect the `eutt` relation.

Definition 8 (Respecting `eutt`). *A predicate, p, of type* `Delay` $A \to \mathbb{P}$ *respects* `eutt` *if, given two elements of* `Delay` A*,* t_1 *and* t_2*, that are contained in the relation* `eutt eq` *then* p *contains* t_1 *if and only if* p *contains* t_2*.*

This restriction rules out predicates that distinguish between trees that only differ in the number of `Tau` nodes, enforcing the idea `Tau` is a *silent* step of computation. This restriction does not rule out predicates that distinguish between divergent computations, represented as `spin`, and convergent computations. To be explicit, this constraint produces the type

$$\{p : (\texttt{Delay}\ A) \to \mathbb{P} \mid p \text{ respects } \texttt{eutt}\} \to \mathbb{P}$$

. This kind of set comprehension can be implemented in Coq as a sigma type. For simplicity, we apply set operations like element containment and subset directly to these sigma types.

Second, the specifications themselves must be *monotonic*.

Definition 9 (Monotonic specification)**.** *A set of* **`eutt`** *respecting sets w is considered to be monotonic if given any to predicates p_1 and p_2 in $A \to \mathbb{P}$, if $p_1 \subseteq p_2$ and p_1 is contained in w then p_2 is also contained in w.*

This constraint is key in proving that the refinement order is monotonic with respect to `bind`. Adding this constraint further complicates the formal type in the following way

$$\{w : \{p : (\texttt{Delay}\ A) \to \mathbb{P} \mid p \text{ respects } \texttt{eutt}\} \to \mathbb{P} \mid w \text{ is monotonic}\}$$

.

Definition 10 (`DelaySpec`)**.** *The* ***`DelaySpec`*** *type family contains the monotonic sets of* **`eutt`** *respecting sets.*

For the remainder of the discussion of `DelaySpec`, we will elide these finer details. It suffices to know that every predicate and specification discussed follows these constraints.

3.3.2. Monad Structure

The following code defines `ret` and `bind` in the `DelaySpec` monad.

$$\begin{aligned} \texttt{ret}\ a &:= \lambda p.(\texttt{ret}\ a) \in p \\ \texttt{bind}\ w\ k &:= \lambda p.(\lambda t.(\exists a.\texttt{voidE} \vdash t \approx_{=} \texttt{ret}\ a \wedge (p \in k\ a)) \vee ((t \in \texttt{diverges}) \wedge t \in p)) \in w \end{aligned}$$

Both definitions are similar to the definitions provided for `IDSpec`. The implementation of `ret` a contains all predicates that accept computations that terminate with the value a. The implementation of `bind` also closely mirrors the implementation for `IDSpec`. Just like before, the `bind` w k specification tests a predicate over `Delay` B, p, by constructing a new predicate over `Delay` A and testing its inclusion in w. This new predicate is the disjunction of two smaller predicates. The left half handles convergent trees and corresponds closely with the original predicate in the `bind` definition from `IDSpec`. If t converges to a value a, then it tests whether p is contained in the specification k a. The right half handles diverge trees. If t diverges, we test if p contains the unique divergent tree `spin`.

This can be thought of as a safe cast of t from `Delay` A to `Delay` B. This is allowed because divergent computations have no return value, and thus the return type does not constrain their behavior. When given this monad structure, taking set equality as the equivalence relation, `DelaySpec` satisfies the monad laws.

3.3.3. Specification Order

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

$$\texttt{DelaySpec} \vdash w_1 \leq w_2 := w_2 \subseteq w_1$$

The direction of the inequality makes sense for the same reason the direction in the `IDSpec` order makes sense, as explained in Section 3.2. This refinement order satisfies the monotonicity constraint for specification monads.

3.3.4. 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.

$$\Theta_D \ t := \lambda p. t \in p$$

3.3.5. 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

$$\texttt{StateDelaySpec}\ A := (\texttt{Delay}\ (S \times A) \to \mathbb{P}) \to (S \to \mathbb{P}).$$

This is the type of functions from postconditions, predicates over possibly divergent computations that produce an output state and a return value, to preconditions, predicates over input states. The automatically generated effect observation is exactly the weakest precondition function over stateful computations.

$$\Theta_{SD}\ m := \lambda p\ s.((m\ s) \in p)$$

3.4. Hoare Logic Extension

Now we will turn our attention specifically to proving specifications on stateful, possibly diverging programs. As discussed at the end of Section 3.3, 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, Θ_{SD}, for free from our approach. This provides all the necessary tools to state the proposition $\Theta_{SD} \vdash m \in w$, which states that a stateful, possibly divergent computation, m of type `StateDelay` A, satisfies a specification, w of type `StateDelaySpec` A.

3.4.1. Embedding Pre and Post Conditions

So far in this chapter, we have focused on the backwards predicate transformer category of specifications. However, these specifications don't directly give us the tools to write intuitive specifications. For example, we often want to write specifications in terms of pre- and post-conditions. As shown by Maillard et al. (2019), the backwards predicate transformers are expressive enough to encode such specifications. The following encoding function maps pre- and post-condition style specifications over stateful, possibly divergent computations to the `StateDelaySpec` monad.

$$\texttt{encode} : (S \to \mathbb{P}) \to (\texttt{Delay}(S \times A) \to \mathbb{P}) \to \texttt{StateDelaySpec}\ A$$

$$\texttt{encode}\ p_i\ p_o\ := \lambda\ s\ p.\ (s \in p_i \land p_o \subseteq p)$$

The `encode` function takes in a precondition over input states, p_i, and a postcondition over possibly divergent computations that return a pair of an output state and value, p_o. It produces a predicate over states, of type S, and output predicates, of type $\texttt{Delay}(S \times A) \to \mathbb{P}$. This predicates accepts all states, s, that are contained in the precondition, p_i, and output predicates, p, that are supersets of the postcondition, p_o.

The connection between this `encode` function and pre- post-condition style specifications becomes clear when placed in context with the unfolded Dijkstra monad definitions. Suppose we want to prove $\Theta_{SD} \vdash m \in (\texttt{encode}\ p_i\ p_o)$. The following sequence of propositions are all justified either by unfolding definitions or simply reasoning about sets and propositional logic.

$$\Theta_{SD} \vdash m \in (\texttt{encode}\ p_i\ p_o) \tag{3.1}$$

$$\texttt{StateDelaySpec} \vdash (\Theta_{SD}\ m) \leq (\texttt{encode}\ p_i\ p_o) \tag{3.2}$$

$$\texttt{StateDelaySpec} \vdash (\lambda\ s\ p.((m\ s) \in p)) \leq (\lambda\ s\ p.\ s \in p_i \wedge p_o \subseteq p) \tag{3.3}$$

$$\forall s\ p, (s \in p_i \wedge p_o \subseteq p) \to (m\ s \in p) \tag{3.4}$$

$$\forall s,\ \ s \in p_i \to (m\ s) \in p_o \tag{3.5}$$

The `StateDelaySpec` monad is also expressive enough to encode specifications that enforce the conjunction of a list of pre- and post-condition style specifications. For each pair of pre- and post-conditions in the list, the following specification enforces that if the precondition holds on the input, then the postcondition holds on the output.

$$\texttt{encode\_list} : ((S \to \mathbb{P}) \times (\texttt{Delay}\ (S \times A) \to \mathbb{P}))^* \to \texttt{StateDelaySpec}\ A$$

$$\texttt{encode\_list conds} := \lambda\ s\ p.\ (\forall\ p_i\ p_o, (p_i, p_o) \in \texttt{conds} \to s \in p_i \wedge p_o \subseteq p)$$

3.4.2. Recovering Hoare Logic

Section 2.8 presents an ITrees semantics for the IMP language. The `StateDelaySpec` monad can express Hoare logic specifications and all the standard Hoare logic inference rules, presented in Figure 3.1, are valid in this encoding.

$$\frac{}{\{P\}\mathsf{skip}\{P\}} \qquad \frac{\{P\}c_1\{Q\} \qquad \{Q\}c_2\{R\}}{\{P\}c_1; ; ; c_2\{Q\}} \qquad \frac{\{P \wedge b\}c_1\{Q\} \qquad \{P \wedge \neg b\}c_2\{Q\}}{\{P\}\mathsf{if}\ (b)\ \mathsf{then}\ \{c_1\}\ \mathsf{else}\ \{c_2\}\{Q\}}$$

$$\frac{}{\{P[x \mapsto a]\}x{::=}a\{P\}} \qquad \frac{\{P \wedge b\}c\{P\}}{\{P\}\mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{done}\{P \wedge \neg b\}}$$

Figure 3.1: (Standard) Hoare-logic rules.

Definition 11 (Hoare logic encoding)**.** *Given a command, c, and two predicates over states, P and Q, the Hoare logic triple, $\{P\}c\{Q\}$, holds if $\mathit{StateDelaySpec} \vdash \mathit{interp\_imp}\ [\![c]\!] \in \mathit{encode}\ P\ (\lambda t.\ \forall\ s, \mathit{voidE} \vdash t \approx_{=} \mathtt{ret}\ (s, ()) \rightarrow s \in Q)$. That is, if the denotation of c is contained in the specification defined by the precondition P and the postcondition that asserts that any output state s is contained in Q.*

Note that the postcondition in the previous definition accepts any divergent ITrees which corresponds to the fact that Hoare logic specifications are partial correctness specifications.

3.4.3. Generalized Correctness

Classic Hoare logic deals only with partial correctness properties, which guarantee that if a program terminates, then its output satisfies a postcondition (Hoare, 1969). Later work built on Hoare logic dealt with total correctness properties, which guarantees that a program *must* terminate and that its output satisfies a postcondition (Jung et al., 2015; Appel, 2011). This chapter provides specifications that can reason more flexibly about termination and divergence. Some programs, like operating systems and servers, are not supposed to terminate unless a shutdown command is given. 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 `DelaySpec` monad, `StateDelaySpec` monad, and other specification monads that come from the base `DelaySpec` monad are all rich enough 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 :=
       x := 0;
       while ~(x * x = y) do {
              x := x + 1;
       }.
```

This program is intended to compute the integer square root of the value at variable y when it is a perfect square and diverge when it is not a perfect square. We can formalize this specification with two pairs of pre- and post-conditions. The final postcondition in this list relies on the `encode_list` function defined in Section 3.4.1.

$$\begin{aligned}
\texttt{pre1} &:= \lambda s.\ \texttt{is\_square}\ (\texttt{get}\ y\ s) \\
\texttt{post1} &:= \lambda t.\ \exists s.\ \texttt{voidE} \vdash t \approx_{=} \texttt{ret}\ (s, ()) \wedge \texttt{get}\ x\ s \times \texttt{get}\ x\ s = \texttt{get}\ y\ s \\
\texttt{pre2} &:= \lambda s.\ \neg(\texttt{is\_square}\ (\texttt{get}\ y\ s)) \\
\texttt{nat\_sqrt\_spec} &:= \ \texttt{encode\_list}\ [(\texttt{pre1}, \texttt{post1});\ (\texttt{pre2}, \texttt{diverges})]
\end{aligned}$$

3.5. Interaction Tree Traces

As discussed in Section 3.1, not all programming languages can be easily modelled by a sequence of monad transformers applied to the `Delay` monad. For example, a language with IO would be represented by ITrees with uninterpreted events, not with the `Delay` monad. This motivates us to go beyond transformations of `DelaySpec` and develop a Dijkstra monad for `itree E` for any arbitrary event type family `E`.

3.5.1. First Attempt at Specification Monad

The natural first choice for a specification monad for arbitrary ITrees would be a straightforward generalization of the `DelaySpec` monad. However, this does not work. Consider the following flawed monadic structure.

$$
\begin{aligned}
\texttt{ITreeSpec}\ A &:= (\texttt{itree}\ E\ A \to \mathbb{P}) \to \mathbb{P} \\
\texttt{ret}_{its}\ a : \texttt{ITreeSpec}\ A &:= \lambda p.\ \texttt{ret}\ a \in p \\
\texttt{bind}_{its}\ w\ g : \texttt{ITreeSpec}\ A &:= \lambda p.\ (\lambda t.(\exists a.\ t \in \texttt{converges}\ a \land p \in (g\ a)) \lor \\
&\qquad (t \in \texttt{diverges} \land (\texttt{div\_cast}\ t) \in p)) \in w
\end{aligned}
$$

This specification monad is the type family of backwards predicate transformers for ITrees, just like `DelaySpec` is the type family of backwards predicate transformers for the `Delay` monad. The `ret` a definition is identical to the definition for `DelaySpec`. It is the set of all predicates that contain the ITree `ret` a.

Once again the `bind` definition is more complicated. It relies on the `converges` predicate, which asserts that a value a is contained in some leaf of the tree t. This serves as a generalization of testing that a tree is equivalent to `ret` a in the `Delay` monad, as is done in the left disjunct of the `bind` specification in `DelaySpec`. It also relies on the `div_cast` function. The `div_cast` function acts as a kind of safe type-cast for a divergent ITree from `itree E A` to `itree E B`. A divergent ITree of type `itree E A` has no return values, only visible event nodes and silent steps of computation. This means it has the same observable behavior as another ITree of type `itree E A`. The `div_cast` function transforms it into this other, computationally identical, ITree. This serves as a generalization of replacing divergent elements of the `Delay` monad with a properly typed `spin`, the only divergent element of the `Delay` monad. Otherwise, the `bind` w g definition is the same as it was for `DelaySpec`. It builds a set of predicates, p, by testing if a new predicate over ITrees is contained in w. This new predicate contains an ITree t, if either t can converge to a value a and the resulting specification g a contains p, or if t diverges and p contains a computationally identical tree `div_cast` t.

However, this definition does *not* satisfy the monad laws. The following predicate and specification provide a counterexample.

$$p_c : \mathtt{itree}\ E\ \mathsf{unit} \to \mathbb{P} := \lambda t.E \vdash t \approx_{=} \mathtt{Vis}\ e\ (\lambda x.\mathtt{ret}\ ())$$

$$w_c : \mathtt{ITreeSpec}\ \mathsf{unit} := \lambda p.\ ((\mathtt{Vis}\ e\ (\lambda x.\mathtt{ret}\ ())) \in p)$$

We can prove that $p_c \in w_c$ and that $p_c \notin \mathtt{bind}_{its}\ w_c\ \mathtt{ret}_{its}$. According to the monad laws, $\mathtt{bind}_{its}\ w_c\ \mathtt{ret}_{its}$ should be equivalent to w_c. 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.

3.5.2. Interaction Traces

To develop a Dijkstra monad for ITrees with uninterpreted events, we first develop a trace model for ITrees. These traces are referred to as ITraces. Intuitively, ITraces represent a single linear path through an ITree. This path consists of a potentially infinite sequence of visible events, paired with answers from the environment. These answers correspond to the particular branch from the node that is taken to create this particular path. We implement ITraces as ITrees with a specially designed event type family `EvAns`.

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

The `EvAns` type family takes in a base event type family `E` and has two constructors. The `evans` constructor packages a visible event, `e : E A`, with a potential answer, `ans : A`. The answer type of this `EvAns` event is `unit`. This means that there is only one branch following this visible event node. An `evans` event is only possibly to create when the original event's answer type is inhabited. The `evempty` constructor signifies the end of an ITrace that corresponds to an event with an uninhabited answer type. The `evempty` constructor takes in an event, `e : E A`, and a function from `A` to the empty type `void`, which serves as a proof that `A` is empty.

Definition 12 (Interaction Trace)**.** *An Interaction Trace (ITrace) of type* ***`itrace E A`*** *is an ITree of*

$$[\textsc{TraceRefPre}]\ \frac{\texttt{e : E A} \quad \texttt{a : A}}{(\texttt{e}, \texttt{evans e a}) \in \texttt{EvRefPre E}}$$

$$[\textsc{TraceRefPost1}]\ \frac{\texttt{e : E A} \quad \texttt{Hempty : A -> void}}{(\texttt{e}, \texttt{evempty e Hempty}) \in \texttt{EvRefPre E}}$$

$$[\textsc{TraceRefPost2}]\ \frac{\texttt{e : E A} \quad \texttt{a : A}}{(\texttt{e}, \texttt{evans e a}, \texttt{a}, ()) \in \texttt{EvRefPost E}}$$

Figure 3.2: Event refinement definition

type ***`itree (EvAns E) A`****.*

Implementing ITraces in terms of ITrees gives us monadic structure and equational theory for free. These features are relevant to working with ITraces. Notably, appending traces is simple to define in terms of `bind`.

Definition 13 (ITrace append)**.** *Given two ITraces, tr_1 and tr_2, tr_2 appended on to tr_1, written as $tr_1 +\!\!+\ tr_2$, is defined as* ***`bind`*** *tr_1 $(\lambda x.tr_2)$.*

Reusing `bind` to define trace appending makes it easy to prove key properties about it.

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 consists of a linear sequence of events that converges to at most one return value.

3.5.3. ITrace Refinement

This section presents a trace refinement relation expressing when an ITrace is contained in the behavior of an ITree. We build the trace refinement relation using `rutt`, defined in Figure 2.11, along with specialized event pre- and post-relations defined in Figure 3.2.

Definition 14 (ITrace refinement)**.** *An ITrace, tr :* ***`itrace E R`****, refines an ITree, t :* ***`itree E R`****, if* *the following proposition holds:* $\mathtt{EvRefPre}; \mathtt{EvRefPost} \vdash tr \approx_{=} t$*. If tr refines t we write* $E \vdash tr \lesssim t$*.*

For inhabited types, `EvRefPre` pairs events `e : E A` with event answers `evans e a` for any possible answer `a`. For uninhabited types, `EvRefPre` pairs events `e : E A` with evidence that `A` is empty. `EvRefPost` enforces that the only continuations that can be paired with events `e` and `evans e a` are `tt` and `a`.

For example, consider the ITree `Vis Get` ($\lambda x.$`ret` x). Suppose we want to prove that it is refined by the ITrace `Vis (evans Get 0)` ($\lambda x.$`ret` 0). When comparing the events, `EvRefPre` checks that the `Get` event in the tree matches the `evans Get 0` event in the trace. Because of `EvRefPost`, the refinement relation then feeds the `0` answer to the event into the continuation for the tree and enforces $\mathtt{stateE} \vdash \mathtt{ret}\ 0 \lesssim \mathtt{ret}\ 0$. This does hold according to the rules of `rutt`. This notion of traces and trace refinement yields a new notion of ITree equivalence.

Definition 15 (Trace equivalence)**.** *Two ITrees,* t_1 *and* t_2*, are trace equivalent if they are refined by the same set of ITraces.*

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

Theorem 2. *Two ITrees are equivalent if and only if they are trace equivalent.*

Knowing that these relations are equivalent ensures that this trace model of ITrees captures all essential information about ITrees.

We provided verified reasoning principles relating ITraces and ITrees through this refinement relation. For example, the following theorem allows us to take an ITrace that refines a `bind` ITree, and decompose it into two parts, one which refines the head, and one which refines the continuation.

Theorem 3 (Refinement inversion for `bind`)**.** *Given an ITree, t, a continuation f, and an ITrace, tr, such that* $E \vdash tr \lesssim \mathtt{bind}\ t\ f$*, tr can be decomposed into another ITrace,* tr'*, and a trace continuation, g, such that* $E \vdash tr' \lesssim t$*, and* $\forall a,\ E \vdash g\ a \lesssim f\ a$*, and* $E \vdash tr \approx_{=} \mathtt{bind}\ tr'\ g$*.*

3.6. ITree Trace Specifications

This section presents a specification monad for ITrees based on the previously introduced ITraces and the inductive IO specification monad presented in Maillard et al. (2019). We begin by creating a straightforward adaptation of the inductive IO specification monad for reasoning about ITrees that terminate. This specification monad will be based on ITraces and will serve to introduce key ideas that underly the specification monad for all ITrees.

Once again, the specification monad is going to be a collection of backwards predicate transformers. These specification monads reason about ITrees in terms of paths through the tree structure. The preconditions reason about the path taken through the ITree so far. And the postconditions reason about the total path taken through the ITree. These paths, which also serve as a log of interactions with the environment, can be expressed as ITraces.

Definition 16 (Event logs). *Given an event type family `E`, the type of event logs over `E`, `log E` contains the convergent elements of `itrace E unit`.*

Using these event logs, we can formalize a specification monad for terminating ITrees.

$$\texttt{TraceSpecInd}\ A := (\texttt{log E} \times A \to \mathbb{P}) \to \texttt{log E} \to \mathbb{P}$$

$$\texttt{ret}_{tsi}\ a := \lambda p\ l.\ ((l, a) \in p)$$

$$\texttt{bind}_{tsi}\ w\ g := \lambda p\ l.((\lambda(l', a).\ ((p, l') \in (g\ a),\ l) \in w))$$

The specification monad `TraceSpecInd` is a backwards predicate transformer. The precondition is a predicate over event logs, representing the previous interactions with the environment. The postcondition is a predicate over event logs, representing future interactions with the environment, and output values. Structurally, these backwards predicate transformers are similar to state backwards predicate transformers where the event logs are the state.

The `ret` a specification contains postconditions p paired with input logs l such that p contains (l, a). This maps postconditions, p, to the set of input logs where,if the computation immediately halts and returns a it will be accepted in p.

The `bind` $w\ g$ closely mirrors the `bind` specification presented in Section 3.2. It takes in a postcondition, p, and an input log, l, creates a new postcondition based on g, and tests if this new postcondition along with the input log is contained in w. This new postcondition takes in an output log, l', and a return value, a, and tests if the pair (p, l') is in the specification $g\ a$. This complicated definition will be justified by the fact that the resulting final Dijkstra monad for ITrees satisfies all the relevant laws.

To go from `TraceSpecInd` to a specification monad suitable for reasoning about possibly divergent ITrees, we need to change the postconditions to reason about potentially infinite sequences of interactions with the environment. This is accomplished simply by replacing predicates over $\texttt{log E} \times A$ with $\texttt{itrace E}\ A$. From there, we need to augment the definitions of `ret` and `bind` to handle potentially infinite ITraces in the postconditions.

$$\begin{aligned} \texttt{TraceSpec}\ A &:= (\texttt{itrace E}\ A \to \mathbb{P}) \to \texttt{log E} \to \mathbb{P} \\ \texttt{ret}_{ts}\ a &:= \lambda p\ l.\ ((l +\!\!+\ \texttt{ret}\ a) \in p) \\ \texttt{bind}_{ts}\ w\ g &:= \lambda p\ l.((\lambda tr.\ ((\exists l'\ a,\ E \vdash tr \approx_{=} l' +\!\!+\ \texttt{ret}\ a \wedge (p, l') \in (g\ a) \\ &\qquad \texttt{diverges}\ tr \wedge (\texttt{div\_cast}\ tr \in p))),\ l) \in w) \end{aligned}$$

The $\texttt{ret}_{ts}\ a$ specification contains all postconditions, p, and input logs, l, such that p contains $l +\!\!+\ \texttt{ret}\ a$. Much like with the `TraceSpecInd` definition, this maps postconditions, p, to the set of input logs where,if the computation immediately halts and returns a it will be accepted in p.

The $\texttt{bind}_{ts}\ w\ g$ specification generalizes the `TraceSpecInd` definition much like the `DelaySpec` definition generalizes the `IDSpec` definition. It maps postconditions, p, to the set of input logs, l, such that l paired with a new postcondition computed using g is contained in w. This new postcondition accepts two kinds of ITraces: convergent ITraces which can be decomposed into an event log l' followed by a return value a, where p paired with l' is contained in the specification $g\ a$;

and divergent traces which have the same visible behavior as traces contained in p.

Theorem 4. *`TraceSpec` is a valid specification monad.*

Given the `TraceSpec` specification monad, all that remains to define a Dijkstra monad is a valid effect observation. The choice of the effect observation follows from the intuition that the preconditions reason about the interactions with the environment that have already taken place and the postconditions reason about the total sequence of interactions with the environment.

$$\Theta_{ts}\ t := \lambda p\ l. \forall tr.\ E \vdash tr \lesssim t \rightarrow (l +\!\!+\ tr) \in p$$

The Θ_{ts} effect observation takes an ITree, t, and a postcondition, p, and maps it to the set of input logs, l, such that for any trace through t, tr, p contains $l +\!\!+\ tr$. The postcondition needs to accept the full sequence of interactions consisting of the input log followed by any valid trace through t.

Theorem 5. *Θ_{ts} is a valid effect observation from ITrees to the `TraceSpec` monad. Futhermore, the computational monad, ITrees, specification monad, `TraceSpec`, and the effect observation, Θ_{ts}, form a valid Dijkstra monad.*

3.7. Trace Specification Examples

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

3.7.1. Example with IO

We first demonstrate how to use the `TraceSpec` monad on a simple example. Recall the nondeterminism example presented in Section 3.1, which is presented here as an ITree.

```
queryUntilFalse : itree NonDet unit :=
  iter (λ x. bind (trigger Decide))
        (λ b. if b then ret (inl ()) else ret (inr ())) ().
```

This computation executes a loop in which a nondeterministic choice determines whether the loop will perform another iteration or will terminate. As discussed in Chapter 2, the `inl` values represent continuation signals for `iter` loops, and the `inr` values represent halting signals. The set of traces that refine this computation contains convergent traces which have a finite sequence of `Decide` events answered with `true` terminated by a `Decide` event answered with `false`, and divergent traces with an infinite sequence of `Decide` events answered with `true`. This behavior can be encoded in `TraceSpec` with little overhead.

3.7.2. Predicates over ITraces

ITraces can be viewed as a kind of potentially infinite stream and benefit from many predicates typically used to reason about streams. For example, we can define a predicate that enforces some constraint on all elements of an ITrace.

Definition 17 (Trace forall predicate)**.** *Given a predicate over events, p_E of type $\forall A.\ E\ A \to A \to \mathbb{P}$, and a predicate over return values, p_R, the trace forall predicate, written as* ***`traceForall`*** *p_E p_R contains precisely the ITraces whose events and recorded answers each satisfy p_E and whose return value, if they converge, satisfies p_R.*

To encode the behavior of the convergent traces, we also need a slightly different specification which relies on two predicates over events, one for the final event in the trace and one for every prior event.

Definition 18 (Trace all but final event predicate)**.** *Given two predicates over events, p_E and p_{EF} of type $\forall A.\ E\ A \to A \to \mathbb{P}$, and a predicate over return values, p_R, the trace all but final event predicate, written as* ***`allButFinal`*** *p_E p_{EF} p_R, contains precisely the converge ITraces where all but the final event and recorded answer in the trace satisfy p_E, the final event and recorded answer satisfies p_{EF}, and the return value satisfies p_R.*

3.7.3. Encoding Simple Specifications

As previously discussed, backwards predicate transformers are very expressive but don't have the structure we typically associate with specifications. So it is useful to write an encoding function from pre- and post-condition style specifications to `TraceSpec`.

```
queryUntilFalse_pre : log NonDet → ℙ := λ l. l = nil
queryUntilFalse_post_trace : itrace NonDet unit → ℙ :=
allButFinal (λ e. e = evans Decide true)
  (λ e. e = evans Decide false) ⊤
queryUntilFalse_post : itrace NonDet unit → ℙ :=
λ tr.
    (tr ∈ diverges →
    tr ∈ traceForall (λ e. e = evans Decide true) ⊤) ∧
    (tr ∈ converges () →
      tr ∈ queryUntilFalse_post_trace)
```

Figure 3.3: Predicates used to define queryUntilFalse specification

$$\texttt{encode\_ts} : (\texttt{log E} \to \mathbb{P}) \to (\texttt{itrace E } A \to \mathbb{P}) \to \texttt{TraceSpec } A$$

$$\texttt{encode\_ts } p_i \; p_o := \lambda p \; l. \; l \in p_i \wedge (\forall tr. (l +\!\!+ \; tr) \in p_o \to (l +\!\!+ \; tr) \in p)$$

With these tools in place, we can define the specification of our example. This specification relies on predicates defined in Figure 3.3. 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 constrain the rest of the behavior in the postcondition, forcing convergent traces to have a sequence of true decisions terminating with a single false and forcing divergent traces to only have true decisions.

```
queryUntilFalse_spec : TraceSpec NonDet unit :=
  encode_ts queryUntilFalse_pre queryUntilFalse_post
```

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.

3.7.4. Example with IO and State

Just like with DelaySpec, we can use the state monad transformer to create a specification monad for the state transform of the ITree monad, stateful computations which return ITrees. This specification monad is sufficient for specifying programs 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 x;
  y := x + y
}
```

```
a ← Input; Store x a;
iter (fun _ ⇒
  b ← Load y; Output y;
  a ← Load x; Store y (x + y))
  ()
```

We can once again develop an encoding of pre- and post-conditions 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.

```
mults_of_n_from_m : ℕ → ℕ → itrace IO unit :=
  cofix f n m. Vis (evans (Output m) tt) (λ x. f n (n + m))
mults_of_n : ℕ → itrace IO unit := λ b. mults_of_n_from_m n 0
```

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

```
pre_mult := λ l s. l = nil ∧ get y s = 0
post_mult := λ tr. ∃ n k. IO ⊢ Vis (evans Input n) k ≈= mults_of_n n
```

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

CHAPTER 4

Semantics For Noninterference Using Interaction Trees

This chapter was previously published as Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. Semantics for Noninterference with Interaction Trees. In Karim Ali and Guido Salvaneschi, editors, 37th European Conference on Object-Oriented Programming (ECOOP 2023), volume 263 of Leibniz International Proceedings in Informatics (LIPIcs), pages 29:1–29:29, Dagstuhl, Germany, 2023b. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-281-5. doi: 10.4230/LIPIcs.ECOOP.2023.29. URL https://drops.dagstuhl.de/opus/volltexte/2023/18222. I was the primary author and did most of the research.

4.1. Introduction

Information-flow properties state that programs respect the information-security policies of their inputs . For instance, *noninterference*—the most basic information-flow property—states that secret data cannot influence publicly observable behavior. There are many languages designed to enforce information-flow properties, guaranteeing that programs treat their sensitive inputs correctly (Pottier and Simonet, 2003; Hritcu et al., 2013; Magrino et al., 2016; Polikarpova et al., 2020).

Information-flow properties state that programs respect the information-security policies of their inputs . For instance, *noninterference*—the most basic information-flow property—states that secret data cannot influence publicly observable behavior. There are many languages designed to enforce information-flow properties, guaranteeing that programs treat their sensitive inputs correctly (e.g., Pottier and Simonet, 2003; Magrino et al., 2016; Polikarpova et al., 2020). The importance of information-security properties has increasingly led to verification efforts for such languages and systems (Jia and Zdancewic, 2009; Azevedo de Amorim et al., 2014). These efforts, however, are mostly limited to source-level guarantees for a single language. For security guarantees to be meaningful, the entire language toolchain must support them.

Verifying a toolchain requires more than just certifying the guarantees for a single language design.

Language toolchains must reason about multiple *interacting* languages. At the source level, programs are often written in diverse languages that interact by embedding one language inside another. For instance, C programs often include embedded assembly code. Toolchains also include compilers that transform code, again requiring cross-language reasoning.

The complexity of multi-language settings makes the already-fraught choice of language representation even more complicated. For instance, much prior work either uses operational semantics defined as relations on syntax or uses trace models defined as predicates over lists or streams of observations (Plotkin, 1981; Leroy, 2009; Jung et al., 2015). Such definitions often require auxiliary constructs, like program counters or evaluation contexts, that make proofs brittle and hard to compose. Unfortunately, these constructs often require different representations in different languages, seriously complicating the task of reasoning about cross-language security.

Interaction Trees (ITrees) (Xia et al., 2020; Zakowski et al., 2021b) provide an alternative: a runnable denotational semantics for effectful, potentially-nonterminating programs, with a library implemented in Coq. ITrees can express the semantics of diverse programming language features, and thus many different languages. This versatility makes ITrees well-suited to cross-language reasoning (Xia et al., 2020) and reasoning about real-world toolchains (Zakowski et al., 2021b; Koh et al., 2019).

Prior works with ITrees reason about program semantics primarily through a notion of equivalence based on *weak bisimilarity*, which considers programs equivalent so long as they differ only by a finite number of silent steps. Information-flow properties, however, require more nuanced reasoning principles because some program behaviors are visible to an attacker while others are not. In this chapter we introduce *indistinguishability* relations for ITrees to capture these intuitions. By defining these indistinguishability relations on ITrees—that is, on a common semantic domain rather than on syntax—we greatly simplify cross-language reasoning.

Since indistinguishability relations model the observations of an adversary, they must accurately reflect an attacker model. That is, they must capture our intuitions about what an adversary can and cannot see. ITrees allow us to define indistinguishability relations parametrically over a large class of *event signatures*, giving proof engineers the ability to specify many attacker models. However, ITrees treat one common observation specially: termination. We thus define two indistinguishability relations, corresponding to two common classes of attacker models: *progress-sensitive* and *progress-insensitive* (Volpano et al., 1996; Volpano and Smith, 1997; Sabelfeld and Myers, 2003). The progress-sensitive relation represents attackers who can distinguish a program that silently diverges from both a program that eventually emits more events and one that terminates. The result is a very strong security guarantee. Unfortunately, enforcing it usually leads to languages which cannot express many common programming tasks. For instance, most type systems for enforcing progress-sensitive noninterference disallow loops that depend on secret data. The progress-insensitive relation, which represents an attacker who cannot determine if a program will make progress in the future, is less demanding, but provides considerably less security (Askarov et al., 2008).

We add both the indistinguishability relations and a variety of metatheoretic results to the ITrees library. Constructing the relations and proving the metatheorems requires careful treatment of the interplay between nontermination and the interactions of a program with its environment, involving delicate mixed inductive-coinductive reasoning. However, the design of the metatheorems allows a proof engineer to avoid manual use of coinduction entirely. Moreover, the results further connect indistinguishability to the standard ITrees notion of bisimilarity, providing compatibility with existing results.

To validate our design, we verify a simple toolchain for cross-language noninterference. We use a simple imperative source language, Imp, and a simple assembly language, Asm. Imp includes exceptions and embedded Asm blocks in addition to standard features. We include exceptions primarily to show how our indistinguishability semantics works with effects that may alter control flow, which is particularly tricky for information-flow reasoning. However, this also requires us to extend the ITrees library orthogonally to our extensions for reasoning about security.

Our toolchain includes two different type systems for Imp and a complier from Imp to Asm. One type system guarantees progress-sensitive noninterference, and the other progress-insensitive noninterference. In addition to standard information flow typing rules, the type systems allow for *semantic typing*: any semantically secure program can be considered well typed. This added flexibility allows programs with embedded assembly to type check without giving a type system to ASM, and it demonstrates the powerful semantic composition of our security reasoning. We further verify that our IMP-to-ASM compiler preserves both kinds of noninterference. Note that this preservation does not rely on the type system, but only on semantic security. Indeed, this is required in order to allow for security preservation with semantic typing.

Section 4.2 reviews background on information-flow control, the IMP language, and its semantics defined with ITrees. The contributions of this chapter are as follows.

- Section 4.3 extends the ITrees library with exceptions and exception handlers.
- Section 4.4 adapts ITrees metatheory to reason about security guarantees, defining progress-sensitive and progress-insensitive notions of indistinguishability and using them to provide definitions of noninterference.
- Section 4.5 uses ITrees and our new relations to prove the correctness of two standard information-flow type systems for IMP, one progress-sensitive and one progress-insensitive. Both systems additionally allow a "semantic typing" (Gregersen et al., 2021) escape hatch for programs that satisfy the semantic security conditions but do not syntactically type check.
- Section 4.6 adapts Xia et al.'s [102] simple compiler from IMP to (simplified) assembly language to include the exceptions and print effects in our variant of IMP. We then show that compiler correctness, as defined by Xia et al., immediately implies security preservation using only the metatheory of indistinguishability.

All definitions and theorems described in this chapter have been formalized in Coq.

4.2. Background

This section reviews background on information-flow control, interaction trees, and IMP.

4.2.1. Information-Flow Control

We represent information-security policies using a set of *information-flow labels* $\mathcal{L}$ that must form a preorder. That is, there is a reflexive, transitive relation $\sqsubseteq$ (pronounced "flows to") on labels where $\ell \sqsubseteq \ell'$ means that any *adversary* who can see information with label ℓ' can also see information with label ℓ. We also identify adversaries with labels. An adversary at label ℓ can only see information with labels that flow to ℓ. Information-flow systems use a variety of orderings, including simply "public" and "secret," subsets of permissions (Zeldovich et al., 2011), lattices over principals making up a system (Myers and Liskov, 1998; Arden et al., 2015; Stefan et al., 2011), and orderings based on logical implication (Polikarpova et al., 2020).

The classic information-flow security policy is *noninterference*: if an adversary cannot distinguish a program's inputs, they should not be able to distinguish its outputs or its interactions with the environment. Because information-flow labels determine which data an adversary can observe, a semantic version of noninterference requires a semantic model of information-flow labels. Sabelfeld and Sands (2001) suggest modeling labels as partial equivalence relations (PERs) on terms. PERs are relations that are symmetric and transitive, but not necessarily reflexive. PERs act like equivalence relations on a subset of their domain. For information-flow security, such PERs are called "indistinguishability relations."

This model further asserts that indistinguishable programs take indistinguishable inputs to indistinguishable outputs. That is, related programs, applied to related inputs, produce related computations. This closure property allows a semantic version of noninterference to be defined as self-relation of a program. A program is related to itself—and noninterfering—if and only if, for every adversary, given any two inputs an adversary cannot distinguish, it produces two computations that adversary cannot distinguish.

As we will see in Section 4.4, indistinguishability gives a natural way to reason about noninterference using ITrees. Requiring every indistinguishability relation to be a PER, however, corresponds to strong assumptions about the adversary. In particular, it requires that the adversary be able to

$$\begin{array}{lcll}
\text{Expressions} & e & ::= & x \mid n \mid e + e \mid e - e \mid e * e \\
\text{Commands} & c & ::= & \mathsf{skip} \mid x := e \mid c_1 \,;\, c_2 \mid \mathsf{while}\ (e)\ \mathsf{do}\ \{c\} \\
 & & \mid & \mathsf{if}\ (e)\ \mathsf{then}\ \{c_1\}\ \mathsf{else}\ \{c_2\} \mid \mathsf{print}(\ell, e) \mid \mathsf{inline}\ \{a\} \\
\text{Inlined Assembly} & a & ::= & \text{(see Section 4.6)}
\end{array}$$

Figure 4.1: Imp syntax, where x is a variable, n is a number, and ℓ is an information-flow label.

distinguish a program that silently diverges from a program that takes arbitrarily long to produce an observable interaction with its environment. Noninterference against this strong adversary is known as *progress-sensitive* noninterference. While this strength provides more security, enforcing progress-sensitive noninterference results in a prohibitively expensive programming model (Section 4.5.1, Sabelfeld and Myers, 2003; Volpano and Smith, 1997). To allow for enforcement of *progress-insensitive* noninterference, the indistinguishability model is often relaxed to not require transitivity (Vassena et al., 2019; Rajani and Garg, 2018; Gregersen et al., 2021). This relaxation makes reasoning about noninterference in programs with loops easier.

4.2.2. Semantics for Imp with Security Labels

To explore how ITrees can help us verify noninterference properties, we will use a simple imperative language, Imp, as a running example and case study. Conveniently, previous work on both ITrees (Xia et al., 2020) and noninterference (Sabelfeld and Myers, 2003) use Imp as case studies, ensuring that the connection we make corresponds with existing tools and techniques in both domains. Our version of Imp, presented in Figure 4.1, includes features not present in the works cited above: the ability to print expressions to one of several output streams, and the ability to inline code from a simple assembly language. Section 4.3 will further extend Imp to allow throwing and catching exceptions. The output streams are indexed by information-flow labels, and we think of stream ℓ as being visible to any adversary at or above ℓ, but no others. Thus, printing secret information to a public stream leaks data.

The assembly language, Asm, is a simplification of standard assembly language. We allow an infinite number of registers, and we assume that the heap is addressed by variables, as in Imp. We do not allow dynamic jumps, only jumps to fixed addresses. Beyond those simplifications, we include features similar to those in Imp: we allow printing to streams indexed by information-flow labels and, as we show later, the Asm semantics can model *uncaught* exceptions, both features necessary for correct compilation of Imp code. We discuss the syntax and semantics of Asm in more detail in Section 4.6.

As in languages like C, embedding Asm in Imp allows developers more control over the performance of their code. For instance, the simple compiler in Section 4.6 would compile the Imp program $y \coloneqq x + 1 \,;\, z \coloneqq x + 2$ to an Asm program that loads data from x into a register twice, once for each assignment. Since Loads are relatively expensive, when the Imp code above appears in a critical loop a developer might replace it with the following Asm code:

$$\begin{array}{rlll}
\textsc{Start}: & \textsc{load} & \$0 & \leftarrow x \\
 & \textsc{add} & \$0 & \leftarrow \$0, 1 \\
 & \textsc{store} & y & \leftarrow \$0 \\
 & \textsc{add} & \$0 & \leftarrow \$0, 1 \\
 & \textsc{store} & z & \leftarrow \$0 \\
 & \textsc{jmp} & \textsc{Exit} &
\end{array}$$

This program starts from the Start label, and terminates the program by jumping to the Exit label. Unlike our compiler's output, this custom Asm only has one load instruction.

Giving semantics to Imp using ITrees requires defining events representing possible interactions between an Imp program and its environment. Imp has three types of events: `stateE` for representing interactions with the heap state, `regE` for representing interactions with the register state, and `printE` for representing output. There are two constructors for `stateE` events, one for reading and one for writing.

$$\texttt{get} : \mathsf{var} \to \texttt{stateE}(\mathbb{N}) \qquad\qquad \texttt{set} : \mathsf{var} \to \mathbb{N} \to \texttt{stateE}(\mathsf{unit})$$

$$\boxed{[\![e]\!]_e : \texttt{itree progE}\ \mathbb{N}} \qquad\qquad \boxed{[\![c]\!]_c : \texttt{itree progE}\ \mathsf{unit}}$$

$$\begin{aligned}
[\![x]\!]_e &= \texttt{trigger}\ \texttt{get}(x) \\
[\![n]\!]_e &= \texttt{ret}\ n \\
[\![e_1{+}e_2]\!]_e &= x \leftarrow [\![e_1]\!]_e\,; \\
&\ \ y \leftarrow [\![e_2]\!]_e\,; \\
&\ \ \texttt{ret}\ (x+y)
\end{aligned}$$

$$\begin{aligned}
[\![\mathsf{skip}]\!]_c &= \texttt{ret}\ () \\
[\![x := e]\!]_c &= n \leftarrow [\![e]\!]_e\,; \texttt{trigger}\ \texttt{set}(x, n) \\
[\![\mathsf{print}(\ell, e)]\!]_c &= n \leftarrow [\![e]\!]_e\,; \texttt{trigger}\ \texttt{print}(\ell, n) \\
[\![c_1\,;c_2]\!]_c &= [\![c_1]\!]_c\,; [\![c_2]\!]_c \\
\left[\!\!\left[\begin{array}{l}\mathsf{if}\ e\\ \mathsf{then}\ \{c_1\}\\ \mathsf{else}\ \{c_2\}\end{array}\right]\!\!\right]_c &= n \leftarrow [\![e]\!]_e\,; \begin{array}{l}\texttt{if}\ n \neq 0\\ \texttt{then}\ \ [\![c_1]\!]_c\\ \texttt{else}\ \ [\![c_2]\!]_c\end{array} \\
[\![\mathsf{while}\ (e)\ \mathsf{do}\ \{c\}]\!]_c &= \texttt{iter}\left(\begin{array}{l}\lambda\_\,.\,n \leftarrow [\![e]\!]_e\,;\\ \qquad \texttt{if}\ n \neq 0\\ \qquad \texttt{then}\ ([\![c]\!]_c\,; \texttt{ret}\ \texttt{inl}())\\ \qquad \texttt{else}\ \texttt{ret}\ \texttt{inr}()\end{array}\right)()\\
[\![\mathsf{inline}\ \{a\}]\!]_c &= [\![a]\!]_{\texttt{asm}}
\end{aligned}$$

Figure 4.2: Imp denotational semantics

The $\texttt{regE}$ events require another two constructors, one for reading and one for writing.

$$\texttt{getreg} : \textit{reg} \to \texttt{regE}(\mathbb{N}) \qquad\qquad \texttt{setreg} : \textit{reg} \to \mathbb{N} \to \texttt{regE}(\mathsf{unit})$$

There is only one constructor for $\texttt{printE}$ events: $\texttt{print} : \mathcal{L} \to \mathbb{N} \to \texttt{printE}(\mathsf{unit})$.

As Imp programs can produce all three types of events, we combine them with disjoint union. The resulting event type for Imp programs is $\texttt{progE} = \texttt{regE} \oplus \texttt{stateE} \oplus \texttt{printE}$. For notational simplicity, we elide the injection operator when using these compound events.

Figure 4.2 presents the denotation of Imp using these events. Note that there are two denotation functions: $[\![\cdot]\!]_e$ for expression and $[\![\cdot]\!]_c$ for commands. As expressions produce numbers and commands have no output, $[\![\cdot]\!]_e$ produces computations of type $\texttt{itree progE}\ \mathbb{N}$, while $[\![\cdot]\!]_c$ produces computations of type $\texttt{itree progE}\ \mathsf{unit}$. The function $[\![\cdot]\!]_{\texttt{asm}}$ gives ITree-based semantics to Asm. Its full definition can be found in the work of Xia et al. (2020); we discuss the modifications necessary to accommodate our changes in Section 4.6.

The denotation for expressions is fairly straightforward, and, importantly for proofs, completely compositional—an expression's meaning is constructed from that of its subexpressions. The denotation of a variable is a `get` event, a literal n becomes `ret` n, and arithmetic expressions simply denote each argument and return the resulting value using bind.

Most commands are equally simple and compositional. `skip` is an immediate `ret`. Both assignment and `print` first denote the argument and then bind the result into the appropriate event. Sequencing is implemented with bind on a unit value that we elide. Conditionals are denoted by first denoting the condition, and then return the denotation of either the left or right command depending on the result.

Loops are more complex and make use of the `iter` combinator. The combinator expects a function that returns `itree progE` $(\mathsf{unit} \oplus \mathsf{unit})$, where a left value indicates "continue" and a right value indicates that the loop should terminate. The function given to `iter` first computes the value of the loop's guard expression. If the value is not zero, it sequences a single denotation of the loop body with `ret inl`$()$, indicating the loop should continue. Otherwise, if the value is zero, it signals to halt the iteration with `ret inr`$()$.

4.2.3. Handlers and Interpretations

As discussed in Chapter 2, the events in an ITree can be thought of as a kind of syntax. Even though we give them names that suggest certain behaviors, like `get` and `set`, nothing about their structure enforces this behavior. Consider the ITree `trigger set`$(x, 0)$; `trigger get`(x): while the names suggest that the result of this `get` should be 0, it actually produces a tree with one branch for every natural number. Likewise, the ITree $[\![c]\!]_c$ representing an Imp program c does not fully express the behavior we would expect from c because it has uninterpreted state events.

The behavior of events is determined by an event handler. For example, consider h_{prog} which uses the standard monadic interpretation of state to interpret `progE` events:

$$\begin{aligned}
h_{prog}(\texttt{get}(x)) &= \lambda(r,h).\,\texttt{ret}\ (r,h,h(x)) \\
h_{prog}(\texttt{set}(x,n)) &= \lambda(r,h).\,\texttt{ret}\ (r,h[x \mapsto n],()) \\
h_{prog}(\texttt{getreg}(x)) &= \lambda(r,h).\,\texttt{ret}\ (r,h,r(x)) \\
h_{prog}(\texttt{setreg}(x,n)) &= \lambda(r,h).\,\texttt{ret}\ (r[x \mapsto n],h,()) \\
h_{prog}(\texttt{print}(\ell,n)) &= \lambda(r,h).\,\texttt{trigger}\ \texttt{print}(\ell,n)\,;\texttt{ret}\ (r,h,())
\end{aligned}$$

Any event handler can be lifted to a function from ITrees to effectful computations using the `interp` function, which traverses an ITree, replacing each event with the effectful computation assigned by the handler. The full semantics of an Imp program is the *interpreted* ITree, `interp` $h_{prog}\ \ [\![c]\!]_c$.

4.2.4. Inlined Asm and Undefined Behavior

Adding support for inlined Asm code introduces a new complication to the semantics of Imp: undefined behavior. To analyze the correctness and security of a language toolchain, we need to define the behavior of source-level programs. The semantics defined in Section 4.2.2 and Section 4.2.3 do that for Imp as long as any inlined Asm has well-defined behavior. However, consider the following Imp program, which contains inlined Asm.

$$\begin{array}{rcrlllll}
p & = & c\,;\mathsf{inline}\ \{\ \textsc{Start}: & \textsc{brz} & \$0 & \text{A1} & \text{A2} & \\
 & & \text{A1}: & \textsc{load} & X & \leftarrow & 0 & \\
 & & & \textsc{jmp} & \textsc{Exit} & & & \\
 & & \text{A2}: & \textsc{load} & X & \leftarrow & 1 & \\
 & & & \textsc{jmp} & \textsc{Exit} & & & \}
\end{array}$$

The inlined Asm program looks at the value in register 0 and, if it is zero, jumps to address A1; otherwise it jumps to address A2. Thus, the value of X after executing program p depends on the value of register \$0 after c is executed. However, it is not clear what the register's value will be when this program is compiled and run, since reasonable compilers could use the register \$0 in different ways—or not at all—to compile the Imp command c, resulting in different register states. We thus consider inlining any Asm program that relies on the initial values of registers to be undefined behavior. We formalize this property in Section 4.5.3. We further take the same approach as CompCert,[4] and only verify the correctness and security of programs that are well-defined.

4.3. Exceptions with Interaction Trees

As mentioned in Section 4.1, we include exceptions in Imp since they are an important example of an effect which can change the control flow. In this section, we show how to model exceptions with ITrees by adding throw and catch constructs to Imp as follows:

$$\text{Commands} \quad c \quad ::= \quad \cdots \mid \mathsf{throw}(\ell) \mid \mathsf{try}\ \{c_1\}\ \mathsf{catch}\ \{c_2\}$$

Note that the throw command includes an information flow label, specifying who may see the exception.

4.3.1. Exceptions as Halting Events

We model exceptions in ITrees as *halting events*. Recall from Section 2 that events create one branch for every possible response from the system. If an event has an uninhabited response type, then that continuation can never be run since the answer type has no values. We call such events *halting* because they force the computation to stop. We formalize this with the following lemma:

Lemma 6. *Suppose A is an uninhabited type and e is an event of type $E\ A$, then given any continuations k_1 and k_2 and any return relation $\mathcal{R}$, $E \vdash \texttt{Vis}\ e\ k_1 \approx_{\mathcal{R}} \texttt{Vis}\ e\ k_2$.*

The continuation of a halting event cannot be run and has no effect on the computational content of the ITree. This allows a programmer to assign such an ITree any desired return type without changing its computational content. This property makes halting events useful for modeling (uncaught) exceptions: an exception can have any type and causes computation to stop. To represent exceptions using this strategy, we use an event type excE with only a single constructor $\texttt{exc} : \mathit{Err} \to \texttt{excE}(\emptyset)$

[4] Personal communication with Xavier Leroy.

which takes the exception's label and produces an event with an empty answer type. This allows us to define $[\![\mathsf{throw}(\ell)]\!]_c = \texttt{trigger}\ \texttt{exc}(\ell)$.

4.3.2. Catching Exceptions

Real-world languages do not just throw exceptions, they also *handle* them. To implement exception handling in ITrees, we use a common monadic interpretation of exceptions: we allow programs to return either a standard return value or an exception. Specifically, we move from an ITree of type $\texttt{itree}\ (\ \texttt{excE}\ Err \oplus E)\ R$ to one of type $\texttt{itree}\ (\ \texttt{excE}\ Err \oplus E)\ (Err \oplus R)$ using `interp` to lift the following h_{exc} event handler to the entire ITree, as described in Section 4.2.3.

$$\begin{aligned}
h_{exc} &: \forall A,\ (\texttt{excE}\ Err \oplus E)\ A \to \texttt{itree}\ (\ \texttt{excE}\ Err \oplus E)\ (Err \oplus A) \\
h_{exc}(\texttt{inl}(\texttt{exc}(e))) &\coloneqq \texttt{ret}\ \texttt{inl}(e) \\
h_{exc}(\texttt{inr}(e)) &\coloneqq x \leftarrow \texttt{trigger}\ \texttt{inr}(e); \texttt{ret}\ \texttt{inr}(x)
\end{aligned}$$

Even though the resulting ITree cannot have exception events, we still assign it a type that allows them so it can cleanly compose with ITrees that do contain exception events. This choice allows monadic bind to apply exception handlers—which may themselves contain exception events—to any left values (exceptions) while leaving right values (normal returns) unmodified. The result is the following exception-handling combinator, where $\texttt{case}\ k_1\ k_2$ chooses the continuation k_1 or k_2 if the return value is `inl` or `inr`, respectively.

$$\texttt{trycatch}(t, k_c) \coloneqq \texttt{interp}\ h_{exc}\ t \gg\!= \texttt{case}\ k_c\ \texttt{ret}$$

This `trycatch` combinator has a straightforward metatheory. In particular, we show how it interacts with the constructors of ITrees, allowing proof engineers to reason about `trycatch` without using manual coinduction.

Theorem 7. *The* ***trycatch*** *operator satisfies the following equivalences:*

$$
\begin{aligned}
E \vdash \textit{trycatch}(\texttt{ret}\ r, k_c) &\approx_= \texttt{ret}\ r \\
E \vdash \textit{trycatch}(\textit{Tau}(t), k_c) &\approx_= \textit{trycatch}(t, k_c) \\
E \vdash \textit{trycatch}(\texttt{Vis}\ \textit{inr}(a)\ k, k_c) &\approx_= \texttt{Vis}\ \textit{inr}(a)\ \lambda x.\textit{trycatch}(k(x), k_c) \\
E \vdash \textit{trycatch}(\texttt{Vis}\ \textit{inl}(\textit{exc}(\varepsilon))\ k, k_c) &\approx_= k_c(\varepsilon)
\end{aligned}
$$

Finally, the `trycatch` operator provides a simple denotation of IMP's try-catch blocks:

$$[\![\mathsf{try}\ \{c_1\}\ \mathsf{catch}\ \{c_2\}]\!]_c = \texttt{trycatch}([\![c_1]\!]_c, \lambda\_.\ [\![c_2]\!]_c)$$

4.4. Indistinguishability of Interaction Trees

To leverage the common semantic domain of ITrees to guarantee the security of a toolchain, we define our indistinguishability relation purely semantically. Intuitively, for programs to be indistinguishable, they must return indistinguishable results and have indistinguishable interactions with their environments.

Since return values can be arbitrary types, we follow `eutt` by parameterizing indistinguishability over a *return relation* $\mathcal{R}$. For indistinguishability, $\mathcal{R}$ describes when two values *appear* to be the same to the adversary. For example, consider a program that outputs a pair (a, b) where a is visible to Alice and b is visible to Bob, but not vice versa. The values $(1, 1)$ and $(1, 2)$ are not equal, but they are indistinguishable from Alice's perspective, as she can only see the first element. We can represent Alice's view of the output with a relation $\mathcal{R}_{\text{Alice}}$ defined by $\mathcal{R}_{\text{Alice}}((a, b), (a', b')) \iff a = a'$.

We could simply use `eutt` with a return relation $\mathcal{R}$ modeling indistinguishability. The resulting relation would model an adversary who can only see some part of the program's output, but it would require the two programs to interact with the environment in precisely the same way. Most settings, however, allow adversaries to see some interactions, but not others. For example, memory may be partitioned into a protected heap the adversary can never see, and an unprotected heap that it can

$$[\textsc{Ret}]\ \frac{\mathcal{R}(r_1, r_2)}{E;\rho \vdash_{ps} \texttt{ret}\ r_1 \approx^{\ell}_{\mathcal{R}} \texttt{ret}\ r_2} \qquad [\textsc{TauTau}]\ \begin{array}{c} E;\rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2 \\ \hline\hline E;\rho \vdash_{ps} \texttt{Tau}(t_1) \approx^{\ell}_{\mathcal{R}} \texttt{Tau}(t_2) \end{array}$$

$$[\textsc{PubVis}]\ \begin{array}{c} \forall a, E;\rho \vdash_{ps} k_1(a) \approx^{\ell}_{\mathcal{R}} k_2(a) \\ e : E\ A \qquad \rho(e) \sqsubseteq \ell \\ \hline\hline E;\rho \vdash_{ps} \texttt{Vis}\ e\ k_1 \approx^{\ell}_{\mathcal{R}} \texttt{Vis}\ e\ k_2 \end{array} \qquad [\textsc{TauL}]\ \frac{E;\rho \vdash_{ps} \texttt{Tau}(t_1) \approx^{\ell}_{\mathcal{R}} t_2}{E;\rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2}$$

$$[\textsc{TauR}]\ \frac{E;\rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} \texttt{Tau}(t_2)}{E;\rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2}$$

Figure 4.3: Inference rules for indistinguishability, where all events are visible

see at all times. Reasoning about security when some events are visible and others are not requires changing `eutt` to account for what the adversary can observe.

4.4.1. Secure Equivalence Up-To Taus

Our indistinguishability relation is called *secure equivalence up-to tau* or `seutt`. In addition to a return relation, `seutt` is also parameterized by a label ℓ, representing what the adversary can see, and a *sensitivity function* ρ that maps events to labels, representing who may observe which events. Intuitively, two ITrees are related by `seutt` if the environment interactions appear the same to an adversary who can see events only at or below label ℓ, and the return values are related by $\mathcal{R}$. We write the relation as $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2$.

Notably, we base the relation on `eutt`, which makes it progress sensitive. Recall from Section 4.2.1 that progress-sensitive noninterference allows any adversary to determine if a program silently diverges, and is often prohibitively expensive to enforce. We will also define `pi-seutt`, a progress-insensitive version of `seutt`, in Section 4.4.3. The judgments take the same form, so we annotate the turnstile with a subscript *ps* or *pi* to distinguish them visually.

For presentation, we separate the rules for `seutt` into three groups: rules covering returns, `Taus`, and public events (Figure 4.3), rules covering secret events that do not halt the program (Figure 4.4), and rules covering secret halting events (Figure 4.5).

$$[\textsc{PrivVisTau}]\ \frac{\begin{array}{c}\forall a, E; \rho \vdash_{ps} k(a) \approx^{\ell}_{\mathcal{R}} t \qquad e : E\ A \\ \neg \mathit{empty}(A) \qquad \rho(e) \not\sqsubseteq \ell\end{array}}{E; \rho \vdash_{ps} \texttt{Vis}\ e\ k \approx^{\ell}_{\mathcal{R}} \texttt{Tau}(t)} \qquad [\textsc{PrivVisIndL}]\ \frac{\begin{array}{c}\forall a, E; \rho \vdash_{ps} k(a) \approx^{\ell}_{\mathcal{R}} t \qquad e : E\ A \\ \neg \mathit{empty}(A) \qquad \rho(e) \not\sqsubseteq \ell\end{array}}{E; \rho \vdash_{ps} \texttt{Vis}\ e\ k \approx^{\ell}_{\mathcal{R}} t}$$

$$[\textsc{PrivVisVis}]\ \frac{\begin{array}{c}\forall (a\!:\!A)(b\!:\!B), E; \rho \vdash_{ps} k_1(a) \approx^{\ell}_{\mathcal{R}} k_2(b) \qquad e_1 : E\ A \qquad e_2 : E\ B \\ \rho(e_1) \not\sqsubseteq \ell \qquad \rho(e_2) \not\sqsubseteq \ell \qquad \neg \mathit{empty}(A) \qquad \neg \mathit{empty}(B)\end{array}}{E; \rho \vdash_{ps} \texttt{Vis}\ e_1\ k_1 \approx^{\ell}_{\mathcal{R}} \texttt{Vis}\ e_2\ k_2}$$

Figure 4.4: Inference rules for indistinguishability, where events are not visible but answer types are inhabited

Public Events and Returns. When an adversary is able to see an event, indistinguishability acts just like weak bisimulation. The rules, found in Figure 4.3, are almost identical to the rules of `eutt`, but with the added requirement that any visible event be visible to the adversary. That is, we require $\rho(e) \sqsubseteq \ell$ in PubVis.

It might seem mysterious that we *require* the event to be visible in PubVis. But allowing this rule to apply no matter the visibility would allow the adversary too much power, since they would know that the same result is returned on both sides of the equivalence. As we will see, the rule for invisible events is stricter. We will also see how this strictness, when proving a program p indistinguishable from itself, corresponds to proving that the behavior of p does not differ in runs in *low-equivalent* environments. If we were to allow high events in PubVis, this would allow our proof to only consider the behavior of p in one environment, breaking our correspondence with information-flow security.

Private Events With Responses. When the adversary is *unable* to view an event, `seutt` cannot act like `eutt`. In this case, the rules are designed to formalize two intuitions. If the computation continues after a secret event, we should treat the event like a `Tau`, since the adversary cannot observe either. If the event halts the computation, the event should be equivalent to a silently nonterminating computation.

The rules in Figure 4.4, along with symmetric analogues of PrivVisTau and PrivVisIndL, handle the case where the event allows computation to continue—that is, the event's answer type is inhabited.

The first rule, PrivVisTau, relates a private event `Vis` e k with a $\texttt{Tau}(t)$. In addition to requiring the event to be secret ($\rho(e) \not\sqsubseteq \ell$) and have a non-empty answer type ($\neg \mathit{empty}(A)$), it also requires the continuation k produce an ITree indistinguishable from t for *every* possible response. This requirement ensures that the adversary's future observations cannot depend on the response to the private event. Note that the requirement that A be non-empty does more than just specify when the rule applies. Without it, a private halting event would trivially satisfy this condition, allowing it to relate to any ITree with a τ in front. Since the adversary can determine when a program has halted, they should be able to distinguish, for example, a program that throws a private exception from a program which, after a `Tau`, prints to a public channel. This rule ensures that this intuition holds.

PrivVisIndL is analogous to TauL, but for secret events instead of `Tau` nodes. This rule has the same premises as PrivVisTau for the same reasons. Moreover, it only removes a node from the head of one ITree, not both. As with the definition of `seutt`, TauL, and TauR, we therefore make PrivVisIndL inductive, not coinductive, to avoid relating a infinite stream of secret events to all other ITrees.

Finally, PrivVisVis removes a private event from the head of both sides of the relation. As with the previous rules, we require both events to be private and have non-empty answer types. This time, we require the continuations of the two events to be indistinguishable for every possible response *of both events separately.* This requirement formalizes the idea that the adversary should not be able to distinguish the program's behavior on any pair of secret responses.

To see the power of this rule, consider whether an adversary who can see l but not h would find the following ITrees indistinguishable from themselves:

$$t_{\text{sec}} \triangleq \begin{array}{l} x \leftarrow \texttt{trigger get}(l); \\ y \leftarrow \texttt{trigger get}(h); \\ \texttt{trigger set}(h, x + y) \end{array} \qquad t_{\text{insec}} \triangleq \begin{array}{l} x \leftarrow \texttt{trigger get}(l); \\ y \leftarrow \texttt{trigger get}(h); \\ \texttt{trigger set}(l, x + y) \end{array}$$

One would hope that t_{sec} would be indistinguishable from itself, while t_{insec} would not be, and indeed

$$[\textsc{EmpVisTau}]\ \begin{array}{c} E;\rho \vdash_{ps} \texttt{Vis}\ e\ k \approx^{\ell}_{\mathcal{R}} t \\ e : E\ A \qquad \mathit{empty}(A) \\ \rho(e) \not\sqsubseteq \ell \\ \hline\hline E;\rho \vdash_{ps} \texttt{Vis}\ e\ k \approx^{\ell}_{\mathcal{R}} \texttt{Tau}(t) \end{array} \qquad [\textsc{EmpVisVisL}]\ \begin{array}{c} \forall b, E;\rho \vdash_{ps} \texttt{Vis}\ e_1\ k_1 \approx^{\ell}_{\mathcal{R}} k_2(b) \\ e_1 : E\ A \qquad e_2 : E\ B \\ \mathit{empty}(A) \qquad \rho(e_1) \not\sqsubseteq \ell \qquad \rho(e_2) \not\sqsubseteq \ell \\ \hline\hline E;\rho \vdash_{ps} \texttt{Vis}\ e_1\ k_1 \approx^{\ell}_{\mathcal{R}} \texttt{Vis}\ e_2\ k_2 \end{array}$$

Figure 4.5: Inference rules for indistinguishability, where events are halting and not visible

that is the case. To (attempt to) prove that either tree is equivalent to itself, we walk through each ITree. Since l is visible, so is $\texttt{get}(l)$, so PubVis applies and requires only that each possible value of x produce an ITree that is indistinguishable from itself. Because h is secret, the adversary should not be able to observe or infer its value, so we must use PrivVisVis to remove $\texttt{get}(h)$. PrivVisVis requires that, for all possible *pairs* of values y_1, y_2, the continuations be indistinguishable. Thus in t_{sec}, $\texttt{trigger set}(h, x + y_1)$ must be indistinguishable from $\texttt{trigger set}(h, x + y_2)$. Since h is secret, so are the `set` events, so PrivVisVis can remove them even when they differ. After removing `set`, the remaining continuation always produces `ret` (), so Ret finishes the proof.

However, in t_{insec}, PrivVisVis does not apply to the `set` events since l is visible. PubVis only relates ITrees starting with the same event, but $\texttt{set}(l, x + y_1) \neq \texttt{set}(l, x + y_2)$ when $y_1 \neq y_2$. As a result, no rule applies after removing $\texttt{get}(h)$, so the adversary can distinguish t_{insec} from itself. In other words, t_{insec} is, indeed, insecure.

Private Halting Events. Finally, we turn to the case where an event the adversary cannot see halts the computation. In this case, the adversary should be unable to tell that the event took place, and therefore should not be able to distinguish a program with a secret halt from a program that never terminates. However, the adversary should still be able to distinguish it from any ITree that contains an event the adversary can see.

This intuition means that a private halting event should not be treated like a `Tau`, as a private non-halting event is, but rather should be indistinguishable from *an infinite stream of* ***`Taus`****.* We formalize this approach with the rules presented in Figure 4.5 along with their symmetric analogues. EmpVisTau peels a single `Tau` off the right ITree, leaving the private halting event on the left unmodified. EmpVisVisL does the same for a private event.

There are two interesting properties about these rules. First, unlike the rules for private events and `Tau`s that leave one side of the equivalence unmodified, these rules are coinductive, not inductive. This choice allows us to relate a private halting event to an entire nonterminating program, as long as that program has no public events. Indeed, no rule allows us to remove a private halting event, as there would be nothing left to compare. Second, EmpVisVisL has no requirement that B, the answer type of the not-necessarily-halting event, be non-empty. This choice avoids the need to explicitly handle the case where both ITrees contain private halts. If B is non-empty, then EmpVisVisL treats the event as a `Tau`. If B is empty, then the first premise of the rule is trivially satisfied, which is desirable, as in that case both ITrees begin with a private halt event and should be equivalent.

4.4.2. The Metatheory of Indistinguishability

The `seutt` relation captures intuitions about when two ITrees are indistinguishable to some adversary, but using it requires a delicate mix of induction and coinduction. To both demonstrate the power of our definition and better support verification, we also develop a library of metatheory for indistinguishability. This library supports reasoning about cross-language toolchains without the need for explicit coinduction, as we will see when we verify the correctness of a security type system and compiler for Imp (Sections 4.5 and 4.6, respectively).

Indistinguishability as a PER Model. Recall from Section 4.2.1 that Sabelfeld and Sands (2001) argue for indistinguishability forming a partial equivalence relation (PER). It would be nice if `seutt` always formed a PER, but because it is parameterized on an arbitrary relation for return values, that is not always the case. Instead, we prove generalized versions of transitivity and reflexivity. In particular, if we let $\overset{\leftrightarrow}{\mathcal{R}}$ denote the reverse relation of $\mathcal{R}$—that is, $\overset{\leftrightarrow}{\mathcal{R}}(x, y) \overset{\triangle}{\iff} \mathcal{R}(y, x)$—then the following theorems hold.

Theorem 8. *For all $\mathcal{R}$, E, ρ, and ℓ, if $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2$, then $E; \rho \vdash_{ps} t_2 \approx^{\ell}_{\overset{\leftrightarrow}{\mathcal{R}}} t_1$.*

Theorem 9. *If* $E;\rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$ *and* $E;\rho \vdash_{ps} t_2 \approx^{\ell}_{\mathcal{R}_2} t_3$ *then* $E;\rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}_1 \circ \mathcal{R}_2} t_3$*.*

Note that if $\mathcal{R}$ is symmetric, then $\mathcal{R} = \overset{\leftrightarrow}{\mathcal{R}}$, and if $\mathcal{R}$ is transitive, then $\mathcal{R} \circ \mathcal{R} \subseteq \mathcal{R}$. These properties allow us to prove the following corollary.

Corollary 1. *If* $\mathcal{R}$ *is a PER, then so is* $E;\rho \vdash_{ps} - \approx^{\ell}_{\mathcal{R}} -$ *for any* E*,* ρ*, and* ℓ*.*

ITree Combinators. ITrees are often defined using the combinators from Section 2, making it important to understand how indistinguishability interacts with those combinators. The definition of `seutt` directly describes how to relate simple programs defined using only `ret` and `trigger`, but they say nothing about larger ITrees built using bind and iteration.

Bind allows for the sequential composition of programs. We would like indistinguishable programs t_1 and t_2 followed by indistinguishable continuations k_1 and k_2 to compose into larger indistinguishable programs $t_1 \gg= k_1$ and $t_2 \gg= k_2$. The following theorem says that this result holds whenever the relation $\mathcal{R}_1$, securely relating t_1 and t_2, puts enough constraints on their possible outputs to ensure that k_1 and k_2 are always securely related at some relation $\mathcal{R}_2$.

Theorem 10. *If* $E;\rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$ *and for all values* a, b*,* $\mathcal{R}_1(a,b)$ *implies* $E;\rho \vdash_{ps} k_1(a) \approx^{\ell}_{\mathcal{R}_2} k_2(b)$*, then* $E;\rho \vdash_{ps} t_1 \gg= k_1 \approx^{\ell}_{\mathcal{R}_2} t_2 \gg= k_2$*.*

Iteration represents loops, which have two parts: an initial value, and a body that produces a value from the previous value. Indistinguishable initial values paired with indistinguishable bodies produce indistinguishable loops, as we can see in the following theorem.

Theorem 11. *If* $\mathcal{R}_1(a_1, b_1)$ *and, for any* a, b*,* $E;\rho \vdash_{ps} k_1(a) \approx^{\ell}_{\texttt{caseR}(\mathcal{R}_1,\mathcal{R}_2)} k_2(b)$ *whenever* $\mathcal{R}_1(a,b)$*, then* $E;\rho \vdash_{ps}$ `iter` $k_1\ a_1 \approx^{\ell}_{\mathcal{R}_2}$ `iter` $k_2\ b_1$*.*

This rule is conceptually similar to a loop invariant from a Hoare-style logic. $\mathcal{R}_1$ is a property that is initially true and is preserved on each iteration except the final one, while the final iteration guarantees that $\mathcal{R}_2$ holds. The $\texttt{caseR}(\mathcal{R}_1, \mathcal{R}_2)$ function lifts two relations to a single relation over sum types such that $\mathcal{R}_1$ is applied to two left values, $\mathcal{R}_2$ is applied to two right values, and no other combination is related.

Relationship with Equivalence Up-To Taus. Recall that weak bisimulation of ITrees requires two ITrees to contain the same pattern of interaction with their environment. Our notion of indistinguishability assumes that adversaries distinguish programs purely based on their interactions with the environment. One would thus expect that combining `eutt` with indistinguishability should result in indistinguishability. The following theorem shows this to be the case.

Theorem 12 (Mixed Transitivity)**.** *If both $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$ and $E \vdash t_2 \approx_{\mathcal{R}_2} t_3$ then we can conclude that $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}_1 \circ \mathcal{R}_2} t_3$.*

This is a very powerful theorem. In particular, many program transformations preserve equality. That is, they take source programs with equivalent-up-to-taus ITree representations to target programs with the same property. Mixed transitivity tells us that compilers built from such transformations also preserve indistinguishability. For instance, since noninterference—the security property we are ultimately considering—is defined as a program being indistinguishable from itself, mixed transitivity supports a very simple proof that the compiler in Section 4.6 preserves noninterference. While this result might be surprising, it reflects the utility of ITrees and indistinguishability. By looking at which labels can distinguish an ITree from itself, we can discover where leaks are possible.

4.4.3. Progress-Insensitive Indistinguishability

The type systems that enforce progress-sensitive noninterference are extremely restrictive. Thus, information-flow control literature mostly studies progress-*insensitive* type systems. These type systems enforce noninterference against adversaries who cannot see when a program has begun to silently loop forever. Intuitively, such adversaries believe that silently looping programs could break out of their loops at any moment, and so do not distinguish them from programs which have produced visible events.

In order to support such reasoning, we introduce `pi-seutt`, a progress-insensitive version of indistinguishability for ITrees.

Definition 19 (`pi-seutt`)**.** *The relation* `pi-seutt`*, the progress-insensitive version of indistinguishability, is defined by modifying the definition of* `seutt` *by completely removing the rules for halting events (all rules in Figure 4.5) and making every other rule coinductive (this modifies* TauL *and* TauR *in Figure 4.3 as well as* PrivVisIndL *in Figure 4.4 and its symmetric counterpart).*

This relation is strictly more permissive than `seutt`, since it relates every ITree to silently diverging ITrees and private halts. These facts can be formalized in the following theorems:

Theorem 13. *If* $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2$ *then* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}} t_2$*.*

Theorem 14. *Given any ITree* t*,* $E; \rho \vdash_{pi} t_{spin} \approx^{\ell}_{\mathcal{R}} t$*.*

Theorem 15. *Given any ITree* t*, if* e *is a halting event, then* $E; \rho \vdash_{pi} \texttt{Vis}\ e\ k \approx^{\ell}_{\mathcal{R}} t$*.*

Just as with the progress-sensitive version of indistinguishability, we can show that indistinguishability plays well with the usual ITree combinators. This allows us to prove ITrees indistinguishable in many cases without resorting to hand-rolled coinduction.

Theorem 16. *If* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$ *and* $E; \rho \vdash_{pi} k_1(a) \approx^{\ell}_{\mathcal{R}_2} k_2(b)$ *whenever* $\mathcal{R}_1(a, b)$*, then* $E; \rho \vdash_{pi} t_1 \gg= k_1 \approx^{\ell}_{\mathcal{R}_2} t_2 \gg= k_2$*.*

Theorem 17. *If* $\mathcal{R}_1(a_1, a_2)$ *and for any* a, a'*,* $E; \rho \vdash_{pi} k_1(a) \approx^{\ell}_{\texttt{caseR}(\mathcal{R}_1, \mathcal{R}_2)} k_2(a')$ *whenever* $\mathcal{R}_1(a, a')$*, then* $E; \rho \vdash_{pi} \texttt{iter}\ k_1\ a_1 \approx^{\ell}_{\mathcal{R}_2} \texttt{iter}\ k_2\ a_2$*.*

Moreover, mixed transitivity again holds, allowing for simple proofs of compiler safety:

Theorem 18 (Mixed Transitivity)**.** *If both* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$ *and* $E \vdash t_2 \approx_{\mathcal{R}_2} t_3$ *then we get* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}_1 \circ \mathcal{R}_2} t_3$*.*

Progress-insensitive indistinguishability behaves differently from the progress-sensitive sibling version in one important way: it does not form a PER. Because it relates a diverging ITree to every other ITree, `pi-seutt` is not transitive. This is not surprising, since progress-insensitive indistinguishability is not a PER (Vassena et al., 2019; Rajani and Garg, 2018; Gregersen et al., 2021). It does, however, retain generalized symmetry, and a weakened but still-useful version of generalized transitivity:

Theorem 19. *If* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}} t_2$ *then* $E; \rho \vdash_{pi} t_2 \approx^{\ell}_{\overset{\leftrightarrow}{\mathcal{R}}} t_1$.

Theorem 20. *If* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$, $E; \rho \vdash_{pi} t_2 \approx^{\ell}_{\mathcal{R}_2} t_3$, *and* t_2 *converges along all paths, then* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}_1 \circ \mathcal{R}_2} t_3$.

An ITree is considered convergent if it is either a `ret` , a `Tau` followed by a convergent ITree, or a non-halting event followed by a continuation that converges for any input.

Unlike progress-sensitive indistinguishability, we can easily show that loops produce no events that are observable to some adversary at ℓ via `pi-seutt`. Suppose that we want to show that `iter` body a_0 emits no events that are observable to some adversary at ℓ. We can do so by showing that `iter` body a_0 and `ret` b are indistinguishable with some return relation $\mathcal{R}$. This shows that the body of the loop both emits no observable events and, if the loop terminates, it returns a value c where $\mathcal{R}(c, b)$. Importantly, we have not made any statement about whether the loop terminates; we have merely said that it will not produce events, regardless of its termination behavior. We formalize this in the following theorem:

Theorem 21. *For any relation* $\mathcal{R}_{inv}$, *if*

$$\mathcal{R}_{inv}(a_0, b) \quad \textit{and} \quad \forall a,\, \mathcal{R}_{inv}(a, b) \implies E; \rho \vdash_{pi} \textit{body } a \approx^{\ell}_{\texttt{leftcase}(\mathcal{R}_{inv}, \mathcal{R})} \texttt{ret } b,$$

then $E; \rho \vdash_{pi}$ `iter` *body* $a_0 \approx^{\ell}_{\mathcal{R}}$ `ret` b, *where the relation* `leftcase` *is defined as follows:*

$$\texttt{leftcase}(\mathcal{R}_1, \mathcal{R}_2)(\boldsymbol{inl}(a), b) = \mathcal{R}_1(a, b) \qquad \texttt{leftcase}(\mathcal{R}_1, \mathcal{R}_2)(\boldsymbol{inr}(a), b) = \mathcal{R}_2(a, b)$$

4.4.4. Noninterference and Interpretation

Recall from Section 4.2.1 that we can define noninterference using an indistinguishability relation on programs by saying that a program is noninterfering if it is related to itself—given indistinguishable inputs, it will produce indistinguishable computations. We could define noninterference on

ITrees using `seutt` (or `pi-seutt`), as they provide such indistinguishability relations by design. This approach produces a sensible definition, but one that assumes an extremely strong adversary.

Consider the following Imp program, where the h_is have label ℓ_h and the l_is have label ℓ_l:

$$\mathsf{if}\ (h_1 = 0)\ \mathsf{then}\ \{h_2 \coloneqq l_1\}\ \mathsf{else}\ \{h_2 \coloneqq l_2\}$$

Since the program writes only to secret variables, intuitively it seems secure. However, according to `seutt`, it is not related to itself at ℓ_l since reading from l_1 and l_2 produce different `get` events with label ℓ_l. All adversaries have the power to observe *reads* of public state, not just writes.

The visibility of public read events is not the only problem. Using just `seutt` also means a computation cannot publicly depend on the result of reading a secret variable, even if a public value were written to that variable. For instance, the following program would also be considered insecure:

$$h \coloneqq l\,;\, \mathsf{print}(\ell_l, h)$$

If h cannot change between assignments, this program is intuitively secure, but `seutt` at ℓ_l requires $\mathsf{print}(\ell_l, h)$ to produce the same output regardless of the value of h, which it clearly does not.

On uninterpreted ITrees, `seutt` models a system where both reads and writes are visible to anyone who can see the variable, and the value of a secret variable may silently change between a read and a write. This model makes perfect sense in some contexts—like distributed computation (Liu et al., 2017)—but we usually consider weaker adversaries.

We can remove these assumptions and model a weaker adversary by interpreting state, as we discussed in Section 4.2.3. Interpreting these programs would result in two meta-level functions (i.e., Coq functions) which take a state as input and produce an ITree returning an output state. For example in Section 4.2.3, we define the semantics of an Imp program c as an interpreted ITree—that is, as a function from states to ITrees—not as a single ITree with state events. We thus adjust our notions of indistinguishability and noninterference to account for this semantic construct.

Intuitively, we start with a family of relations $\mathcal{R}_{S,\ell}$ that describes when states are indistinguishable to an adversary at level ℓ and use it to define the following observational equivalence. For technical reasons, we require $\mathcal{R}_{S,\ell}$ to be an equivalence relation at all labels. For Imp, we use a relation $\cong^{\ell}_{\Gamma}$ which only requires states to agree on a variable x if the label of x flows to ℓ.

Definition 20 (Stateful Indistinguishability)**.** *Two stateful computations p_1 and p_2 are* px-statefully indistinguishable under $\mathcal{R}_{S,\ell}$ and $\mathcal{R}$ at label ℓ *if, for every pair of states σ_1 and σ_2 such that $\mathcal{R}_{S,\ell}(\sigma_1, \sigma_2)$,*

$$E; \rho \vdash_{px} p_1\ \sigma_1 \approx^{\ell}_{\mathcal{R}_{S,\ell} \times \mathcal{R}} p_2\ \sigma_2$$

$$\textit{where } \mathcal{R}_{S,\ell} \times \mathcal{R}((\sigma_1', a_1), (\sigma_2', a_2)) \overset{\triangle}{\iff} \mathcal{R}_{S,\ell}(\sigma_1', \sigma_2') \textit{ and } \mathcal{R}(a_1, a_2)$$

As described above, stateful indistinguishability with $\cong^{\ell}_{\Gamma}$ defines security against an adversary who can observe public writes, but not secret writes or secret reads. This indistinguishability relation leads to a much more common definition of noninterference, and it is the one we will use in our case studies in Sections 4.5 and 4.6.

Definition 21 (Noninterference)**.** *A stateful computation is* px-noninterfering *with state relations $\mathcal{R}_{S,\ell}$ and return relation $\mathcal{R}$ if, given any label ℓ, it is px-statefully indistinguishable from itself under state relation family $\mathcal{R}_{S,\ell}$ and return relation $\mathcal{R}$.*

4.5. Security Sensitive Type Systems For Imp

To see how to use this theory of indistinguishability and ITrees, we now provide an information-security guarantee for an example toolchain for Imp. We begin by verifying two information-flow type systems, and proceed with a simple compiler in Section 4.6. The two notions of noninterference—progress sensitive and progress insensitive—require slightly different type systems, so we use our ITrees-based semantics to formally verify that both enforce their respective notions of noninterference. As is common in such type systems, we assume $\mathcal{L}$ forms a join semilattice with a unique least element $\bot$ representing “completely public.”

$$\frac{\Gamma(x) \sqsubseteq \ell}{\Gamma \vdash x : \ell} \qquad\qquad \frac{}{\Gamma \vdash n : \ell} \qquad\qquad \frac{\Gamma \vdash e_1 : \ell_1 \qquad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash e_1 \odot e_2 : \ell_1 \sqcup \ell_2}$$

Figure 4.6: Typing rules for expressions in security-typed Imp.

Shared Typing Rules

$$[\textsc{Skip}]\ \frac{}{\Gamma; pc \vdash_{px} \mathsf{skip} \diamond \bot} \qquad\qquad [\textsc{If}]\ \frac{\begin{array}{c}\Gamma \vdash_{px} e : \ell \\ \Gamma; pc \sqcup \ell \vdash_{px} c_1 \diamond \ell_{ex} \qquad \Gamma; pc \sqcup \ell \vdash_{px} c_2 \diamond \ell'_{ex}\end{array}}{\Gamma; pc \vdash_{px} \mathsf{if}\ (e)\ \mathsf{then}\ \{c_1\}\ \mathsf{else}\ \{c_2\} \diamond \ell_{ex} \sqcup \ell'_{ex}}$$

$$[\textsc{Assign}]\ \frac{\begin{array}{c}\Gamma \vdash_{px} e : \ell \\ pc \sqcup \ell \sqsubseteq \Gamma(x)\end{array}}{\Gamma; pc \vdash_{px} x \coloneqq e \diamond \bot} \qquad\qquad [\textsc{Seq}]\ \frac{\Gamma; pc \vdash_{px} c_1 \diamond \ell_{ex} \qquad \Gamma; pc \sqcup \ell_{ex} \vdash_{px} c_2 \diamond \ell'_{ex}}{\Gamma; pc \vdash_{px} c_1\ ;\ c_2 \diamond \ell_{ex} \sqcup \ell'_{ex}}$$

$$[\textsc{Try}]\ \frac{\Gamma; pc \vdash_{px} c_1 \diamond \ell_{ex} \qquad \Gamma; pc \sqcup \ell_{ex} \vdash_{px} c_2 \diamond \ell'_{ex}}{\Gamma; pc \vdash_{px} \mathsf{try}\ \{c_1\}\ \mathsf{catch}\ \{c_2\} \diamond \ell'_{ex}} \qquad\qquad [\textsc{Print}]\ \frac{\Gamma \vdash_{px} e : \ell \qquad pc \sqcup \ell \sqsubseteq \ell'}{\Gamma; pc \vdash_{px} \mathsf{print}(e, \ell') \diamond \bot}$$

Progress-Sensitive Typing Rules

$$[\textsc{While-PS}]\ \frac{\Gamma \vdash_{ps} e : \bot \qquad \Gamma; \bot \vdash_{ps} c \diamond \bot}{\Gamma; \bot \vdash_{ps} \mathsf{while}\ (e)\ \mathsf{do}\ \{c\} \diamond \bot}$$

$$[\textsc{Throw-PS}]\ \frac{}{\Gamma; \bot \vdash_{ps} \mathsf{throw}(\bot) \diamond \bot}$$

Progress-Insensitive Typing Rules

$$[\textsc{While-PI}]\ \frac{\Gamma \vdash_{pi} e : \ell \qquad \Gamma; pc \sqcup \ell \sqcup \ell_{ex} \vdash_{pi} c \diamond \ell_{ex}}{\Gamma; pc \vdash_{pi} \mathsf{while}\ (e)\ \mathsf{do}\ \{c\} \diamond \ell_{ex}}$$

$$[\textsc{Throw-PI}]\ \frac{pc \sqsubseteq \ell_{ex}}{\Gamma; pc \vdash_{pi} \mathsf{throw}(\ell_{ex}) \diamond \ell_{ex}}$$

Figure 4.7: Typing rules for commands in security-typed Imp.

4.5.1. Two Type Systems

Both type systems have two typing judgments: one for expressions and one for commands. The typing judgments for expressions take the form $\Gamma \vdash e : \ell$, where Γ is a map from variables to information flow labels, and ℓ is a label. The judgment says that e is well-typed and depends only on information at or below label ℓ. The typing rules for expressions, which are the same for both type systems, are presented in Figure 4.6.

The typing rules for commands are presented in Figure 4.7. As these rules differ between the progress-sensitive and progress-insensitive type systems, we annotate the turnstyles with ps for progress-sensitive rules, *pi* for progress-insensitive rules, and *px* for rules that are identical in both type systems.

The typing judgments for commands take the form $\Gamma; pc \vdash_{px} c \diamond \ell_{ex}$, where pc and ℓ_{ex} are information-flow labels. The pc label is a *program-counter label* that tracks the sensitivity of the control flow, while the second label ℓ_{ex} is an upper bound on the label of any exceptions c might raise. Note that the rules listed in Figure 4.7 do not include any way to type check an inlined Asm program. We address this concern in Section 4.5.3.

Program-counter labels are a standard technique to control *implicit information flows*—that is, information leaked by the control flow (Denning and Denning, 1977; Sabelfeld and Myers, 2003). For example, consider the following program where h has label ℓ_h and l has label ℓ_l with $\ell_h \not\sqsubseteq \ell_l$:

$$\mathsf{if}\ (h = 0)\ \mathsf{then}\ \{l \coloneqq 0\}\ \mathsf{else}\ \{l \coloneqq 1\}$$

While l is only ever explicitly set to constant values, its final value clearly depends on the secret h. The pc label allows us to detect and eliminate these flows by tracking the sensitivity of the control flow. Specifically, the If rule requires the condition's label to flow to the pc in each branch, and the Assign rule requires the pc to flow to the label of the variable being assigned. In the above example, the label of the condition $h = 0$ is ℓ_h, so If requires c_1 and c_2 to type check with a pc where $\ell_h \sqsubseteq pc$. Since $\Gamma(l) = \ell_l$, Assign requires $pc \sqsubseteq \ell_l$. Transitivity of $\sqsubseteq$ thus requires $\ell_h \sqsubseteq \ell_l$, which it does not, so the program correctly fails to type check.

Exceptions can affect the control flow of a program, and therefore can also cause implicit flows of information. Consider the following program.

$$\mathsf{if}\ (h = 0)\ \mathsf{then}\ \{\mathsf{throw}(\ell_h)\}\ \mathsf{else}\ \{\mathsf{skip}\}\ ; l \coloneqq 1$$

Much like the previous example, this program only assigns a constant to l, yet it still leaks the value of h. We use a standard technique (Myers, 1999; Pottier and Simonet, 2003) that relies on exception labels in the typing judgment. As previously mentioned, the exception label of a program c is an upper bound on the labels of any exception c might raise. To eliminate exception-based leaks, the Seq rule increases the *pc* label of the second command by the exception label of the first. The Try rule makes similar use of the exception label, increasing the *pc* in the catch block, as that command only executes if an exception is thrown.

The Skip rule is simple, as skip can never have an effect. Print produces a flow of information to an output channel labeled ℓ', so it checks that ℓ' may safely see both the expression being written and the fact that this command executed.

The rules for while loops and throw statements are different for the progress-sensitive and progress-insensitive type systems, so we handle them separately.

Progress-Sensitive While and Throw Rules. In a progress-sensitive setting, the adversary can observe nontermination. As a result, a program's termination behavior can only safely depend on completely public information. While-PS enforces this requirement in a standard, but highly restrictive way (Volpano and Smith, 1997): the loop condition and the *pc* of the context must both be the fully public label $\bot$. Moreover, any exceptions thrown in the body of the loop could also influence termination behavior, so those must be fully public as well.

Recall from Section 4.4 that a low observer cannot distinguish between an uncaught secret exception and an infinite loop. Thus non-public exceptions create the same implicit flows as while loops, so Throw-PS restricts exceptions in much the same way as While-PS restricts loops: everything must be fully public.

Progress-Insensitive While and Throw Rules. In a progress-insensitive setting, the adversary cannot see nontermination, so secrets can safely influence the termination behavior of a program. The While-PI rule therefore allows loops with any *pc*. Since both the loop condition and any exceptions the loop body throws influence whether the body is run, While-PI increases the *pc* in the loop body by both the loop guard label and the body's exception label.

For the same reason, Throw-PI is more permissive than its progress-sensitive counterpart. In particular, the label on the exception just needs to be at least as secret as the *pc* label.

4.5.2. Proving Security

Both type systems enforce their respective notions of noninterference (Definition 21). Unlike many existing proofs of noninterference, our proofs using ITrees proceed by simple induction over the syntax of Imp. This simplicity is made possible by the combination of two facts: our Imp semantics is given by simple induction using ITrees combinators, and those combinators interact with indistinguishability in predictable ways, as described by the metatheory of Section 4.4.

Type systems are inherently compositional: we are able to conclude that a program is secure knowing nothing about subprograms other than that they also type check. However, our semantic definition of noninterference is *not* fully compositional. To see this, consider the Imp program $p = l := h; \mathsf{throw}(\ell)$. This program updates the state in an insecure way, assigning a high-security value to a low-security variable, and then throws a low-security exception. In fully interpreted programs, the updated state is part of the return value, but adversaries cannot observe that return value if an exception is thrown (see Section 4.3), making p semantically secure. However, if we catch the exception, the adversary once again can see the effect of the assignment $l := h$. Thus, p does not compose securely.

In order for our type system to enforce security compositionally, it enforces two properties beyond noninterference. Each rules out programs which, like p above, are secure but do not compose securely. The first describes how state and exceptions interact in a secure setting, which will rule out the example program above. The second, called *confinement*, defines how effects are bound by the type system.

Interaction of Exceptions and State. Our first goal is to semantically rule out programs like p above, allowing us to reason compositionally about exception handlers. In order to do so, we need to reason about what state updates are performed before an exception is thrown. However, since in our semantics of Imp we interpret state events while leaving exceptions as ITree events, the result state of an Imp program is forgotten when an exception is thrown.

This correctly models our adversary, who cannot distinguish between private exceptions and silently diverging programs. But in order to achieve compositionality, we need to keep information about the final state before an exception is raised. We accomplish this with a condition on an alternative semantics for Imp programs. In this semantics, exceptions are interpreted into the standard sum type representation before state events are interpreted. This interpretation, $\texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ \ [\![c]\!]_c)$, is a stateful function that returns a final state along with either a result of type `unit` or the label of an exception. We can inspect this final state to ensure that the program always takes indistinguishable states to indistinguishable states.

We formalize this property as follows, where the relation $\cong^{\ell}_{\Gamma}$ requires that states agree on a variable x only when $\Gamma(x) \sqsubseteq \ell$, as in Section 4.4.4.

Definition 22 (Exceptions-and-State Property)**.** *A command c satisfies the* px–exceptions-and-state property *if* $\texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ \ [\![c]\!]_c)$ *is statefully indistinguishable from itself under* $\cong^{\ell}_{\Gamma}$ *and* $\top$ *at every label* ℓ*.*

Note the use of $\top$ as the output relation means we ignore whether or not c threw an exception, while we still ensure that the final states are indistinguishable. Ignoring this information in this property is acceptable because it is captured by our standard noninterference condition.

Confinement. Even with the exceptions-and-state property, implicit flows, like the motivating our use of *pc* labels, can still break compositionality. Confinement fixes this.

In the typing judgment for commands, the *pc* and ℓ_{ex} labels are both designed to constrain effects. If a command type checks with *pc* and ℓ_{ex}, it should have no effects visible *below pc* and no (uncaught) exceptions *above* ℓ_{ex}. Semantically, a program has no visible effects below *pc* if, for any label ℓ where $pc \not\sqsubseteq \ell$, it is indistinguishable from `skip`. For any uncaught exception terminating a ITree, we simply check that the exception's label flows to ℓ_{ex}. We formalize this idea into the following property called *confinement*.

Definition 23 (Confinement)**.** *A command c is* px-confined to *pc* with ℓ_{ex} exceptions*, if, for all* *labels ℓ such that $pc \not\sqsubseteq \ell$, the following conditions hold.*

1. *c is indistinguishable from* **skip** *at ℓ:* $\texttt{interp}\ h_{prog}\ \llbracket c \rrbracket_c$ *and* $\texttt{interp}\ h_{prog}\ \llbracket \textsf{skip} \rrbracket_c$ *are px-statefully indistinguishable under $\cong^{\ell}_{\Gamma}$ and $=$ at ℓ.*

2. *c makes no modifications to the state visible at ℓ:* $\texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ \llbracket c \rrbracket_c)$ *and* $\texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ \llbracket \textsf{skip} \rrbracket_c)$ *are px-statefully indistinguishable under $\top$ and $=$ at ℓ.*

3. *For all initial state heap states h and register states r where c throws an exception, the label of that exception flows to ℓ_{ex}:*

$$E \vdash (\texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ \llbracket c \rrbracket_c))(r, h) \approx_{=} \texttt{ret}\ (r', h', \textit{\textbf{inr}}(\ell'_{ex})) \implies \ell'_{ex} \sqsubseteq \ell_{ex}$$

Together, these definitions restrict programs to those that compose securely, as required by the type system. With this compositionality property, we can prove that our type system enforces the conjunction of all three properties.

Theorem 22. *If $\Gamma; pc \vdash_{px} c \diamond \ell_{ex}$, then c is px-noninterfering (Definition 21), satisfies the px–exceptions-and-state property, and is px-confined to pc with ℓ_{ex} exceptions.*

4.5.3. Semantic Typing and Inline Asm

Both type systems above enforce security, but are highly conservative. Many secure programs fail to type check, notably including any secure program with inlined Asm. To support our goal of cross-language security reasoning and address this concern without the need to introduce a type system for Asm, we provide a *semantic typing* (Jung et al., 2015) rule.

One would hope that the three conditions discussed above would be sufficient. However, the possibility of undefined Asm behavior (see Section 4.2.4) necessitates an additional condition. We thus introduce the notion of *inline validity*, which requires inlined Asm to depend only on the initial heap state, not the initial register state, thereby ruling out undefined behavior.

Definition 24 (Inline Validity)**.** *An ASM program a is* inline-valid *if, given any two register states*

r_1 and r_2, and any heap states h, runs with (r_1, h) and (r_2, h) produce the same changes to the heap. That is, if $p = \texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ [\![a]\!]_{\texttt{asm}})$, then

$$\textit{\texttt{printE}} \vdash p(r_1, h) \approx_{\top\times=} p(r_2, h).$$

Note that any Asm program that only ever reads from a register after it has written to that register will satisfy this property. We also lift this definition to whole Imp programs by applying it separately to each inlined Asm block.

Definition 25 (Validity)**.** *c is a valid IMP program if any inlined ASM program it contains is an inline-valid ASM program.*

Including validity with our other semantic conditions is sufficient to guarantee security, so we can safely define the following semantic typing rule.

$$[\textsc{Semantic}]\ \frac{\begin{array}{c} c \text{ is px-noninterfering} \\ c \text{ satisfies the px–exceptions-and-state property} \\ c \text{ is px-confined to } pc \text{ and } \ell_{ex} \\ c \text{ is valid} \end{array}}{\Gamma; pc \vdash_{px} c \diamond \ell_{ex}}$$

Adding this new rule to both type systems allows them to reason about multi-language programs including inline Asm and larger systems, even when the syntactic type system cannot reason about every component. Importantly, Semantic is sound from a security perspective. That is, Theorem 22 continues to hold for both extended type systems.

4.6. Preserving Noninterference Across Compilation

For a compiled language like Imp, noninterference is only part of the story. After all, rather than run Imp code directly, programmers instead compile Imp to Asm and run the Asm. Compilation can change programs significantly, and can introduce insecurity in the process. Thus, we need to

$$
\begin{array}{llcl}
\text{Registers} & r & ::= & \$0 \mid \$1 \mid \ldots \\
\text{Operands} & o & ::= & r \mid n \\
\text{Instructions} & i & ::= & \textsc{add}\ r_1 \leftarrow r_2, o \mid \textsc{sub}\ r_1 \leftarrow r_2, o \mid \textsc{mul}\ r_1 \leftarrow r_2, o \\
 & & \mid & \textsc{eq}\ r_1 \leftarrow r_2, o \mid \textsc{leq}\ r_1 \leftarrow r_2, o \mid \textsc{not}\ r \leftarrow o \\
 & & \mid & \textsc{mov}\ r_1 \leftarrow r_2 \mid \textsc{load}\ r \leftarrow x \mid \textsc{store}\ x \leftarrow r \mid \mathsf{print}(\ell, r) \\
\text{Branches} & b & ::= & \textsc{jmp}\ \mathrm{A} \mid \textsc{brz}\ r\ \mathrm{A1}\ \mathrm{A2} \mid \textsc{raise}\ \ell \\
\text{Blocks} & B & ::= & \mathrm{A} : i_1 ; \cdots ; i_n ; b \\
\text{Programs} & p & ::= & \textsc{Start} : i_1 ; \cdots ; i_n ; b \\
 & & & B_1 ; \cdots ; B_m
\end{array}
$$

Figure 4.8: Secure ASM syntax where x is a variable, A is an address, n is a natural number, and ℓ is an information-flow label.

ensure that the compiler translates noninterfering IMP programs into noninterfering ASM programs. We now turn our attention to the proof-engineering effort involved in providing such an assurance. In particular, we show that (a) adding exceptions and information-flow labels to IMP does not complicate the proof of compiler correctness, and (b) turning a proof of correctness into a proof of noninterference preservation is simple using mixed transitivity (Theorem 12).

Note that, to build our compiler, we had to fix the number of information-flow labels. We thus specialize our discussion of IMP from Section 4.5 to the two-point lattice $\mathcal{L} = \{\top, \bot\}$. Using any other finite lattice would require only minimal changes.

4.6.1. ASM, Its Semantics, and the Compiler

Figure 4.8 presents the syntax of ASM, the simple assembly language that our compiler targets. An ASM program is a sequence of *blocks*, where each block starts at some address A and consists of a sequence of straight-line instructions followed by a single jump. The first block must be at the special address START.

Most ASM instructions write to exactly one register, computing the written value from a combination of other registers and integer constants. For instance, ADD $\$0 \leftarrow \$1, 1$ takes the value of register \$1, adds one, and stores the result in register \$0. The MOV instruction copies the value of one register into another, while LOAD and STORE move information between registers and the heap. Finally, the PRINT instruction prints information to a stream, depending on the label ℓ.

Jumps are either direct jumps, conditional jumps, or exceptions. A direct jump JMP A immediately moves execution to the beginning of the block with address A. A conditional jump BRZ r A1 A2 move execution to A1 if register r contains zero and A2 otherwise. The RAISE ℓ branch raises an exception. Note that there is no equivalent of catching an exception. We assume that ASM programs always jump to either the address of one of the program's blocks or a special EXIT address.

Rather than representing ASM syntax directly in our Coq code, we take a more compositional approach and represent *sub–Control-Flow Graphs (sub-CFGs).* These represent the structure of part of an ASM program. While a complete ASM program contains a unique START address, sub-CFGs may contain multiple addresses accessible to the outside. We refer to addresses which are accessible to the outside as *input* addresses. Likewise, sub-CFGs may jump to undefined addresses, whereas complete ASM programs always jump either to a defined address or EXIT. We refer to the undefined addresses a sub-CFG may jump to as its *output* addresses. Thus, a complete ASM program is a sub-CFG with exactly one input address (START) and exactly one output address (EXIT).

Intuitively, sub-CFGs execute starting at some input address, potentially jumping internally several times before they jump to some output address. To represent this pattern, we give sub-CFGs semantics as functions from an address to an ITree that returns an address. That is, the semantics of a sub-CFG takes as input the input address at which to start executing, and produces an ITree that returns the output address the program jumps to. This structure is due to Xia et al. (2020), and their semantics needed only minor changes to accommodate printing and exception-throwing.

In Xia et al.'s original compiler, IMP code always mapped to complete ASM programs. However, to accommodate exception throwing, our compiler has an extra step of indirection. We map IMP programs to sub-CFGs with exactly one input address but *three* output addresses. The first represents EXIT, as in a complete ASM program, while the second two represent the location of exception handler code. Thus, we compile throw(ℓ) to a jump to the second address if $\ell = \bot$ and the third address if $\ell = \top$. To compile a try-catch command, we place one copy of the handler at the second address and a second copy at the third address. That means any exception will jump to the handler code, regardless of the label of the exception, matching the semantics we gave IMP in Section 4.3.

Note that we still need separate addresses for each label to properly compile *uncaught* exceptions.

For inlined Asm code, we would hope to include it in the compiled code directly with no changes. Unfortunately, if inlined Asm throws an exception with a raise instruction, the surrounding Imp code can catch it, but embedding the raise unmodified in the compiled output would render the exception uncatchable. To support catching these exceptions, we process inlined Asm to replace raise instructions with jumps to the appropriate address. This change causes the inlined exception to properly jump to the handler code.

While the infrastructure described above translates Imp code into sub-CFGs, the end goal of our compiler is to translate complete Imp programs into complete Asm programs. The final step uses the two output addresses for exceptions by linking the sub-CFG of the complete Imp program with *two different* handlers. The low-security exception handler raises a low-security exception, while the high-security exception handler raises a high-security exception. Thus, any Imp code that raises an exception compiles to a complete Asm program that raises that same exception, while Imp code that catches an exception compiles to a complete Asm program with equivalent control flow.

4.6.2. Compiler Correctness

We adapt Xia et al.'s [2020] proof of compiler correctness to account for the modifications we have made to Imp and Asm. We formalize correctness by comparing the source and the target programs—after interpretation—using weak bisimilarity. Intuitively, two stateful programs are weakly bisimilar if, whenever they are given *related* start states, the resulting ITrees are weakly bisimilar. We use a return relation $\mathcal{R}_{\text{env}}$. $\mathcal{R}_{\text{env}}$ ignores the register files and compares heaps using a relation $\cong$, which ensures that they map equal variables to equal values. We can now state the correctness theorem for the `compile` function.

Theorem 23. *For any initial heap states h_1, h_2 such that $h_1 \cong h_2$, any register states r_1, r_2, and a valid IMP command c, the following equation holds*

$$\texttt{excE} \oplus \texttt{printE} \vdash \texttt{interp}\ h_{imp}\ [\![c]\!]_c\ (r_1, h_1) \approx_{\mathcal{R}_{env}} \texttt{interp}\ h_{asm}\ [\![\texttt{compile}(c)]\!]_{\texttt{asm}}\ (r_2, h_2)$$

where $\mathcal{R}_{env}((\_, h_1, \_), (\_, h_2, \_)) \iff h_1 \cong h_2$.

Notably, the changes necessary to adapt Xia et al.'s [2020] proof of correctness to our modified compiler are small and isolated. Most cases of the inductive proof, corresponding to existing language features, needed only cosmetic changes. The new language features required new, but conceptually uninteresting, cases.

4.6.3. Compiler Security

We finally turn to our ultimate goal: proving that our compiler preserves security. There are two important notions of security for our compiler, both of which require cross-language reasoning. The first is that secure source programs are indistinguishable—by all adversaries—from target programs. This property directly relates an Imp program to an Asm program. The second is that the compiler preserves noninterference. While noninterference itself is a property of a single program, *preserving* noninterference is a property of a translation between two languages, which requires cross-language reasoning.

In order to formalize the idea of a secure Imp program being indistinguishable from its compilation, we need to compare these programs, even though they come from different languages. Because we defined `seutt` purely semantically, we can use it as easily as if we were comparing programs in the same language. We use the return relation $\mathcal{R}^{\ell}_{\Gamma}$, which again ignores the register file and ensures that they map equal *visible* variables to equal values. The theorem then takes the following form.

Theorem 24. *For any valid IMP program c, if* `interp` h_{prog} $[\![c]\!]_c$ *is noninterfering with state relation* $\mathcal{R}^{\ell}_{\Gamma}$ *and return relation =, and if c is a valid IMP program, then the following* `seutt` *equation holds for any label* ℓ*, arbitrary register states* r_1, r_2 *and heap states* h_1, h_2 *such that* $h_1 \cong^{\ell}_{\Gamma} h_2$.

$$\texttt{\textit{excE}} \oplus \texttt{\textit{printE}} \vdash_{px} \texttt{interp}\ h_{prog}\ \ [\![c]\!]_c\,(r_1, h_1) \approx^{\ell}_{\mathcal{R}^{\ell}_{\Gamma}} \texttt{interp}\ h_{prog}\ \ [\![\texttt{\textit{compile}}(c)]\!]_{\texttt{asm}}\,(r_2, h_2)$$

Our second theorem is simply that our compiler takes noninterfering Imp programs to noninterfering Asm programs.

Theorem 25 (Noninterference Preservation)**.** *For a valid IMP program c, if* `interp` h_{prog} $[\![c]\!]_c$ *is noninterfering with state relations* $\mathcal{R}^{\ell}_{\Gamma}$ *and return relation* $=$*, then the same holds for its compilation. That is,* `interp` h_{prog} $[\![\mathit{compile}(c)]\!]_{\texttt{asm}}$ *is noninterfering with* $\mathcal{R}^{\ell}_{\Gamma}$ *and* $=$*. This result holds for both progress-sensitive and progress-insensitive noninterference.*

Notably, the proofs of both theorems follows directly from Theorem 23 and mixed transitivity, showing the utility of mixed transitivity for cross-language security reasoning.

CHAPTER 5

Interaction Tree Specifications

This chapter was previously published as Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott. Interaction Tree Specifi- cations: A Framework for Specifying Recursive, Effectful Computations That Supports Auto- Active Verification. In Karim Ali and Guido Salvaneschi, editors, 37th European Conference on Object-Oriented Programming (ECOOP 2023), volume 263 of Leibniz International Proceedings in Informatics (LIPIcs), pages 30:1–30:26, Dagstuhl, Germany, 2023c. Schloss Dagstuhl – Leibniz- Zentrum für Informatik. ISBN 978-3-95977-281-5. doi: 10.4230/LIPIcs.ECOOP.2023.30. URL https://drops.dagstuhl.de/opus/volltexte/2023/18223. I was the primary author and did most of the research.

5.1. Introduction

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 (He et al., 2021; Matsushita et al., 2022, 2020; Astrauskas et al., 2019; Ho and Protzenko, 2022). That is, there exists a program in a functional language whose behavior is equivalent to the original, heap-manipulating program. This functional program is called a *functional model.* Most prior work relies only implicitly on the functional model. In other work, such as VST (Appel, 2011), it is idiomatic for users to invent a functional model, prove it correct with respect to the original program, and then directly reason about the functional model. In contrast, the Heapster tool (He et al., 2021) automatically reifies the functional models into concrete code, represented in Coq as an ITree. Proof engineers can then verify properties about the derived functional code, and ensure those properties hold on the original program. This chapter presents a variation of ITrees for writing specifications over this derived functional code and shows how to reason about these specifications.

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 an ITree-based 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. This chapter presents techniques to write and prove specifications over the resulting functional programs. Alongside Heapster, these techniques form a pipeline for verifying low-level, heap manipulating programs.

In this work, we present *interaction tree specifications*, or ITree specifications. ITree specifications are an *auto-active verification framework* for programs based on ITrees. Auto-active verification is a verification technique that merges user input and automated reasoning to leverage the benefits of each. 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 (Maillard et al., 2019; Swamy et al., 2013; Ahman et al., 2017; Swamy et al., 2016). 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 presented in Chapter 3. 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. This chapter 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, abstract specifications, and combinations of both. For example, consider the following computation `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
              ).
```

This specification is defined with `rec_fix_spec`, a recursion operator (defined in Section 5.4) 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 5.6, 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 ← ∃_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 `∃_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 5.2 and Section 5.3. The contributions of this chapter are as follows:

- ITree specifications, a data structure for representing specifications over monadic, recursive, interactive programs, presented in Section 5.4

- a specification refinement relation over ITree specifications, along with collection of verified, syntax-directed proof rules for refinement also presented in Section 5.4,

- tools for encoding and proving refinements involving total correctness specifications in ITree specifications presented in Section 5.5,

- an auto-active verification technique briefly discussed in Section 5.6
- an evaluation of the presented techniques in the form of verifying a collection of realistic C functions using ITree specifications and Heapster is presented in Section 5.6.

5.2. Interaction Trees Background

Section 5.6 evaluates the ITree specifications framework by using it in concert with the Heapster translation tool to verify C code. However, the translation tool cannot target ITrees as defined in Chapter 2. This is due to interactions between Coq universe levels and event type families, which are used to model recursive calls. In order to avoid this issue, we develop a variant of ITrees that is designed to interact differently with Coq universe levels (Coq development team, 2023).

The key difference in this variant is that event signatures are represented as types, in `Type`, rather than type families, in `Type → Type`. This change necessitates a further change to the representation of the response type of events. In the original definition, the response type is carried in the type of a particular event. The event `e : E A` is an event with the event type signature `E` that has the response type `A`. For the new definition, the event has a type, `E`, and its response type is determined by a separately defined function, in `E → Type`. For ease of use, this function is provided in a type class.

This section first provides background information about universe levels in Coq and then uses that information to show an example of code that we cannot write in the original variant of ITrees. Finally, it introduces the new definition and demonstrates that it can handle that code example.

5.2.1. Coq Universe Levels

In Coq, there is no separation between types and expressions. Just as it contains functions of type `nat → bool`, it also contains functions of type `nat → Type` or even `Type → Type`. In Coq, `Type` is both a type and an expression that itself has a type. For many purposes, we can treat `Type` as an inhabitant of `Type`. In fact, when run with default settings, Coq will report that `Type : Type`. However, this is a simplification of the actual behavior of the type system. Type theories where `Type` is in `Type` are inconsistent due to Girards Paradox (Coquand, 1999).

$$[\textsc{Cont}]\ \frac{\texttt{Type@\{a\}} : \texttt{Type@\{b\}}}{\texttt{a} \leq \texttt{b}}$$

$$[\textsc{Quant}]\ \frac{\begin{array}{c} \texttt{c : } \forall\ \texttt{(A : Type@\{a\}), ...} \to \texttt{t arg1 ... argn} \\ \text{c is a constructor of t} \qquad \texttt{t arg1 ... argn : Type\{b\}} \end{array}}{\texttt{a < b}}$$

Figure 5.1: Universe levels in Coq

```
CoInductive itree (E : Type@{a} → Type@{b}) (R : Type@{c}) : Type@{d} :=
 | Ret (r : R) : itree E R
 | Tau (t : itree E R) : itree E R
 | Vis : ∀{A : Type@{a}} (e : E A) (k : A → itree E R), itree E R.
```

Figure 5.2: ITree definition with explicit universe levels

Girard's Paradox is analogous to Russell's Paradox in set theory. And much like with Russell's Paradox, type theories typically avoid Girard's Paradox by replacing a unified type of types with a collection of *type universes*, each indexed by a natural number. In Coq, these universe levels are generated automatically during type checking. A Coq type at universe level `a` can be written explicitly as `Type@{a}`. In this system, a type universe contains types at strictly lower universe levels. In particular, `Type@{a} : Type@{b}` exactly when `a < b`.

Much like Russell's Paradox, Girard's Paradox is only possible in type theories where a type can be defined by quantifying over a collection of types that includes itself. Universe levels are designed to prevent users from defining such types. To this end, many typing rules in Coq contain constraints on the universe levels. For the purposes of this chapter, we only need to focus on the constraints generated when an inductive or coinductive type constructor quantifies over types. Figure 5.1 presents this constraint as an inference rule. We focus on this case because it applies to the original ITrees definition, namely the `Vis` constructor, presented with explicit universe levels in Figure 5.2. Given a term `c` which quantifies over a type `A : Type@{a}`, where `c` is a constructor for a type `t arg1 ... argn` in type universe level `b`, Coq will enforce that `a < b`.

```
Inductive addE : Type@{a} → Type@{b} :=
| add_intro (n : nat) : addE (nat → itree voidE nat).

Definition lift {E R}: itree voidE R → itree E R :=
    (* omitted *)

Definition add (n : nat) : itree voidE (nat → itree voidE nat) :=
   mrec (
      fun '(add_intro n) ⇒
         match n with
         | 0 ⇒ ret (fun m ⇒ ret m)
         | S n ⇒ addf  ← trigger (add_intro n);;
                ret (fun m ⇒ fmap S (lift (addf m)))
         end
   ) (add_intro n).
```

Figure 5.3: ITree model of addition

5.2.2. Approaching General Fixpoints with Interaction Trees

The verification framework presented in this chapter relies on a translation tool from well-typed programs to ITrees. Modelling these well-typed programs requires modelling recursive functions that can return functions. This section presents a concrete example of a function that can return functions and explains why that poses problems for the ITree representation. The following code implements addition by pattern matching on the first natural number to construct a function from natural numbers to natural numbers.

```
fix add n.
    match n with
    | 0 => λ m. m
    | S n => λ m. S (add n m)
```

This implementation of addition begins by pattern matching on the first argument. In the zero case, it returns the identity function. In the successor of `n` case, it returns a function that recurses on `n` and its argument, and returns the successor of the result. In this definition, recursive calls to the `add` function return a function from natural numbers to natural numbers. This is in contrast to the typical way of defining `add`, where a recursive call requires both natural number inputs, and returns the natural number output.

Modelling code like poses a problem for the previously presented definition of ITrees. Figure 5.2 presents the definition of ITrees with explicit universe levels. It shows that that all response types, `A`, must be in universe level `a` and that the full ITree datatype, `itree E R`, is in universe level `d`. The constraint presented in Figure 5.1 further indicates that `a` must be strictly less than `d` because the `Vis` constructor quantifies over `Type@{a}`.

Figure 5.3 proposes a model of the previously defined addition function written in ITrees with the `mrec` combinator. It models recursive calls that return functions with the `addE` event. The only constructor of `addE` takes in a natural number and has the response type with models possibly divergent functions from natural numbers to natural numbers, `nat` → `itree voidE nat`. Much like the code it models, it pattern matches on the first natural number. In the zero case, it returns a function that returns its argument. In the successor of `n` case, it triggers a recursive call containing the payload `n`, and binds that call to a continuation that returns a function that applies the recursive call response to its argument, coerces the result to an ITree with `addE` events, and maps the successor function across that ITree.

However, this code is rejected by the Coq type checker because it violates universe level constraints. We already know that `a < d` because of constraints generated by the `Vis` constructor. Because the recursive calls all have type `addE (nat` → `itree voidE nat)`, we have the additional constraint that `d` ≤ `a`. We use a type of functions into ITrees as the response type of `addE`. This type of functions lives at the same universe level as ITrees and is used in a context that requires it to be coerced to `Type@{a}`. And there is no way to satisfy the system of inequalities `d` ≤ `a < d`.

In the following section, I present a version of ITrees that does not share this inconsistency.

5.2.3. Alternate Definition of Interaction Trees

Like the definition presented in Chapter 2, this alternate definition of ITrees represents programs as potentially infinite trees whose nodes are labelled with events. However, in this definition, events are all inhabitants of a single event type `E`, rather than a type family. The response type, which determines the type that indexes branches of this event node , is determined by a separately defined

```
Class EncodingType (E : Type@{a}) : Type :=
  response_type : E → Type@{a}.

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

Figure 5.4: Alternate ITrees definition

function. The alternate definition of ITrees is presented in Figure 5.4. The `EncodingType` typeclass contains a function, `response_type`, from the type of events into `Type`. ITrees are defined for any event type which has an `EncodingType` instance, and uses the `response_type` function to compute the input type of the continuation `k` in the `Vis` constructor. We leverage the `EncodingType` typeclass in order to avoid explicitly including a particular `response_type` function every time we reference the `itree` type.

This definition avoids any quantification over types, replacing them with applications of the `response_type` function. Figure 5.4 also includes explicit universe levels for many of the types, which demonstrate why this definition lacks the problematic universe level constraints of the original. The definition fixes a universe level `a` and enforces that both the type of events, `E`, and the output of the `response_type` function are inhabitants of the type universe level `a`. Because the `Vis` constructor uses the `response_type` function to compute the input type of the continuation instead of using type quantification, we can assign both the type of ITrees, `itree E R`, and the type of events, `E`, the same universe level.

With this definition, we can write the code presented in Figure 5.3 with minimal modifications and actually make it type check. This properly typed code is presented in Figure 5.5. Because the definition is so close to the original, we can rewrite all of the combinators and metatheory presented in Chapter 2. Key definitions referenced in the rest of this chapter are presented in Appendix A.

5.3. Functional Model 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 5.6. In the evaluation, we demonstrate

```
Inductive addE : Type@{a} :=
   add_intro (n : nat).

Instance addE_encoding : EncodingType addE :=
   fun _ ⇒ nat → itree void nat.

Definition add (n : nat) : itree void (nat → itree void nat) :=
   mrec (
      fun '(add_intro n) ⇒
         match n with
         | 0 ⇒ ret (fun m ⇒ ret m)
         | S n ⇒ addf ← trigger (add_intro n);;
                ret (fun m ⇒ fmap S (lift (addf m)))
         end
   ) (add_intro n).
```

Figure 5.5: Alternate definition of ITree model of addition

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 (Astrauskas et al., 2019; Matsushita et al., 2020, 2022; He et al., 2021) 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 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, *functional model extraction* is the idea that the functional model can be extracted automatically as an artifact that can be used for verification. Functional model 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 functional models that are extracted from this type-checking process. The Heapster tool (He et al., 2021) is an implementation of the idea of functional model 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

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

Figure 5.6: An Abbreviated Grammar of the Heapster Type System

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 functional model extraction process in this section and illustrate it with an example.

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 5.6 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 pointers, among other value types not discussed here. Heapster uses the CompCert memory model (Leroy and Blazy, 2008), 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 5.6, $\mathsf{ptr}\ ((rw, e) \mapsto \tau)$, 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;
```

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:

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

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

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[5] 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 5.7 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

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

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

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 5.7. In order to type-check the `NULL` comparison 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 $\mathsf{int64}$. 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 $\mathsf{int64}$. 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 functional model extraction to extract a pure functional model of the function's behavior. Functional model 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 functional model 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 $\mathtt{interp}T$ of value types that we omit here). The equality type $\mathsf{eq}(e)$ is translated to the Coq unit type `unit`, meaning that the extracted model contains no information. 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

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

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. (2021) for more detail. The translation of `is_elem` is given as a Coq model `is_elem_spec` in Figure 5.8. At the top level, this mdeol 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 understand the rest of the model, we step through the Heapster type-checking depicted in

```
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 if found *)
                     else rec (x,tl))                       (* recursive call *)
                  (unfoldList l)).                          (* unfolded argument *)
```

Figure 5.8: Extracted Functional Model for `is_elem`

Figure 5.7. The first step of that type assignment unfolds the permission type list64⟨W⟩ on `l`. The corresponding portion of the model 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 model 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 model, 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 model, 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 model performs a recursive call, passing the same value for `x` and the tail of the input list for `l`.

5.4. 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_∀ (A : type) : SpecEvent E
  | Spec_∃ (A : type) : SpecEvent E
```

.

The Spec_vis constructor allows you to embed a base event e : E into the type SpecEvent E. The Spec_∀ constructor signifies universal quantification, and the Spec_∃ constructor signifies existential quantification.

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.

5.4.1. ITree Specification Refinement

The notion that a program adheres to a specification is defined in terms of refinement over specifications. Refinement is the main judgment involved in using ITree specifications, and is 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 containing a single concrete ITree, 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 5.4.4.

The ITree specification refinement relation is based on the idea of refinement of logical formulae with the eutt relation. In the refinement relation, we 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

$$[\textsc{RefinesRet}]\ \frac{\texttt{RR r1 r2}}{\texttt{RPre};\texttt{RPost} \vdash (\texttt{ret r1}) \sqsubseteq_{\texttt{RR}} (\texttt{ret r2})} \qquad [\textsc{RefinesTau}]\ \frac{\texttt{RPre};\texttt{RPost} \vdash \texttt{t1} \sqsubseteq_{\texttt{RR}} \texttt{t2}}{\texttt{RPre};\texttt{RPost} \vdash \texttt{Tau t1} \sqsubseteq_{\texttt{RR}} \texttt{Tau t2}}$$

$$[\textsc{RefinesTauL}]\ \frac{\texttt{RPre};\texttt{RPost} \vdash \texttt{t1} \sqsubseteq_{\texttt{RR}} \texttt{t2}}{\texttt{RPre};\texttt{RPost} \vdash \texttt{Tau t1} \sqsubseteq_{\texttt{RR}} \texttt{t2}} \qquad [\textsc{RefinesTauR}]\ \frac{\texttt{RPre};\texttt{RPost} \vdash \texttt{t1} \sqsubseteq_{\texttt{RR}} \texttt{t2}}{\texttt{RPre};\texttt{RPost} \vdash \texttt{t1} \sqsubseteq_{\texttt{RR}} \texttt{Tau t2}}$$

$$[\textsc{SpecVis}]\ \frac{\texttt{RPre e1 e2} \qquad \forall\ \texttt{a b}, \texttt{RPost e1 e2 a b} \rightarrow \texttt{RPre};\texttt{RPost} \vdash (\texttt{k1 a}) \sqsubseteq_{\texttt{RR}} (\texttt{k2 b})}{\texttt{RPre};\texttt{RPost} \vdash \texttt{Vis Spec\_vis e1 k1} \sqsubseteq_{\texttt{RR}} \texttt{Vis Spec\_vis e2 k2}}$$

$$[\textsc{ForallR}]\ \frac{\forall\ \texttt{a}, \texttt{RPre};\texttt{RPost} \vdash \texttt{t} \sqsubseteq_{\texttt{RR}} \texttt{k a}}{\texttt{RPre};\texttt{RPost} \vdash \texttt{t} \sqsubseteq_{\texttt{RR}} \texttt{Vis Spec\_}\forall\ \texttt{k}} \qquad [\textsc{ExistsR}]\ \frac{\exists\ \texttt{a}, \texttt{RPre};\texttt{RPost} \vdash \texttt{t} \sqsubseteq_{\texttt{RR}} \texttt{k a}}{\texttt{RPre};\texttt{RPost} \vdash \texttt{t} \sqsubseteq_{\texttt{RR}} \texttt{Vis Spec\_}\forall\ \texttt{k}}$$

$$[\textsc{ForallL}]\ \frac{\exists\ \texttt{a}, \texttt{RPre};\texttt{RPost} \vdash \texttt{k a} \sqsubseteq_{\texttt{RR}} \texttt{t}}{\texttt{RPre};\texttt{RPost} \vdash \texttt{Vis Spec\_}\forall\ \texttt{k} \sqsubseteq_{\texttt{RR}} \texttt{t}} \qquad [\textsc{ExistsL}]\ \frac{\forall\ \texttt{a}, \texttt{RPre};\texttt{RPost} \vdash \texttt{k a} \sqsubseteq_{\texttt{RR}} \texttt{t}}{\texttt{RPre};\texttt{RPost} \vdash \texttt{Vis Spec\_}\forall\ \texttt{k} \sqsubseteq_{\texttt{RR}} \texttt{t}}$$

Figure 5.9: Inference rules for ITree specifications refinement relation

corresponding Coq quantifier. This means that both a `Spec_`∀ on the right and a `Spec_`∃ on the left get eliminated to a Coq ∀. And both a `Spec_`∃ on the right and a `Spec_`∀ on the left get eliminated to a Coq ∃. ITree specifications form a lattice with refinement serving as the preorder, `Spec_`∀ acting as the complete meet, and `Spec_`∃ 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`.

Definition 26 (ITree Specification Refinement)**.** *Given:*

- *event signatures* `E1` *and* `E2`*;*

- *return types* `R1` *and* `R2`*;*

- *a precondition relation over* `E1` *and* `E2`*,* `RPre`*;*

- *a postcondition relation over* `E1` *and* `E2`*,* `RPost`*;*

- *and a return relation over* `R1` *and* `R2`*,* `RR`*,*

refinement up to `RPre`*,* `RPost` *and* `RR`*, a relation between* `itree E1 R1` *and* `itree E2 R2`*, is defined with the inference rules presented in Figure 5.9. We write this relation as* $RPre; RPost \vdash t1 \sqsubseteq_{RR} t2$*.*

Several of the inference rules presesnted in Figure 5.9 work exactly like corresponding inference rules in the `rutt` relation. In particular, the REFINESRET, REFINESTAU, REFINESTAUL, and REFINESTAUR rules handle return values and `Tau` nodes in the standard way. The SPECVIS rule handles `Spec_vis` nodes just as the `rutt` relation handles any event nodes. Just like RUTTVIS, SPECVIS 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`. The added complications of this rule allow us to reason about mutually recursive functions. This rule ensure 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 inference rules dealing with quantifier events. This definition uses only inductive inference rules 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. The choice to only use inductive rules for quantifier events ensures 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 by FORALLR and EXISTSR. Quantifiers on the left get translated into their dual quantifier at the Coq level. Eliminating a Spec_∀ on the left gives you an ∃, enforced by FORALLL. Eliminating a Spec_∃ on the left gives you a ∀, enforced by EXISTSR.

5.4.2. 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_∀ t) (fun _ ⇒ Tau (phi1)).
CoFixpoint phi2 : itree_spec E R := Vis (Spec_∀ t) (fun _ ⇒ phi2).
```

The spin specification represents a silently diverging computation. The phi1 specification is an infinite stream that alternates between Spec_∀ 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_∀L rule to get rid of the Spec_∀ 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_∀ 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 ITree specifications. Problems like this arise with any ITree specifications that consist of infinitely many quantifier nodes with nothing between them.

$$[\textsc{PaddedRet}]\ \frac{}{\texttt{padded (ret } r\texttt{)}} \qquad [\textsc{PaddedTau}]\ \frac{\texttt{padded t}}{\texttt{padded (Tau t)}}$$

$$[\textsc{PaddedVis}]\ \frac{\forall a, \texttt{padded (k a)}}{\texttt{padded Vis } e\ (\lambda a.\texttt{Tau } (k\ a))}$$

Figure 5.10: `padded` Definition

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

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

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

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

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

Figure 5.11: Basic Specifications

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 is presented in Figure 5.10. 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 27 (Padded Refinement)**.** *Given: a precondition relation,* `RPre`*; a postcondition relation,*

```
CoFixpoint interp_mrec_spec {R : Type}
  (bodies : ∀(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_∀ A) k ⇒ Vis (@Spec_∀ E _ A) (fun x : response_type (Spec_∀ A) ⇒ interp_mrec_spec
       bodies (k x))
  | Vis (Spec_∃ A) k ⇒ Vis (@Spec_∃ 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 : ∀(d:D), (itree_spec (D + E)) (response_type d)) (init : D) :=
  interp_mrec_spec bodies (bodies init).
```

Figure 5.12: `mrec_spec` Definition

`RPost`; a return relation, `RR`; and two specifications, `phi1` and `phi2`; the specifications `phi1` and `phi2` are contained in the relation `padded_refines RPre RPost RR` if and only if `RPre`; `RPost` $\vdash$ `pad phi1` $\sqsubseteq_{RR}$ `pad phi2`. We write the padded refinement relation as `RPre`; `RPost` $\vdash_p$ `phi1` $\sqsubseteq_{RR}$ `phi2`.

In Figure 5.11, we introduce several simple ITree specifications that implement quantification over some types, and assumption and assertion of propositions. The `∀_spec` and `∃_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 `∀_spec` and `∃_spec` to define assumption and assertion, respectively, as `Prop` is part of the restricted grammar of types that `SpecEvent` can quantify over.

5.4.3. 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.

```
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) :
   (∀ (d2 : D2), R1 d1 d2 → R2 d2 d3 →
   ∃ b, PR1 d1 d2 a b ∧ PR2 d2 d3 b c) →
   RComposePostRel R1 R2 PR1 PR2 d1 d3 a c.
```

Figure 5.13: Post relation composition

Theorem 26 (Padded Refinement Respects Bind)**.** *If* $\mathtt{RPre}; \mathtt{RPost} \vdash_p phi1 \sqsubseteq_{\mathtt{RR}} phi2$ *and given any* $\mathtt{r1}$ *and* $\mathtt{r2}$ *contained in* $\mathtt{RR}$*,* $\mathtt{RPre}; \mathtt{RPost} \vdash_p \mathtt{kphi1\ r1} \sqsubseteq_{\mathtt{RS}} \mathtt{kphi2\ r2}$*,*

then $\mathtt{RPre}; \mathtt{RPost} \vdash_p \mathtt{bind\ phi1\ kphi1} \sqsubseteq_{\mathtt{RS}} \mathtt{bind\ phi2\ kphi2}$*.*

We prove that the `padded_refines` relation is transitive. To state the transitivity result in full generality, we need a `PostRel` relational composition operator. This operator is defined in Figure 5.13. In addition to taking two post condition relations to compose, it relies on two precondition relations known as *coordinating relations*. An *coordinating event* between `d1 : D1` and `d3 : D2` is any event `d2 : D2` which is related to `d1` by the first coordinating relation and is related to `d3` by the second. To relate a four tuple `d1 : D1`, `d3 : D3`, `a : response_type d1`, and `c : response_type d3`, we need to prove that for any coordinating event `d2`, there exists some *coordinating answer* `b : response_type d2` such that the first post condition relates `d1`,`d2`,`a`,`b` and the second post condition relates `d2`,`d3`,`b`,`c`.

Theorem 27 (Transitivity of Padded Refinement)**.** *If* $\mathtt{RPre1}; \mathtt{RPost1} \vdash_p phi1 \sqsubseteq_{\mathtt{RR1}} phi2$ *and* $\mathtt{RPre2}; \mathtt{RPost2} \vdash_p phi2 \sqsubseteq_{\mathtt{RR2}} phi3$*, then*

$\mathtt{RPre1} \circ \mathtt{RPre2}; \mathtt{RComposePostRel\ RPre1\ RPre2\ RPost1\ RPost2} \vdash_p phi1 \sqsubseteq_{\mathtt{RR1} \circ \mathtt{RR2}} phi3$*.*

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 5.12. 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

$$[\textsc{concreteret}]\ \frac{}{\texttt{concrete (ret }r\texttt{)}} \qquad\qquad [\textsc{concretetau}]\ \frac{\texttt{concrete t}}{\texttt{concrete (Tau t)}}$$

$$[\textsc{concretevis}]\ \frac{\forall a, \texttt{concrete (k a)}}{\texttt{concrete (Vis (Spec\_vis }e\texttt{) }k\texttt{)}}$$

Figure 5.14: `concrete` Definition

hold, but strong enough to imply post conditions.

Theorem 28 (Padded Refinement Respects MRec)**.** *If recursive call events* `i1` *and* `i2` *are contained in the precondition invariant* `RPreInv`*, and given any recursive call events,* `d1` *and* `d2`*, contained in* `RPreInv`*,* $\texttt{SumRel RPreInv RPre}; \texttt{SumPostRel RPostInv RPost} \vdash_p \texttt{bodies1 d1} \sqsubseteq_{\texttt{RPostInv d1 d2}} \texttt{bodies2 d2}$*, then*
$\texttt{RPre}; \texttt{RPost} \vdash_p \texttt{mrec\_spec bodies1 i1} \sqsubseteq_{\texttt{RPostInv i1 i2}} \texttt{mrec\_spec bodies2 i2}$*.*

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.

Definition 28 (Strict Refinement)**.** *Specification* `phi1` *strictly refines specification* `phi2` *if and only if* $\texttt{eq}; \texttt{PostRelEq} \vdash_p phi1 \sqsubseteq_{\texttt{eq}} phi2$*. In this case, we write* $\texttt{phi1} \leq \texttt{phi2}$*.*

As a corollary of Theorem 27, strict refinement is a transitive relation, and is strong enough to allow rewrites under the context of any other application of `padded_refines`.

5.4.4. Interaction Tree Specification Incompleteness

One way to interpret ITree specifications is as sets of ITrees. Figure 5.14 defines a predicate of *concrete* ITree specifications, which correspond to executable ITrees. 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 :=
  ∀_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 ∀ 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`[6]. In each case, apply the right ∃ rule and choose the corresponding branch. However, given any relations `RPre`, `RPost`, `RR`, we cannot prove `padded_refines RPre RPost RR top1 top2`. This is because the only way to eliminate the `Spec_∀` 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.

5.5. 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.

[6] 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) →
           (∀ (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 5.15: `rec_fix_spec` Definition

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  ← ∃_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.

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 specifications. 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 5.15. 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 `∀ (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 ← ∃_spec nat;;
                trepeat n (
                         a' ← ∃_spec A;;
                         assert_spec (Pre a' ∧ Rdec a' a);;
                         rec a'
                       );;
                b ← ∃_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 matches 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[7]. 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 29 (Total Spec Fix Correctness)**.** *If Rdec is a well-founded relation, then total_spec_fix Pre Post Rdec a strictly refines total_spec Pre Post 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.

```
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 : ∀l1 l2,
    merge (l1,l2) ≤ total_spec merge_pre
        merge_post (l1,l2).
```

Figure 5.16: Merge implementation

5.5.1. 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 and combines them into one larger sorted list which contains all the original elements. In Figure 5.16, 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.[8]

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

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

[8] We manage this assumption with the ReSum typeclass. This typeclass is discussed conceptually in Chapter 2 and the code of this particular implementation is provided in Appendix A.

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 5.1. 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 prove the correctness of server_impl.

Theorem 30 (Server Correctness). *server_impl tt strictly refines server_spec tt.*

5.6. Automation and Evaluation

5.6.1. 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.

Further implementation details are provided in the artifact. It is important to keep in mind that 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.

5.6.2. Evaluation

He et al. (2021) discussed using Heapster to verify the interface of mbox, a key datastructure in the implementation of the Encapsulating Security Payload (ESP) protocol of IPSec. This section extends this discussion by using ITree specifications to write and verify specifications the functional specifications produced by Heapster. It also compares the task of verifying one example function using Heapster and ITree specifications with the task of verifying the same function using the VST separation logic (Appel, 2011). It also presents a table with all the mbox functions verified with Heapster and ITree specifications along with a coarse measurement of the effort involved in the verification.

5.6.2.1 Data Representation

The mbox datastructure, whose type is presented in Figure 5.17a, represents a data packet as a linked list of buffers. Each buffer is a segment of a 128 element array of unsigned 8-bit integers. The buffer consists of len integers starting at the index start in the array, data.

```
typedef struct mbox_c {
    size_t start;
    size_t len;
    struct mbox_c *next;
    uint8_t data[128];
} mbox_c;
```

(a) mbox type in C

```
Definition mbox_coq :=
   list (int64 *
         int64 *
         vector 128 uint8).

Fixpoint mbox_vst
   (m : mbox_coq) (p : ptr) :=
   match m with
   | [] ⇒ p = NULL ∧ emp
   | (x,y,v) :: m ⇒
      ∃ p', p ↦ (x,y,p',v)
         * mbox_vst p' m
   end.
```

(b) mbox representation invariant in VST

```
mbox_heapster = μ X.
    (ptr(W,0) ↦ int64 *
    ptr(W,8) ↦ int64 *
    ptr(W,16) ↦ X *
    array(W,24,128,uint8))
        ∨ eq(NULL)
```

(c) mbox type in Heapster

Figure 5.17: mbox representation information

He et al. (2021) 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.

The functional specification extraction relied heavily on the Heapster type for mbox presented in Figure 5.17c, mbox_heapster. This recursively defined type defines the memory layout of a valid mbox on the heap. The mbox_heapster type accepts pointers that either point to a pair of 64-bit integers, a 128 element array of unisgned 8-bit integers, and another mbox_heapster pointer, or are null. Pointer structures that satisfy this heapster type represent the same information as contained in the mbox_coq type presented in Figure 5.17b. The mbox_coq type is the type of lists of tuples which contain two 64-bit integers and a 128 element vector of unsigned 8-bit integers.

A state-of-the-art technique to verify this code is in a separation logic like VST (Appel, 2011). Separation logics like VST provide tools for reasoning about disjoint segments of the heap, like

```
mbox_len_vst :=
DECLARE mbox_len_c
WITH (p : ptr), (m1 : mbox_coq)
PRE
    PARAMS (p)
    SEP (mbox_rep m1 p)
    PROP (⊤)
POST
    ∃ (x : int64) (m2 : mbox),
    RETURN (x)
    SEP (mbox_rep m2 p)
    PROP (x = mbox_len_coq m1
            ∧ m1 = m2)
```

(a) VST specification for `mbox_len_c`

```
mbox_len_heapster :=
    (m : mbox_heapster) ⊸
    (m : mbox_heapster)*
    (ret : int64)

Definition mbox_len_itree_spec
   := total_spec top
      (fun m1 (m2,x) ⇒
         x = mbox_len_coq m1
         ∧ m1 = m2).
```

(b) Heapster type and ITree specification for `mbox_len_c`

Figure 5.18: Reasoning about `mbox_len_c`

the separating conjunction operator *. These tools can be used to define *representation invariants*, predicates that express that the information in a particular segment of the heap represents some abstract datastructure. The `mbox_vst` heap predicate, presented in Figure 5.17b, is a representation invariants for the `mbox` datastructure written in the VST separation logic. The heap predicate, `mbox_vst m p`, asserts that the heap segment pointed to by pointer `p` represent the `mbox_coq` value `m`. An empty list `mbox` is represented by the null pointer. An `mbox` with data at its head, `(x,y,v)::m` is represent by a pointer `p` which points to a segment of data containing 64-bit integers `x` and `y`, a pointer `p'` that represents the `mbox`, `m`, and a 128 element vector of unsigned 8-bit integers, `v`.

This heap predicate is recursively defined and closely resembles the Heapster type for `mbox`. The primary difference, beyond syntax, is that `mbox_vst` predicate ensures that a pointer structure represents a particular `mbox m`, while `mbox_heapster` ensures that a pointer structure represents some `mbox`. This difference is due to the fact that Heapster types are used solely to reason about memory safety. These types are not present in the functionality reasoning steps where the particular `mbox` matters.

5.6.2.2 Example Program

Consider the `mbox_len_c` function presented in the following code.

```
size_t mbox_len_c(const mbox *m) {
    // Add up the cumulative lengths of the mbox chain
    size_t total = 0;
    while (m != NULL) {
        total += m->len;
        m = m->next;
    }
    return total;
}
```

This function takes in a pointer to an `mbox`, loops through it, and returns the total sum of the lengths of each buffer. The function also leaves its input unchanged, neither mutating any of the data values nor deallocating any of the memory. The behavior of this heap-manipulating C code is equivalent to the following Coq code, which sums up the lengths of each buffer in an `mbox` using a functional list fold.

```
Definition mbox_len_coq m := fold add (map (fun (x,y,z) ⇒ y) m) 0.
```

With the Heapster-ITree specification pipeline, the specification of this behavior is split into two parts. First, the function is assigned the Heapster type, `mbox_len_heapster` presented in Figure 5.18b. This type ensures that the `mbox_len_c` function takes in a valid `mbox`, returns a 64-bit integer, and leaves a valid `mbox` in the location of the input. The typing derivation that proves that `mbox_len_c` has type `mbox_len_heapster` is then used to automatically construct a functional specification. Finally, we prove that the functional specification has the desired behavior, ensured by the fact that it refines the `mbox_len_itree_spec` specification also presented in Figure 5.18b. This specification states that given an arbitrary input, the function returns the length of the input, as defined by `mbox_len_coq`, and an `mbox` equal to the input.

To verify the same behavior with VST, we need a specification that contains the information contained in both `mbox_len_heapster` and `mbox_len_itree_spec`. Figure 5.18a presents a streamlined version of the `mbox_len_vst` specification for `mbox_len_c`. The `DECLARE` clause determines which function this specification is for. The `WITH` clause introduces variables in scope for the rest of the specification. In this specification, it introduces a pointer, `p`, that is the input to the function and an `mbox`, `m1`, that is the `mbox` that is represented by the data at `p`.

The rest of the specification is broken up into the precondition, `PRE`, and the postcondition, `POST`. Both the precondition and postcondition contain `SEP` and `PROP` clauses. Each `SEP` clause lists heap predicates which describe the layout of the heap, either before the execution of the function in the case of the precondition, or after the execution in the case of the postcondition. This corresponds roughly to the information contained in Heapster types. Each `PROP` clause lists propositions that describe the abstract mathematical values this function is reasoning about. This corresponds roughly to the information included in ITree specification.

The precondition also contains the `PARAMS` clause. The `PARAMS` clause declares the inputs to the function. In this case, the pointer `p` is the sole input to the function. The precondition's `PROP` clause is trivial in this case.

The `POST` clause begins using an existential quantifier to introduce values that are needed to describe the output. In this case, it introduces a 64-bit integer representing the returned length, and an `mbox`, `m2`, representing the unchanged datastructure represented at `p`. The postcondition also contains the `RETURN` clause. The `RETURN` clause declares the value that the function returns. In this case, it is the integer `x`. The postcondition's `SEP` clause guarantees that `p` satisfies the representation invariant for `m2`. Finally, the postcondition's `PROP` clause asserts that the return value, `x`, is equal to the length of the input `mbox`, `m1`, and that the input and output `mbox` values are the same.

A major difference between these two ways to verify the behavior of `mbox_len_c` is that the Heapster-ITree specifications method provides a firm separation between reasoning about the structure of the heap and reasoning about the abstract data transformations performed by the program. While the VST specification does split provide different `SEP` and `PROP` clauses that break up the reasoning in a similar way, this is a shallow interface on top of a logic in which the user must reason about these

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 5.19: Verified `mbox` functions

concepts simultaneously.

In contrast, with Heapster and ITree specifications this conceptual separation is reified in the separate stages of a verification pipeline. A proof engineer reasons about memory safety and heap layout while writing the Heapster types. The Heapster tool then automatically generates a functional specification. Finally, the proof engineer reasons separately about the behavior of the functional specification. The ITree specifications auto-active verifier reduces this task to number of choices of invariants and proofs of first order propositions.

Anecdotal Experience. While verifying the `mbox_len_c` function in VST, I found that this separation between heap layout reasoning and functionality reasoning was a leaky abstraction. In particular, the separation is not maintained at all when writing proofs, just when writing the specifications. This made the proof noticeably more challenging. For C functions that can be given a valid Heapster type, I believe verification using Heapster and ITree specifications is easier than verification using VST. However, the Heapster type system is conservative compared to a full separation logic like VST, so there remains C code that cannot be verified with the novel techniques in this chapter.

5.6.2.3 Verified Functions

Figure 5.19 presents the full list of verified functions. 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.

CHAPTER 6

Related Work

The work in this dissertation is part of a large and growing literature on program verification. This chapter surveys the most closely related work, starting with modern formal verification in interactive theorem provers. The remainder of the chapter is split into sections corresponding to work related specifically to Chapters 3, 4, and 5 respectively.

6.1. Formal Verification in Interactive Theorem Provers

The primary purpose of this dissertation is to provide language independent formal specification tools. In particular, it develops specification tools for the low level languages that ITrees are particularly well suited to representing. These tools are formalized in the Coq proof assistant (Coq development team, 2023). 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, 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. There has only recently been success in modelling concurrent programs with a variant of ITree semantics (Chappe et al., 2023). Formal verification is also commonly done in the Agda (Norell, 2007) and Isabelle/HOL (Nipkow et al., 2002) proof assistants. $F^\star$, a functional, general-purpose programming language with dependent types and algebraic effects, is also commonly used for program verification (Swamy et al., 2011).

While the literature is full of useful and powerful specification tools, most are specialized to a particular language or system. For example, VST-Floyd (Appel, 2014) is a separation logic for C, and CFML is a separation logic for OCaml (Charguéraud, 2011). The only work I am currently aware of that has a goal similar to the overall dissertation is the Iris separation logic (Jung et al., 2015). The Iris logic is both language independent and expressive enough represent a wide array of kinds of specifications, including noninterference (Frumin et al., 2019). Unlike this work, Iris is focused on higher level languages, with features like arbitrary recursion, recursive types, and higher order state.

There is other literature that falls in the intersection of ITrees and specifications. Koh et al. (2019) uses ITrees to specify the set of acceptable observable behaviors from server code. Both Conditional Contextual Refinement (Song et al., 2023) and DimSum (Sammler et al., 2023) extend ITrees with quantifier events and use the resulting structure as a language of specifications.

6.2. Dijkstra Monads Forever

Work on creating logics to verify program specifications for effectful languages 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.

Chapter 3 builds directly on the Dijkstra monad literature (Maillard et al., 2019; Swamy et al., 2013). 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 provides only partial correctness guarantees, it is less expressive than the framework presented in Chapter 3.

The Dijkstra monad 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.

6.3. Secure Interaction Trees

Goguen and Meseguer (1982) introduced noninterference to formalize confidentiality; that is, the intuitive notion that secret data does not leak to an adversary. Volpano et al. (1996) enforce progress-insensitive noninterference with a type system, and Volpano and Smith (1997) modify the type system enforce progress-sensitive noninterference. These results led to a long line of work introducing noninterference to increasingly complicated settings (e.g., Myers and Liskov, 1998; Myers, 1999; Abadi et al., 1999; Zdancewic and Myers, 2002; Pottier and Simonet, 2003; Tsai et al., 2007; Russo et al., 2008; Rafnsson and Sabelfeld, 2014; Algehed and Russo, 2017; Milano and Myers, 2018; Vassena et al., 2018). Proving the security of these varied type systems led to complicated arguments for noninterference, but also gave rise to an informal library of proof techniques. The work in Chapter 5 fits into a tradition of proof techniques for noninterference via models.

Most models view noninterference either as a trace (hyper)property or as the result of an indistinguishability relation. These perspectives are not mutually exclusive; we can view two programs as indistinguishable if they produce equivalent traces. Their focus, however, can be quite different. Trace-based models view noninterference as a 2-safety hyperproperty (Clarkson and Schneider, 2010). That is, noninterference can be falsified using finite prefixes of two traces. Specifically, for any interfering program there are two inputs that differ only on secrets but produce distinguishable events after a finite number of steps.

Indistinguishability models focus more on building compositional relations. Pioneered by Abadi et al. (1999) and Sabelfeld and Sands (2001), these models use PERs and define secure programs as those that are self-related. Two such approaches have yielded recent notable results. First, logical-relations techniques (Reynolds, 1983) inductively assign each type a binary relation. By constructing the relation to reflect the security requirements of the type, logical relations can reason about information flow control and noninterference (Vassena et al., 2019; Rajani and Garg, 2018; Gregersen et al., 2021). Second, bisimulation approaches directly match up program executions to define indistinguishability (Smith, 2003; Focardi et al., 2002).

This work straddles these methods. ITrees intuitively collect all possible traces of a program into one infinite data structure. Our binary indistinguishability relation on ITrees is thus combining the hyperproperty model of noninterference with the indistinguishability model. Moreover, our indistinguishability relation is built on top of weak bisimulation. To give meaning to a type system, we also build a small logical relation connecting types to our bisimulation arguments.

To remain practical, many languages provide only progress-insensitive guarantees (e.g., Magrino et al., 2016; Liu et al., 2017; Volpano et al., 1996; Pottier and Simonet, 2003), despite the fact that termination channels can leak arbitrary amounts of data (Askarov et al., 2008). Techniques for enforcing progress-sensitive guarantees (Volpano and Smith, 1997; Sabelfeld and Myers, 2003) exist, but have seen little use. Recent work attempts to unify the two by explicitly considering termination leaks as declassifications (Bay and Askarov, 2020). Like other models of noninterference (Gregersen et al., 2021), `seutt`, defined in Section 4.4, is naturally progress-sensitive, giving a strong guarantee. Section 4.4 also includes the progress-insensitive `pi-seutt` to give ITree-based semantics to more-practical systems as well.

The reasoning principles in Chapter 4 apply to effectful languages and model effects as is standard with ITrees. The information-flow community also studies effects deeply since they can leak information. Traditionally, information-flow languages use a program-counter label to reason about effects, as we saw in Section 4.5. Recent work by Hirsch and Cecchetti (2021) connects program-counter labels with monads, giving the former semantics using the latter.

Secure compilation is a very active research area. For instance, Barthe et al. (2004) show how to securely compile to a low-level Asm-like target language. However, they use a type system for the target language to enforce security. Other efforts focus on particular language features, such as cryptographic constant time (Barthe et al., 2018). Moreover, until recently, most work on secure compilation focused on fully-abstract compilation Leroy (2009). Unfortunately, (Abate et al., 2019) recently showed that full abstraction is not sufficient to guarantee preservation of hyperproperties like noninterference. Our Mixed Transitivity theorems (Theorems 12 and 18) show that *equivalence-preserving* compilation does preserve noninterference.

Beyond work on secure compilation, most research on noninterference does not address multiple interacting languages. In one notable exception, Focardi et al. (2005) examine the relationship between a process-calculus–based notion of security and a language-based notion of security, using a simpl imperative language similar to Imp. They translate their version of Imp into CCS and show that they preserve Imp's security guarantees. However, their work contains only pencil-and-paper proofs, rather than formally verifying their translation or its security.

Finally, this work focuses on an approach for verifying language toolchains, but running any program requires hardware. Most language-based security and verification work assumes the hardware is predictable and reliable, but cannot enforce security. Hardware enforcement of information-security properties (Zhang et al., 2012; Zagieboylo et al., 2019) provides dynamic enforcement of properties like noninterference at the cost of space and power usage. Combining these mechanisms with our approach could reduce the overhead of hardware enforcement for verified-secure programs and provide a means to guarantee that interactions with unverified programs remain safe.

6.4. Interaction Tree Specifications

Chapter 5 builds on the work of Chapter 3, providing a second Dijkstra Monad for reasoning about interaction trees. ITree specifications 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 ITree specification 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.

Chapter 3 and Chapter 5 both provide a Dijkstra monad for ITrees that is expressive enough to reason about termination-sensitive properties. Both Dijkstra monads are capable of expressing specifications that allow for specifying infinite behavior. However, Chapter 3 does not provide reasoning principles for general recursion. The fact that ITree specifications represent specifications as ITrees enabled the creation of elegant reasoning principles for recursion. It also enabled to creation of an auto-active verification tool to greatly reduce the effort in writing proofs of specification refinement.

The ultimate goal of Chapter 5 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 (Appel, 2014) and CFML (Charguéraud, 2011). 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 OCaml, ITree specifications can be used to specify any programs with an ITrees based semantics. When paired with Heapster techniques (He et al., 2021), 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 pointer manipulation and heap layout specific reasoning, freeing the verifier to focus on the underlying mathematical structures. In separation logics like VST, proof engineers can invent a functional specification, prove it correct with respect to the original program, and then reason further about the functional specification. Heapster generates functional specifications automatically from Heapster typing derivations. This enables a verification pipeline where reasoning about memory safety can be fully separated from reasoning about functional correctness. However, this automation comes at a cost in the form of lessened expressiveness. The Heapster type system is conservative and will reject some programs whose behavior could be verified in a full separation logic like VST.

The primary innovation of Chapter 5 is to turn ITrees into a specification language by adding quantifier events. Similar ideas were independently developed in Song et al. (2023) and Sammler et al. (2023). In Sammler et al. (2023), ITrees with quantifiers are used to denote and specify programs with multilanguage features, i.e. linking code from two different languages. In Song et al. (2023), ITrees with quantifiers are used to model programs and specifications in a framework designed to have the common benefits of both refinement-based and separation logic-based specification frameworks. Common to both of these works is using ITrees to model *modules*, separately defined pieces of code that communicate through external calls. Both use events to model inter-module communication. One major distinction between this line of work and other ITrees work is precisely how ITrees are used to model computation. In most ITrees work, a program's control flow is made explicit with ITree combinators, and events are given semantics using handlers and interpretation, as described in Chapter 2. These papers give semantics to all events, quantifier or otherwise, with a novel simulation relation, reasoning about ITrees as a form of labelled transition system. The ITrees are then reasoned about in terms of a trace semantics. This approach has produced impressive results, but these kinds of ITrees semantics miss out on some common advantages of the ITrees approach that the work in this dissertation retains. The semantics in Song et al. (2023) and Sammler et al. (2023) are not executable even when they lack any quantifier events. This prevents such semantics from being used as a reference implementation, or from being tested with testing tools like QuickChick (Lampropoulos and Pierce, 2018; Lampropoulos et al., 2018). This approach is also too separate from other ITree work to take full advantage of it. For example, Chapter 5 makes heavy use of a recursion operator adapted from earlier ITrees work (Xia et al., 2020). As such, this chapter benefitted greatly from previous ITrees work which used this operator, and contributed to the body of work by showing how to reason about that operator. Recursion in Song et al. (2023) and Sammler et al. (2023) is novel to that work, and cannot be easily compared to recursion in previous ITrees work.

CHAPTER 7

Conclusion

7.1. Contributions

This dissertation presents reusable, language independent tools for different calls of specifications over programs with ITree semantics. Each tool leverages the flexibility of ITrees semantics to make sure that work can be reused across different programming languages. These tools are presented and discussed in Chapters 3, 4, and 5.

Dijkstra Monads Forever. Chapter 3 presents techniques for creating algebraic-effect aware specifications for programming languages that use ITree semantics. These techniques adapted the Dijkstra monad (Swamy et al., 2013; Ahman et al., 2017) techniques by providing a specification monad for ITrees. Chapter 3 also provides example specifications that can be represented in these specification monads.

Semantics for Noninterference with Interaction Trees. Chapter 4 presents termination sensitive and insensitive indistinguishability relations for ITrees. These relations enable reasoning about information flow in programming languages that use ITree semantics. Chapter 4 also shows how to use these relations to reason about a noninterference type system in a simple language with inlined assembly code.

Interaction Tree Specifications. Chapter 5 augments ITrees with logical quantifiers to serve as a language of specifications for ITrees. Chapter 5 also presents verified metatheory for reasoning about specification refinement along with an auto-active tool for proving specification refinement. The chapter also demonstrates how to use ITree specifications with the Heapster (He et al., 2021) tool to verify real C programs. This demonstration includes a comparison between verifying a particular example program with Heapster types and ITree specifications and verifying the same program in the VST separation logic (Appel, 2011).

7.2. Future Work

7.2.1. Dijkstra Monads Forever

Maillard et al. (2020) extends the Dijkstra Monad framework from dealing strictly with unary programming logics to dealing with relational programming logics. This allows the formalization of specifications that relate the behavior of two different programs assuming their inputs satisfy some input condition. The work accomplishes this by introducing the concept of a *simple relational specification monad*. Simple relational specification monads serve as the type of specifications for computations. These techniques may be able to be used to provide a useful relational program logic for ITrees.

In Chapter 3, the Dijkstra monad for ITrees is shown to be a valid and expressive domain for writing and verifying specifications. However, it remains unclear how effective this tool is for reducing the burden of verification of computations written with ITrees. Further research could investigate creating further tooling, for example a library of proof tactics, that utilizes the structure of Dijkstra monads to simplify this kind of reasoning. Investigating the tooling underlying the $F^\star$ (Swamy et al., 2011) programming language could be particularly useful, as this language uses Dijkstra monads along with refinement types to produce verified code.

Semantics for Noninterference with Interaction Trees.

Future work could investigate generalizations of the indistinguishability relations presented in this paper that are heterogeneous in the event type signature, more like `rutt` than `eutt`. This added flexibility would allow the formalization of reasoning principles for indistinguishability over mutually recursively defined computations, computed with the `mrec` combinator. This could enable reasoning about programming languages with mutually recursively defined functions.

Future work can also further investigate noninterference and interpretation, already partially discussed in Section 4.4.4. This section restricted attention to state. Using insights from Yoon et al. (2022), which presents monadic interpreters which generalize the `interp` function over ITrees, researchers may be able to create sound conditions for ensuring that handlers respect interpretation. That is, ensuring that handlers take indistinguishable source computations to indistinguishable target computations.

Interaction Tree Specifications.

Chappe et al. (2022) presented a variant of Interaction Trees designed to represent nondeterministically branching computations called Choice Trees. They accomplish this by adding a new constructor to ITrees that is analogous to an existential choice operator. Future work could investigate the connections between Choice Trees and ITree specifications, and answer the question of whether either subsumes the other. And if neither subsumes the other, then researchers could investigate adapting the ITree specifications to Choice Trees. This would serve as a step towards adapting the ITree specifications framework to concurrent programs.

APPENDIX A

Alternate Interaction Trees Definition

Figures A.1 and A.2 presented in this appendix present a definition of ITrees where events are represented by a single type rather than a type family, as discussed in Chapter 5. The definitions presented in this appendix are highly similar to counterpart definitions presented in Chapter 2. Please refer to that chapter for intuitive explanations.

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

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.

CoFixpoint interp_mrec {R : Type}
  (bodies : ∀(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.

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

Figure A.1: Alternate definitions for Interaction Trees and key operators

$$[\textsc{EuttRet'}]\ \begin{array}{c}\texttt{RR r1 r2} \\ \hline \texttt{eutt RR (ret } r1\texttt{) (ret } r2\texttt{)}\end{array} \qquad [\textsc{EuttTau'}]\ \begin{array}{c}\texttt{eutt RR t1 t2} \\ \hline\hline \texttt{eutt RR (Tau t1) (Tau t2)}\end{array}$$

$$[\textsc{EuttVis'}]\ \begin{array}{c}\texttt{∀a, eutt RR (k1 a) (k2 a)} \\ \hline\hline \texttt{eutt RR (Vis e k1) (Vis e k2)}\end{array} \qquad [\textsc{EuttTauL'}]\ \begin{array}{c}\texttt{eutt RR t1 t2} \\ \hline \texttt{eutt RR (Tau t1) t2}\end{array}$$

$$[\textsc{EuttTauR'}]\ \begin{array}{c}\texttt{eutt RR t1 t2} \\ \hline \texttt{eutt RR t1 (Tau t2)}\end{array}$$

Definition 29. *Given:*

- *an event signature `E`;*
- *return types `R1` and `R2`;*
- *and a return relation over `R1` and `R2`, `RR`,*

equivalence up to taus with `RR`, a relation between `itree E R1` and `itree E R2`, is defined with the inference rules presented in Figure A.2. We write this relation as `eutt RR t1 t2`.

Figure A.2: `eutt` definition for alternate Interaction Trees definition

```
Class ReSum (E1 : Type) (E2 : Type) `{EncodingType E1} `{EncodingType E2} :=
{
  resum : E1 → E2;
  resum_ret : ∀{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} :
  ∀ (e1 : E1), (itree E2 (response_type e1)) :=
    fun e ⇒ Vis (resum e) (fun x ⇒ Ret (resum_ret x)).
```

Figure A.3: `ReSum` definition for alternate Interaction Trees definition

BIBLIOGRAPHY

Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1999. doi: 10.1145/292540.292555.

Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *IEEE Computer Security Foundations Symposium (CSF)*, 2019. doi: 10.1109/CSF.2019.00025.

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.

Maximilian Algehed and Alejandro Russo. Encoding dcc in haskell. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2017. doi: 10.1145/3139337.3139338.

Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19717-8. URL http://dl.acm.org/citation.cfm?id=1987211.1987212.

Andrew W. Appel. *Program Logics - for Certified Compilers.* Cambridge University Press, 2014. ISBN 978-1-10-704801-0. URL http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB.

Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 375(2104), 2017. ISSN 1364-503X. doi: 10.1098/rsta.2016.0331. URL http://rsta.royalsocietypublishing.org/content/375/2104/20160331.

Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *IEEE Computer Security Foundations Symposium (CSF)*, July 2015. doi: 10.1109/CSF.2015.42.

Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *European Symposium on Research in Computer Security (ESORICS)*, pages 333–348. Springer, 2008. doi: 10.1007/978-3-540-88313-5_22.

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.

Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. *SIGPLAN Not.*, 49(1):165–178, January 2014. ISSN 0362-1340. doi: 10.1145/2578855.2535839. URL https://doi.org/10.1145/2578855.2535839.

Gilles Barthe, Amitabh Basu, and Tamara Rezk. Security types preserving compilation. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 2–15, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24622-0.

Gilles Barthe, Benjamin Greégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant time". In *IEEE Computer Security Foundations Symposium (CSF)*, 2018. doi: 10.1109/CSF.2018.00031.

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

Johan Bay and Aslan Askarov. Reconciling progress-insensitive noninterference and declassification. In *IEEE Computer Security Foundations Symposium (CSF)*, June 2020. doi: 10.1109/CSF49147.2020.00015.

Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3): 227–270, 2007. doi: 10.1016/j.tcs.2006.12.034. URL https://doi.org/10.1016/j.tcs.2006.12.034.

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. Vst-floyd: A separation logic tool to verify correctness of C programs. *J. Autom. Reasoning*, 61(1-4):367–422, 2018. doi: 10.1007/s10817-018-9457-5. URL https://doi.org/10.1007/s10817-018-9457-5.

Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005. ISSN 1860-5974. doi: 10.2168/LMCS-1(2:1)2005. URL http://www.lmcs-online.org/ojs/viewarticle.php?id=55.

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

Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. Choice trees: Representing nondeterministic, recursive, and impure programs in coq. Nov 2022. doi: 10.1145/3571254. URL http://arxiv.org/abs/2211.06863. arXiv:2211.06863 [cs].

Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. Choice trees: Representing nondeterministic, recursive, and impure programs in coq. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571254. URL https://doi.org/10.1145/3571254.

Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. *SIGPLAN Not.*, 46(9):418–430, sep 2011. ISSN 0362-1340. doi: 10.1145/2034574.2034828. URL https:

//doi.org/10.1145/2034574.2034828.

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.

Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security (JCS)*, 18(6):1157–1210, 2010. doi: 10.3233/JCS-2009-0393.

Thierry Coquand. An analysis of girard's paradox. 10 1999.

Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, jul 1977. ISSN 0001-0782. doi: 10.1145/359636.359712. URL https://doi.org/10.1145/359636.359712.

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. ISSN 0001-0782. doi: 10.1145/360933.360975. URL https://doi.org/10.1145/360933.360975.

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

Riccardo Focardi, Carla Piazza, and Sabina Rossi. Proof methods for bisimulation based information flow security. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2002.

Riccardo Focardi, Sabrina Rossi, and Andrei Sabelfeld. Bridging language-based and process calculi security. In *FoSSaCS*, 2005. doi: 10.1007/978-3-540-31982-5_19.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. Compositional non-interference for fine-grained concurrent programs. *CoRR*, abs/1910.00905, 2019. URL http://arxiv.org/abs/1910.00905.

Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy (S&P)*, 1982. doi: 10.1109/SP.1982.10014.

Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434291. URL https://doi.org/10.1145/3434291.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 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.*, pages 653–669, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg, and David Costanzo. Building certified concurrent os kernels. *Commun. ACM*, 62(10):89–99, September 2019. ISSN 0001-0782. doi: 10.1145/3356903. URL https://doi.org/10.1145/3356903.

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.

Andrew K. Hirsch and Ethan Cecchetti. Giving semantics to program-counter labels via secure effects. *Proceedings of the ACM on Programming Languages*, 5(POPL), January 2021. doi: 10.1145/3434316.

Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages*, 6(ICFP):711–741, Aug 2022. ISSN 2475-1421. doi: 10.1145/3547647. arXiv:2206.07185 [cs].

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. Testing noninterference, quickly. In *ICFP*, 2013. doi: 10.1145/2500365.2500574.

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.

Limin Jia and Steve Zdancewic. Encoding information flow in Aura. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 17–29, 2009.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 2015. doi: 10.1145/2676726.2676980.

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.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158154. URL https://doi.org/10.1145/3158154.

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.

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From c to interaction trees: Specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2019. doi: 10.1145/3293880.3294106.

Daniel Kästner, Ulrich Wünsche, Jörg Barrho, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS 2018: Embedded Real Time Software and Systems*. SEE, January 2018. URL http://xavierleroy.org/publi/erts2018_compcert.pdf.

Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, 2018. URL https://softwarefoundations.cis.upenn.edu/qc-current/index.html.

Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *PACMPL*, 2(POPL):45:1–45:30, 2018. doi: 10.1145/3158133. URL http://doi.acm.org/10.1145/3158133.

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

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.

Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. Fabric: Building open distributed systems securely by construction. 25(4–5):319–321, May 2017. doi: 10.3233/JCS-0559.

Tom Magrino, Jed Liu, Owen Arden, Chin Isradisaikul, and Andrew C. Myers. Jif 3.5: Java information flow. Software release, 2016. URL https://www.cs.cornell.edu/jif.

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.

Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. The next 700 relational program logics. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–33, Jan 2020. ISSN 2475-1421, 2475-1421. doi: 10.1145/3371072.

Gregory Malecha, Greg Morrisett, and Ryan Wisnesky. Trace-based verification of imperative programs with i/o. *J. Symb. Comput.*, 46(2):95–118, February 2011. ISSN 0747-7171. doi: 10.1016/j.jsc.2010.08.004. URL http://dx.doi.org/10.1016/j.jsc.2010.08.004.

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.

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.

Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2023. URL http://coq.inria.fr. Version 8.17.1.

Mae P. Milano and Andrew C. Myers. MixT: A language for mixing consistency in geodistributed transactions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018. doi: 10.1145/3192366.3192375.

Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 1999. doi: 10.1145/292540.292561.

Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy (S&P)*, 1998. doi: 10.1109/SECPRI.1998.674834.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. *SIGPLAN Not.*, 41(9):62–73, September 2006. ISSN 0362-1340. doi: 10.1145/1160074.1159812. URL https://doi.org/10.1145/1160074.1159812.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic.* Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3-540-43376-7.

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

Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3): 271–307, 2007. doi: 10.1016/j.tcs.2006.12.035. URL https://doi.org/10.1016/j.tcs.2006.12.035.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Logical Foundations.* Software Foundations series, volume 1. Electronic textbook, May 2018. Version 5.5. http://www.cis.upenn.

edu/~bcpierce/sf.

Gordon Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures*, pages 1–24, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45315-4.

Gordon D Plotkin. *A structural approach to operational semantics.* Aarhus university, 1981.

Gordon D Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, 9(4), December 2013. doi: 10.2168/LMCS-9(4:23)2013. URL https://lmcs.episciences.org/705.

Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. Liquid information flow control. *Proceedings of the ACM on Programming Languages*, 4 (ICFP), August 2020. doi: 10.1145/3408987.

François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):117–158, January 2003. doi: 10.1145/596980.596983.

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. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *IEEE Symposium on Security and Privacy*. IEEE, May 2020. URL https://www.microsoft.com/en-us/research/publication/evercrypt-a-fast-verified-cross-platform-cryptographic-provider/.

Willard Rafnsson and Andrei Sabelfeld. Compositional information-flow security for interactive systems. In *IEEE Computer Security Foundations Symposium (CSF)*, 2014. doi: 10.1109/CSF.2013.8.

Vineet Rajani and Deepak Garg. Types for information flow control: Labeling granularity and semantic models. In *IEEE Computer Security Foundations Symposium (CSF)*, 2018. doi: 10.1109/CSF.2018.00024.

John Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, 1983.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74, 2002. doi: 10.1109/LICS.2002.1029817. URL https://doi.org/10.1109/LICS.2002.1029817.

Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *ACM SIGPLAN Haskell Symposium*, 2008. doi: 10.1145/1411286.1411289.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003. doi: 10.1109/JSAC.2002.806121.

Andrei Sabelfeld and David Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001. doi: 10.1023/A:1011553200337.

Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. Dimsum: A decentralized approach to multi-language semantics and verification. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571220. URL https://doi.org/10.1145/3571220.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. *SIGPLAN Not.*, 50(6):77–87, June 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737964. URL https://doi.org/10.1145/2813885.2737964.

Lucas Silver and Zdancewic. Dijkstra monads forever: Termination-sensitive specifications for interaction trees. Nov 2020. doi: 10.5281/zenodo.4312937.

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.

Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. Semantics for Noninterference with Interaction Trees (Artifact). *Dagstuhl Artifacts Series*, 9(2):6:1–6:2, 2023a. ISSN 2509-8195. doi: 10.4230/DARTS.9.2.6. URL https://drops.dagstuhl.de/opus/volltexte/2023/18246.

Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. Semantics for Noninterference with Interaction Trees. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:29, Dagstuhl, Germany, 2023b. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-281-5. doi: 10.4230/LIPIcs.ECOOP.2023.29. URL https://drops.dagstuhl.de/opus/volltexte/2023/18222.

Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott. Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations That Supports Auto-Active Verification. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:26, Dagstuhl, Germany, 2023c. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-281-5. doi: 10.4230/LIPIcs.ECOOP.2023.30. URL https://drops.dagstuhl.de/opus/volltexte/2023/18223.

Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott. Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations That Supports Auto-Active Verification (Artifact). *Dagstuhl Artifacts Series*, 9(2):8:1–8:2, 2023d. ISSN 2509-8195. doi:

10.4230/DARTS.9.2.8. URL https://drops.dagstuhl.de/opus/volltexte/2023/18248.

Geoffery Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Computer Security Foundations Workshop (CSFW)*, 2003. doi: 10.1109/CSFW.2003.1212701.

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. Conditional contextual refinement. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi: 10.1145/3571232. URL https://doi.org/10.1145/3571232.

Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Nordic Conference on Security IT Systems (NordSec)*, October 2011. doi: 10.1007/978-3-642-29615-4_16.

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.

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.

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*. *SIGPLAN Not.*, 51(1):256–270, jan 2016. ISSN 0362-1340. doi: 10.1145/2914770.2837655. URL https://doi.org/10.1145/2914770.2837655.

Tsa-ching Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow in haskell. In *IEEE Computer Security Foundations Symposium (CSF)*, 2007. doi: 10.1109/CSF.2007.6.

Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. MAC: A verified static information-flow control library. *Journal of Logical and Algebraic Methods in Programming (JLAMP)*, 95, 2018. doi: 10.1016/j.jlamp.2017.12.003.

Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. From fine- to coarse-grained dynamic information flow control and back. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019. doi: 10.1145/3290389.

Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 1997. doi: 10.1109/CSFW.1997.596807.

Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security (JCS)*, 4(3), 1996. doi: 10.3233/JCS-1996-42-304.

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.

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.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993532. URL https://doi.org/10.1145/1993498.1993532.

Irene Yoon, Yannick Zakowski, and Steve Zdancewic. Formal reasoning about layered monadic interpreters. *Proceedings of the ACM on Programming Languages*, 6(ICFP):254–282, Aug 2022. ISSN 2475-1421. doi: 10.1145/3547630.

Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. Using information flow to design an ISA that controls timing channels. In *IEEE Computer Security Foundations Symposium (CSF)*, June 2019. doi: 10.1109/CSF.2019.00026.

Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for llvm ir. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021a. doi: 10.1145/3473572. URL https://doi.org/10.1145/3473572.

Yannick Zakowski, Calvin Beck, Irene Yoon, Ilya Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for llvm ir. *Proceedings of the ACM on Programming Languages*, 5(ICFP), 2021b.

Steve Zdancewic and Andrew C Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2-3), 2002. doi: 10.1023/A:1020843229247.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. *Communications of the ACM*, 54(11):93–101, November 2011. doi: 10.1145/2018396.2018419.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2012. doi: 10.1145/2254064.2254078.