MODULAR SEMANTICS AND METATHEORY FOR LLVM IR

Euisun Yoon

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 A. Zdancewic, Schlein Family President's Distinguished Professor of Computer and Information Science

Graduate Group Chairperson
Mayur Naik, Professor of Computer and Information Science

Dissertation Committee
Benjamin C. Pierce, Henry Salvatori Professor of Computer and Information Science
Stephanie Weirich, ENIAC President's Distinguished Professor of Computer and Information Science
Val B. Tannen, Professor of Computer and Information Science
Derek Dreyer, Scientific Director, Max Planck Institute for Software Systems

MODULAR SEMANTICS AND METATHEORY FOR LLVM IR

ACKNOWLEDGEMENT

Thank you to my wonderful thesis advisor, Steve Zdancewic. His expansive technical range and expertise, paired with his sharp intuition and openness, makes him an advisor that can grant an impressive degree of freedom in research. I was able to work in a relaxed and flexible environment during my Ph.D. thanks to his persistent trust and support.

Thank you to the computer scientists who will be permanent examples for me. Thank you to Benjamin Pierce, who shows you can do it all, and all of it with care. Thank you to Adrian Sampson, who taught me that scientific research is a spunky ordeal. Thank you to Stephanie Weirich, for the unbeatable boardgame nights and her inspiring craftsmanship in programming. Thanks to Bob Harper, a *plein-d'esprit* scientist who snipes students down with pretty proofs.

Thank you to those who hosted me during my Ph.D. Thank you to Derek Dreyer, who has been welcoming and insightful during my internship at MPI-SWS and visits to Saarbrücken. Thank you to my brilliant collaborators at MPI-SWS: Simon Spies, Lennard Gäher, and Youngju Song. Thank you to the compilers team at Jane Street for a pleasant stay in London.

Thanks to all the vibrant members of PLClub. I will not forget the whimsical hallway encounters of Levine Hall, ranging from my cereal-loving office neighbor's accusatory designation of myself as a rodent ("You are like a *mouse*, squeak, squeak!"), to conversations hinting about how NASA fire-fighting can be solved through sheaf theory.

Thank you to my family. Thank you to the artists, Jisang Kim and Shinja Kim, who taught me to live freely. Thank you to my father, for his love of books, writing, and cycling. Thank you to my brother for being himself. Thank you to my loving grandfather.

Thank you to my friends in Philadelphia. I would not be here without Paul He, who is North America's leading Gwen Stefani and ALDI enthusiast, and a warm, caring friend. Thanks to fellow witch-survivor Linh Hoang, who knows how to throw together the best picnics. Thanks to the pyromancer Stephen Mell for his undying enthusiasm.

Thank you infinitely to my dearest *petit-pote*, Yannick Zakowski.

ABSTRACT

MODULAR SEMANTICS AND METATHEORY FOR LLVM IR

Euisun Yoon

Stephan A. Zdancewic

The appealing guarantees of formally verified software comes in tandem with the high cost of verification. To reduce the cost of formal verification, modularity is crucial because it eases both the elaboration and reuse of proofs. This thesis focuses on developing a modular semantics and metatheory for realistic low-level languages, with a focus on LLVM IR. First, we define VIR, a modular and executable semantics for a large sequential subset of LLVM IR, which is based on layered, monadic interpreters. Unlike a traditional small-step semantics, VIR has an executable semantics which can be extracted into an executable definitional interpreter. Second, we develop a formal metatheory for reasoning about layered interpreters, giving an extensible theory for lifting interpreters and structural rules, characterizing interpretable monads and a relational reasoning framework for reasoning about equivalences across interpretation. Finally, we develop a relational separation logic framework for verifying program transformations on VIR, with a fresh perspective on verifying transformations with external calls.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Chapter 1

Introduction

With great computational power comes great responsibility. The history of computing unveiled the cost of errors in safety-critical systems. In the 1980s, the Therac-25 radiation therapy machine administered approximately 100 times the intended dose of radiation due to a software bug, resulting in lethal radiation poisoning. In 1996, an integer overflow error led to the self-destruction of Ariane flight V88 [LL97], leading to an estimated loss of $370 million USD. In 2018, Meltdown [LSG+18] and Spectre [KHF+19] was discovered, showcasing that new classes of vulnerabilities in widespread commercial microprocessors can be found in the modern day. The reliability of systems is imperative, especially in the ever-evolving landscape of computing.

The pursuit of reliability has led to the development of *formal methods*. In complex systems, software bugs occur despite intensive testing, causing erroneous program behavior at runtime or introducing subtle bugs that are difficult to track down. The consequences can be negligible for low-assurance software, but safety-critical systems need a guarantee that it will be bug-free. Formal methods, such as model checking [ELN+13, Lam99], static analysis [APH+08], and proof-carrying code [Nec97], give mathematical models to the correctness of program behavior. It gives a robust specification, analysis, and verification of software systems. Among these technologies, *formal verification* gives a strict form of mathematical certainty that is suitable for critical software.

Formal verification can demonstrate the *absence of bugs* in programs, and this rigor is especially useful for high-assurance systems. Formal verification is the process of giving *machine-checked* formal proofs about the behavioral correctness of programs. A formal proof in a logical system is a sequence of propositions that is derived from a set of axioms and rules of inference. The logical system can be seen as a meta-language for mathematical proofs, and its rules of inference can be written as a program. *Theorem provers* are software that decides the validitiy of proofs, where its kernel represents the logical system, and proofs are represented as decidable programs that are validated in the kernel.

CompCert [Ler09], the first formally verified optimizing C compiler, demonstrated that mechanical

verification is possible and essential for showing the correctness of realistic software. For its mechanized proofs, CompCert used the Coq proof assistant both for programming the compiler and for proving its soundness. A study by Yang *et al.* [YCER11] uncovered major bugs present in mainstream compilers such as GCC and LLVM, but CompCert was found to be "seemingly unbreakable", where bugs were only discovered in unverified parts of the compilation chain. Since its development, various verification projects and tools have emerged, such as the CertiKOS certified operating system [GSC+16], a verified web server [KLL+19], the verified software toolchain for C [App11], and the CakeML bootstrapped certified compiler [KMNO14].

The appealing guarantees of formally verified software come in tandem with the high cost of verification. To reduce the cost of formal verification, **modularity** is crucial because it eases both the elaboration and reuse of proofs. This thesis focuses on developing a **modular semantics and metatheory** for realistic low-level languages, specifically LLVM IR.

1.1 Modular Verification for LLVM IR

The LLVM compiler infrastructure is an attractive target for formal verification. It was initially developed as a research tool with the goal of exploring advanced compiler optimizations and modern compilation techniques. The LLVM framework has since evolved into a robust compiler infrastructure that is widely used in both academic and commercial settings. One of the key strengths of LLVM is its portability—the framework is designed to be platform-independent, making LLVM a versatile choice for compiler development. LLVM's portability is achieved through its intermediate representation (IR), which acts as a common language that can be used by different front-ends and back-ends. This dissertation builds a modular foundation for reasoning about program transformations in LLVM IR.

The high-level contributions of this dissertation are the following:

- A *modular* and *executable* semantics for the sequential semantics of LLVM IR, based on layered, monadic interpreters.
- A formal metatheory for reasoning about layered interpreters, giving an extensible theory for lifting interpreters and structural rules.
- A relational separation logic framework for verifying program transformations which has modular

reasoning about state, with a fresh perspective on verifying transformations with external calls.

Each of these contributions are divided into parts, which are described further below.

Part I. A Layered Semantics. VIR, *A modular and executable formal semantics for LLVM IR.*

Formalizing and understanding the semantics of LLVM IR is a substantial effort, and an area of active research [ZNMZ12, LKS+17, KKS+18, LG20b]. The first mechanized formal semantics of LLVM IR, Vellvm [ZNMZ12], was based on a small-step operational semantics, a similar approach to CompCert. During my Ph.D., I have contributed to developing a novel semantics of LLVM IR: *VIR*, a modular semantics for LLVM IR which gives a reference interpreter for free. The modular semantics constructs a separation of concerns between different side-effects of the language. Unlike a traditional small-step semantics, VIR has an *executable* semantics which can be extracted into an executable definitional interpreter. The extracted interpreter removes the additional burden of maintaining and proving the correspondence between the formal semantics and interpreter.

Part II. A Layered Equational Framework. eqmR, *An extensible metatheory for layered semantics.*

The VIR semantics uses *layered* interpreters, which give a modular definition for the semantics of LLVM IR. When structuring a semantics as complex as LLVM IR, interpretation takes place in layers: several interpreters are successively composed, each handling different side-effects of the language. There are issues that arise when *building* and *reasoning* about complex interpreter stacks, which I have resolved by characterizing a rich class of *interpretable monads* and a relational reasoning framework for reasoning about equivalences across interpretation.

Part III. A Separation Logic. Velliris, *A relational separation logic framework for verifying program transformations in LLVM IR.*

In Part I, demonstrated proofs of optimizations reasoned only about control flow and did not involve stateful transformations. For larger pieces of code, or for optimizations involving *state* change, proofs of optimizations become quickly excruciating without a method to support localized reasoning about state. In fact, one could not use the VIR semantics to adequately reason about larger software because it was flawed: it assumed external function calls do not change state. I have developed a separation logic framework for verifying program transformations that resolves these concerns. It is based on the Iris framework

[JKJ+18], particularly in the style of Simuliris [GSS+22], and extends the semantics of external calls for VIR. In particular, I developed logical specifications of external calls with LLVM IR *function attributes*, which specify the possible memory effects of the call-site or function. This results in a modular logical framework for reasoning about stateful LLVM IR programs, and a novel method of proving program transformations correct under the presence of external calls.

1.1.1 Credits

This thesis work is joint work with many collaborators, as listed below, each tied to a publication or a work in preparation for submission.

- Part I: 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.
- Part II: Irene Yoon, Yannick Zakowski, 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.
- Part III: Irene Yoon, Simon Spies, Lennard Gäher, Youngju Song, Derek Dreyer, Steve Zdancewic. Velliris: A Relational Separation Logic for LLVM IR. (In submission.)

1.1.2 Note on Mechanization

All results in this dissertation have been mechanized in the Coq proof assistant and are open-source software. Each part consists of substantial new pieces of mechanized proofs, and are available as a software artifact tied to the publication corresponding to each part.[1] Part I: ~30K LOC, Part II: ~10K LOC (without the ~8K LOC case study), and Part III: ~30K LOC.[2] My personal technical contributions are the following: for Part I,

[1] Part I: https://zenodo.org/records/4777179, Part II: https://zenodo.org/records/6913915, and Part III is in preparation for artifact submission, and its DOI will be available publicly along with the publication of Part III.

[2] Lines of code using the cloc tool, disregarding mechanization not relevant to this thesis. Since size and complexity of software cannot be measured solely with lines of code, we present this metric here only as an accessible measure of proof effort. All technical details necessary to understanding this dissertation are typeset.

proofs related to memory and pick events. The variant of VIR semantics discussed in this dissertation also diverges slightly from the presentation in Zakowski et al., as described in Part I. For Part II, I am the lead developer, and for Part III, I am responsible for the complete mechanized proof.

Chapter 2

Background: Monadic Interpreters and Interaction Trees

Since their inception, monads and monadic interpreters have been recognized as appealing and mathematically elegant ways to define programs and their semantics, especially in the presence of I/O, state, nondeterminism, failure, or other effects [Mog89a, Ste94, LH00, LHJ95]. The monad laws, suitably extended with domain-specific equations that capture the semantics of effects, enable reasoning about the equivalence of monadic programs, and, more generally, yield powerful relational program logics (such as Dijkstra monads [AHM+17, MAA+19, MHRVM20]) that can be used to prove properties ranging from the correctness of program optimizations to information-flow noninterference [Ben04].

It is no surprise, then, that when it comes to *formalizing* the behavior of complex language semantics or the behavior of interactive systems, monads play a crucial role. They are particularly well suited for defining the semantics of effects when the metalanguage is *pure* and *total*, which is the case when embedding a language semantics into dependent type theory, such as in Coq. Moreover, by working with *free monads* [Swi08] and monadic interpreters, one can obtain a flexible, general-purpose reasoning framework for effectful computations. Variations of this idea have appeared throughout the literature, for instance as the *program monad* in the FreeSpec project [LRGCH18], as *I/O-trees* [HS00], and as McBride's *general monad* [McB15].

In this dissertation, we focus on *interaction trees* [XZH+20] (ITrees), a recent realization of this approach as a Coq library. Interaction Trees [XZH+20] (ITrees) are a data structure that represents effectful and potentially divergent computations. ITrees let us define (in Coq) domains for building compositional (denotational) semantics of languages; they modularize the *effects* of such a semantics while still retaining executability.

ITrees are defined as a coinductive variant of the *freer monad* [KI15] and are also closely related to *resumption monads* [PG14]. The core data structure, `itree E`, represents a computation as a possibly-infinite tree with nodes labeled by events `e` drawn from an event signature `E`. An event `e` can be thought of as a point at which the computation interacts with its environment, allowing the environment to supply a

response `r`. The node corresponding to `e` in the ITree also has the continuation of the computation, given as a function of `r`.

2.1 Interaction Trees: A Free Monad Supporting General Recursion

Interaction Trees are a data structure for representing computations interacting with an external environment through *visible events*. It has a coinductive datatype, modeling potentially diverging computations. Unlike other ways of specifying semantics in Coq (*e.g.* relational operational semantics), ITrees can be extracted into executable programs.

The definition of the ITree datatype, as well as the type signatures of its main combinators, are shown in Figure 2.1.[3] The datatype takes as its first parameter a signature—described as a family of types `E : Type → Type`—that specifies the set of interactions the computation may have with the environment. The `Vis` constructor builds a node in the tree representing such an interaction, followed by a continuation indexed by the return type of the event. The second parameter, `R`, is the *result type*, the type of values that the computation may return if it halts. The constructor `Ret` builds such a pure computation, represented as a leaf. Finally, the `Tau` constructor models an internal, non-observable step of computation, allowing the representation of silently diverging computations; `Tau` is also used for guarding corecursive definitions.

ITrees are equipped with four main primitive combinators. As expected, `itree E` forms a monad at any signature `E`: pure computations can be embedded with `ret`, and computations can be sequenced with `bind`. The `trigger e` combinator builds the minimal computation performing the event `e`, which immediately returns the answer from the environment.[4] Finally, ITrees support fixed-point combinators such as `iter` which encodes terminal recursion.

To illustrate how to model computations with ITrees, consider a signature describing printing on one hand, and interaction with a single memory cell storing a natural number on the other. The cell can be read or updated, and values can be sent to the external printer: notice how each event specifies the nature of the answer it expects from the environment in the index type.

[3] The signature of ITrees is presented with a positive coinductive datatype for expository purposes. The actual implementation is defined in the negative style.

[4] In Section 5.2, we introduce a more general version of `trigger`, and the overloading is handled by module namespaces (i.e. this ITree-specific trigger will be referred as `ITree.trigger`)

```
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). (* event e yielding an answer in A *)

Notation "E ⤳ F" := (∀ X, E X → F X).
(* Embedding of pure computations *)
Definition ret {E : Type → Type} {R : Type} (v : R) : itree E R.
(* Sequencing computations *)
Definition bind {E : Type → Type} {T U : Type} (u : itree E T) (k : T → itree E U) : itree E U.
(* Atomic itrees triggering a single event. *)
Definition trigger {E : Type → Type} : E ⤳ itree E.
(* Fixed-point combinator *)
Definition iter {E : Type → Type} {R I: Type} (body : I → itree E (I + R)) : I → itree E R.
```

Figure 2.1: Interaction trees: definition and type signature of its main combinators

```
Variant printE :=
| Print : nat → printE unit.
```

```
Variant cellE :=
| Get : cellE nat
| Put : nat → cellE unit.
```

A computation that writes the value 3 to the cell, reads the content of the cell, and prints it to `stdout` can be represented as follows:[5]

```
_ ← trigger (inr1 (Put 3));; x ← trigger (inr1 Get);; trigger (inl1 (Print x))
```

This computation has type `itree (printE +' cellE) unit`, where `+'` is the *disjoint sum* operator.

ITrees are an implementation of the freer monad with a coinductive model of divergence. The events contained in a tree are uninterpreted; they assume no predetermined semantics. For instance, the traditional algebraic law ensuring that the `Get` operation in the previous example should return 3 is not accounted for at this stage. Such semantics of the effects manipulated is given in a separate step that enriches the structure by interpreting events into appropriate monads.

The notion of equivalence of computations over interaction trees (before interpretation) is a weak bisimulation observing the uninterpreted events and the returned values. This relation is referred to as *equivalence up-to taus*, or `eutt` for short, and ensures co-termination and trace equivalence. Congruence, monadic, and iterative laws are proved with respect to `eutt`.

The iterative laws used in ITrees, which imply that continuation trees of type `A → itree E B` form a traced monoidal category [BÉ93], can be also generalized for any arbitrary monad. It corresponds to Kleisli arrows (i.e. functions of type `A → M B` given a monad `M`) forming a traced monoidal category. We call any such monad which satisfies the iterative laws an *iterative monad*.

[5] We write `(x ← t;; k)` and `t >>= k` as notations for `(bind t (fun x ⇒ k))` and `(bind t k)`.

As a Coq library, ITrees come with a rich equational theory of equivalences *up-to-tau*, i.e. up-to the weak bisimulation that observes the uninterpreted events performed by the computations, the pure values they returned, and their potential divergence. This notion of weak equivalence is central to the verification of correctness of program transformations.

2.1.1 Monadic Implementation of Effects

The modularity of ITree-based semantics is embodied by the `interp` function. Through `interp`, a *handler*, which maps events into an iterative monad, can be freely lifted to a whole tree, essentially folding over the tree to produce a monadic computation.

To illustrate this idea, consider a handler that implements the memory cell events via a state monad, while leaving print events as uninterpreted.

```
Definition handle_cell : printE +' cellE ~> stateT nat (itree printE) :=
fun _ e n => match e with
           | inl1 (Print x) => trigger Print x;; Ret (n, tt)
           | inr1 Get       => Ret (n, n)
           | inr1 (Put m)   => Ret (m, tt)
           end.
```

This handler gives a semantics to `printE` and `cellE` events through pattern matching on the sum type. Such handlers can then be lifted by `interp`.

```
Definition interp_cell : itree (printE +' cellE) ~> stateT nat (itree printE) :=
  interp handle_cell.
```

In particular, computations for which all events are interpreted into an executable structure can be extracted as (potentially divergent) executable interpreters.

Compositional Semantic Combinators ITrees are monads and they support rich fixed point combinators, allowing for compositional definitions of a wide range of semantics. We write `ret` r for the pure monadic "return" operation; the `bind` operator composes two ITrees sequentially. We write $x \leftarrow t$;; $k(x)$ for `bind` t $(\lambda x\,.\,k(x))$. The `trigger` e operator, defined by `trigger` $e \triangleq$ `Vis` e $(\lambda x\,.\,$`Ret` $x)$ invokes the event e, yielding the answer from the environment.

The event signatures used by ITrees compose—a feature we exploit heavily in VIR. Given two event type E and F, we can form their disjoint union $E \oplus F$. Intuitively, an ITree of type `itree(E` $\oplus$ `F) R` can trigger events from either E or F.

Fixed-point combinators allow for modeling loops and recursive programs. The `iter` combinator allows for conveniently modeling iteration and tail recursive calls. Consider its type: `iter` $(body : A \to$ `itree` $E\ (A \oplus B)) : A \to$ `itree` $E\ B$. Here, A can be thought of as the type of an accumulator parameterizing the body of the iterator. Executing the body may result in either a new accumulator value which signals that the body should be executed again, or in a value of type `B`, signaling that the iteration has terminated. For non-tail-recursive calls, ITrees support a general combinator for mutually recursive computations, `mrec` $(defs : D \rightsquigarrow$ `itree` $(\mathtt{D} \oplus \mathtt{E})) : D \rightsquigarrow$ `itree` `E`,[6] where a D event represents a call to one of the mutually defined functions whose behaviors are given by *defs*. Besides recursive calls in D, the functions might trigger other events, E. The `mrec` combinator ties the recursive knot and returns computations only interacting through E.

Executable Semantics through *Coq Extraction* Lastly, ITrees are *executable*: they can be extracted to OCaml in order to be run. We exploit this property to derive the reference interpreter for LLVM described in the next chapter.

[6] We use $E \rightsquigarrow M$ for the polymorphic type $\forall \alpha, E\ \alpha \to M\ \alpha$ and leave most instantiations of the type parameter implicit.

Part I

A Layered Semantics for LLVM IR

Chapter 3

Modular Semantics for LLVM IR

3.1 Introduction

The CompCert C compiler [Ler09] was pivotal to the history of verified compilation, paving the way to large-scale software verification of real-world programming languages [RPS+19a]. Its introduction provided the backbone for a variety of innovative technologies [App11, GSC+16, ŠevčíkVN+13, SCK+19, BBG+20] and energized similar verification efforts for other programming languages [KMNO14, ZNMZ12, BCF+14, JJKD17].

Most of these projects define the semantics of the programming language using *relationally-specified transition systems* given by *small-step operational semantics*. Roughly speaking, such semantics are defined by a predicate `step` : *config* → *config* → ℙ, where ℙ is the type of propositions and `step c1 c2` means that configuration `c1` can transition to configuration `c2`. Importantly, the relationship between `c1` and `c2` is typically *not* expressed as a function that computes `c2` from `c1`, so this relation isn't "executable" in the sense that there is no way to extract code that would implement this step behavior. To say how a program evolves over time, one needs to consider many small steps: `step c1 c2` then `step c2 c3`, *etc.*, to finally halt at some configuration or go on stepping forever. From a proof-technique standpoint, these approaches often rely on (backward) simulations that connect the behavior of one `step` relation to another relation `step′`, which requires carefully crafting elementary simulation diagrams and stitching them together co-inductively to obtain termination-sensitive results.

These techniques have had widespread success; however, they also have some drawbacks. First, they often lack *compositionality*: the desired small-step operational semantics is not usually definable purely by induction on syntax. Second, and relatedly, they often lack *modularity*: side effects of the language become reified in the step relation, often leading to additional components such as program counters, heaps, or pieces of program text that are needed to define the relation but complicate the invariants needed to reason about it. Finally, because a relational model is not *executable*, it is difficult to test the language semantics

during its development, which is a useful way to validate the model's correctness. Lack of executability also precludes the use of tools like QuickChick [LP18]. An alternative is to write painstakingly hand-crafted interpreters—CompCert [Ler09], Vellvm [ZNMZ12], and JSCert [BCF+14] went to significant lengths in this regard—but that incurs the additional burden of proving (and maintaining) the correspondence between the operational semantics and the interpreter.

Compositionality, modularity, and executability are critical to ease the design, development, and upkeep of a formal language semantics, especially for large "real world" languages whose features are complex and evolving over time. In this dissertation, we demonstrate how to achieve these properties simultaneously and at scale: we formalize in Coq a large and expressive subset of the sequential portion of the LLVM. To do so, we draw on classic ideas about how to structure monadic interpreters [Ste94] and make heavy use of *interaction trees* [XZH+20], a recent Coq formalism that provides (1) expressive monadic combinators for defining compositional semantics, (2) effect handlers for the modular interpretation of effectful programs, and (3) a coinductive implementation that can be extracted into an executable definitional interpreter. These features allow for a strong separation of concerns: each syntactic sub-component can be given a self-contained meaning, and each effect of the language can be defined in isolation via an effect handler.

Moving away from traditional small-step operational semantics to an ITrees-based semantics not only simplifies the language definition, but also allows us to explore alternative means of proving compiler and optimization correctness properties. In particular, ITrees support a rich theory of refinement that facilitates relational reasoning proofs, much in the style of Maillard *et al.*'s Dijkstra monads [MAA+19], Swierstra and Baanen's predicate transformers [SB19] or Benton's relational Hoare logic [Ben04], letting us prove program equivalences largely by induction and elementary rewriting. Though some of the relevant theory was presented in the paper by Xia *et al.* [XZH+20], nondeterminism in the LLVM IR prompted us to develop new machinery for working with "propositional interpreters," a key ingredient needed to establish the proof of adequacy of the extracted interpreter.

We focus on the LLVM framework [LA04] because it is an attractive target for formal verification: it is a widely used, industrial-strength codebase; its intermediate representation (IR) provides a comparatively small and reasonably well-defined core language; and many of its analyses, program transformations, and optimizations, operate entirely at the level of the LLVM IR itself. Since the LLVM ecosystem supports

many source languages and target platforms, it is a natural fulcrum to amplify the impact of formal modeling and verification efforts. Moreover, there is ample existing work that aims to build formal semantics for (oftentimes just parts of) the LLVM IR. Notable examples include the Vellvm [ZNMZ12, ZNMZ13], Alive [LMNR15, MN17], Crellvm [KKS+18], and K-LLVM [LG20a] projects, as well as attempts to characterize LLVM's undefined behaviors [LKS+17], its concurrency semantics [CV17], and memory models [KHM+15a, LHJ+18b]. As witnessed by research activity surrounding it, LLVM IR's semantics isn't straightforward to specify, or even necessarily well-defined. Features like poison, undef, and integer–pointer casts, are complicated to model independently, and even more so together. We believe LLVM IR's complexities make it all the more important to formalize. While the semantics we present here is not the final word on the subject—most notably, the current memory model is not adequate for justifying some useful LLVM IR optimizations—we believe that we have developed the semantic ingredients needed to (eventually) define a "complete" model. Moreover, the emphasis we have put into the modularity of our semantics shall allow us to improve its quality over time to better approach (and react to changes in) "the" LLVM IR semantics.

The new VIR (Verified IR) development described here aims to fill the same niche as Vellvm, sharing that project's goal of being a platform for verified LLVM optimizations and compilers, but incorporating the insights of the works mentioned above and built using modern proof engineering-techniques—in particular, ITree-based monadic semantics form its core specification technology. While the work by Xia *et al.* demonstrated ITrees in a "toy" setting, here we aim to use them *at scale*—our treatment of LLVM's phi-nodes, mutually recursive functions, undef values, pointers, and other rich data types is all new in comparison. As such, our results also provide a novel and useful recipe for how to formalize large, complicated language semantics in theorem provers based on dependent type theory. In summary, this dissertation makes several contributions:

VIR Design We present VIR, a compositional, modular and executable formal semantics in Coq for a realistic sequential subset of LLVM IR. The semantics exhibits a principled structure, easing its development. VIR's syntax is structurally represented as interaction trees that distinguish different effects: local environment, stack, global identifiers, memory model, nondeterminism, external function calls, *etc.*. These effects are implemented by independent event handlers in the style of algebraic effects [PP03a] and composed

together with no additional syntax. We give a novel semantic model that is defined in terms of a fully "propositional" specification to capture the nondeterministic quirks of the language, but we also implement an executable reference interpreter that shares almost all of the code with the propositional semantics. Sections 3.2–3.3 describe this design, introducing the requisite background about ITrees along the way.

Metatheory We demonstrate how the compositional semantics gives rise to a primitive, but very expressive relational proof method, enabling termination-sensitive refinements of programs to be established without the use of explicit simulation diagrams or coinduction. The model justifies a definition of "correct program transformation" that can be proved at different levels of abstraction, leveraging the modularity of the semantics. In particular, programs that do not involve non-deterministic features can be reasoned about from the perspective of a deterministic semantics. This general-purpose proof infrastructure—many of our metatheoretic results apply to interaction tree semantics broadly and are not specific to VIR—also lets us prove the correctness of the VIR executable interpreter with respect to the model almost for free. Section 4.1 covers these results.

As alluded to above, there is a large body of prior work from which we draw inspiration. Section 4.3 compares our approach to the closest.

3.2 VIR Syntax

The primary focus of this dissertation is the use of monadic interpretation of interaction trees to define and reason about a compositional, modular, and executable semantics for a "real-world" programming language as exemplified by LLVM IR. Our formal development[7] covers most features of the core sequential fragment of LLVM IR 11.0.0 as per its informal specification[8], including: the basic operations on 1-, 8-, 32-, and 64-bit integers, `Doubles`, `Floats`, structs, arrays, pointers, and casts; `undef` and `poison`; SSA-structured control-flow-graphs, global data, mutually-recursive functions, and support for intrinsics. The main features that are currently unsupported are: some block terminators (`switch`, `resume`, indirect branching, `invoke`), the `landing_pad` and `va_arg` instructions, architecture-specific floats and opaque types. The list of supported intrinsics is small, but user-extensible. From a semantics perspective, the main limitation of VIR

[7] Available at https://github.com/vellvm/vellvm
[8] https://llvm.org/docs/LangRef.html

$$
\begin{aligned}
\tau &::= \texttt{i64} \mid \texttt{i1} \mid [\tau] \mid \tau* \\
\mathit{id},\ \mathit{bid} &::= \mathit{string} \\
\mathit{exp} &::= @\mathit{id} \mid \%\mathit{id} \mid \mathit{i64} \mid \mathit{i1} \mid \texttt{undef}_\tau \mid \mathit{exp}\ \texttt{op}\ \mathit{exp} \\
&\quad \mid \texttt{GEP}\ (\tau_1,\ \mathit{exp},\ \mathit{list}\ \mathit{exp}) \\
\mathit{instr} &::= \mathit{exp} \mid \texttt{call}\ (\mathit{exp},\ \mathit{list}\ \mathit{exp}) \mid \texttt{alloca}\ (\tau) \\
&\quad \mid \texttt{load}\ (\tau,\ \mathit{exp}) \mid \texttt{store}\ (\mathit{exp},\ \mathit{exp}) \\
\mathit{term} &::= \texttt{branch}\ (\mathit{exp},\ \mathit{bid},\ \mathit{bid}) \mid \texttt{return}\ (\mathit{exp}) \mid \ \ldots \\
\mathit{phi} &::= \Phi\ (\mathit{list}\ (\mathit{bid}, \mathit{exp})) \\
\mathit{block} &::= \{\mathit{entry} : \mathit{bid};\ \mathit{phis} : \mathit{list}\ (\mathit{id}, \mathit{phi}); \\
&\quad \mathit{code} : \mathit{list}\ (\mathit{id}, \mathit{instr});\ \mathit{term} : \mathit{term}\} \\
\mathit{cfg} &::= \{\mathit{name} : \mathit{id};\ \mathit{args} : \mathit{list}\ \mathit{bid};\ \mathit{entry} : \mathit{id};\ \mathit{body} : \mathit{list}\ \mathit{block}\} \\
\mathit{mcfg} &::= \texttt{mrec}\ (\mathit{cfg}, \ldots, \mathit{cfg})
\end{aligned}
$$

Figure 3.1: A minimal subset of VIR's syntax

has to do with the interaction between undefined values and the memory model: our implementation is sound, but prohibits the verification of some LLVM IR optimizations. See the discussions in Section 3.3.3 and 7.3 for more about these considerations.

For expository purposes, we restrict our presentation to a representative subset of VIR.

3.2.1 Syntax

VIR's syntax is shown on Figure 3.1. At the top-level, a VIR program is a mutually recursive *cfg* (*mcfg*) defined as a set of mutually recursive functions. Each function is a single control-flow-graph (*cfg*), which is a record that holds a name, formal variables binding its arguments, a block identifier as its entry point, and a list of blocks as its operational content.

Blocks are records holding an entry label, Φ−nodes, a list of instructions, and a terminator. The Φ−nodes are used to maintain SSA form [CFR+91], dynamically assigning different values to a variable depending on the identity of the predecessor block in the control flow. The *code* field contains a list of instructions (*instr*) paired with registers (*id*) destined to receive the value computed by the associated instruction. The code is set in a three-address-style format and intended to be executed sequentially after the Φ−nodes are set. The instructions we consider here are the evaluation of expressions, function calls, and memory operations such as allocation, loads, and stores. Finally, a *terminator* determines how the control flow should continue after a block. We include conditional branches and return statements as terminators.

We consider a subset of expressions (*exp*) supported by VIR: global (@i) and local (%i) identifiers, 64-bit integers, 1-bit integers, basic arithmetic operators (ranged over by op), and "get element pointer" (GEP) operations, used to access components in array-like data structures. As a consequence, VIR types τ include: `i64`, `i1`, arrays $[\tau]$, and pointers $\tau*$.

3.2.2 Dynamic Values

The semantics of VIR relies upon the domain of dynamic values that the language can manipulate. The core of these dynamic values are the so-called *defined values*.

$$dv \in \mathcal{V} ::= \texttt{none} \mid i \mid g \mid a \mid [\texttt{list}\ (\mathcal{V})] \mid \texttt{poison}$$

The void value, `none`, is a placeholder for operations with no meaningful return values. VIR supports 1, 8, 32 and 64 bit integers[9], but in this dissertation we only consider 64-bit integers (i) and 1-bit integers (g). Memory addresses (a) are given an abstract type `Addr` to allow for plugging memory models with different pointer representations into our semantics, a feature facilitated by the modularity of our semantics—Section 3.3.3 describes the implementation of our main memory model. VIR supports all of LLVM IR's structured values, but for simplicity we present only arrays, noted as [_].

Infamously, LLVM IR supports *poisoned values* (`poison`) representing a *deferred* undefined behavior [LKS+17]. Deferred UB is instrumental for aggressive optimizations, but a semantic subtlety. The `poison` value is a tainting mark: it propagates to all values that depend on it, so equations such as $\texttt{poison} + \texttt{poison} \equiv 2 * \texttt{poison} \equiv \texttt{poison}$ hold true. Although accounting for `poison` entails numerous semantic peculiarities, `poison` is modeled as its own defined value.

In contrast, the $\texttt{undef}_\tau$ value, a different model for deferred undefined behaviors supported by LLVM IR, admits a *set semantics*, representing all defined values of a given type τ. Operations that need to know the specific defined value at play behave non-deterministically over the set of values when acting upon `undef`. However, "reading" the same instance of an $\texttt{undef}_\tau$ value twice is not guaranteed to return the same value: $\texttt{undef}_{i64} + \texttt{undef}_{i64} \equiv \texttt{undef}_{i64}$ holds true, but $\texttt{undef}_{i64} + \texttt{undef}_{i64} \not\equiv 2 * \texttt{undef}_{i64}$ is an inequality, as the right hand side cannot be odd.

[9] We use CompCert's finite integers in our development.

To account for these peculiarities, we introduce *under-defined values* (uv):

$$uv \in \mathcal{V}_u ::= \uparrow \mathcal{V} \mid \texttt{undef}_\tau \mid \texttt{op}\ \mathcal{V}_u\ \mathcal{V}_u$$

Under-defined values are a superset of defined values—we write $\uparrow$ for the corresponding injection—but they also contain the special value $\texttt{undef}_\tau$ (we omit the subscript τ when the type is unimportant). Extending the semantics of arithmetic operations to a set interpretation of $\texttt{undef}_\tau$ would prevent us from interpreting two successive "reads" to an under-defined value differently. Instead, we can manipulate "symbolic" values built from any supported VIR arithmetic operator over $\mathcal{V}_u$.

3.3 A Modular LLVM IR Semantics

The toolbox provided by ITrees suggests a methodology for building denotational domains for a wide variety of programming languages. Given a syntax Lang, we proceed in three steps:

1. Identify the events $\mathcal{E}$ a program $p \in$ Lang may trigger;
2. By induction on Lang, use the ITree combinators to compute a representation of programs as elements of $\texttt{itree}\ \mathcal{E}\ A$, where A is an appropriate result type;
3. Define a handler for each family of events in $\mathcal{E}$ and use those to interpret the result of step 2.

The first step identifies the effects that programs in Lang may have, and abstracts them via a typed interface of events. The second step internalizes the control-flow and the potential divergence of Lang. The last step breathes life into the modular semantics, giving each event meaning, and completes the picture by combining these interpretations of effects.

This section applies this recipe to build our formal model of VIR. We inventory VIR's effects in Section 3.3.1 and derive from it the sets of events we manipulate. Section 3.3.2 describes how to represent each syntactic piece of VIR as an interaction tree, building up to the representation of *mcfg*s. Section 3.3.3 defines the concrete semantics of each category of effects through the definition of the handler for their corresponding events. Finally, Section 3.3.4 ties every component together and tackles the initialization of the memory to obtain the complete semantic model of VIR.

Global and local state

$$\mathcal{G} \triangleq \texttt{GRd}^{\mathcal{V}}(l) \mid \texttt{GWr}^{()}(l, v)$$
$$\mathcal{L} \triangleq \texttt{LRd}^{\mathcal{V}_u}(l) \mid \texttt{LWr}^{()}(l, v)$$
$$\mathcal{S}_{\mathcal{L}} \triangleq \texttt{LPush}^{()}(args) \mid \texttt{LPop}^{()}$$

Internal, external, and intrinsic calls

$$\mathcal{C} \triangleq \texttt{Call}^{\mathcal{V}_u}(a, uargs)$$
$$\mathcal{C}_{\mathcal{E}} \triangleq \texttt{Call}_{\texttt{E}}{}^{\mathcal{V}}(a, dargs)$$
$$\mathcal{I} \triangleq \texttt{Intrinsic}^{\mathcal{V}}(f, dargs)$$

Memory model interactions

$$\mathcal{M} \triangleq \texttt{MPush}^{()} \mid \texttt{MPop}^{()} \mid \texttt{Load}^{\mathcal{V}_u}(\tau, l) \mid \texttt{Store}^{()}(a, v) \mid \texttt{Alloca}^{\mathcal{V}}(\tau) \mid \texttt{GEP}^{\mathcal{V}}(\tau, v, vs) \mid \texttt{PtoI}^{\mathcal{V}}(a) \mid \texttt{ItoP}^{\mathcal{V}}(i)$$

Nondeterminism and UB

$$\mathcal{P} \triangleq \texttt{Pick}^{\mathcal{V}}(uv) \qquad \mathcal{U} \triangleq \texttt{UB}^{\emptyset}$$

Failure and debugging

$$\mathcal{F} \triangleq \texttt{Throw}^{\emptyset} \qquad \mathcal{D} \triangleq \texttt{Debug}^{()}(msg)$$

Figure 3.2: VIR events. (Superscripts indicate return types.)

3.3.1 An Inventory of LLVM's Events

Figure 3.2 depicts the eleven categories of events that can be triggered by a VIR program. At this point we specify the *types* of the events, which constrain the types of the handlers that will concretely implement their semantics.

Global state and *local state* events, $\mathcal{G}$ and $\mathcal{L}$ respectively, describe reads and writes to the global and local environments. The global environment is a read-only map that sends global identifiers to their corresponding memory addresses, and is written to only at its initialization. In contrast, the local environment represents stack frames for function calls, and is mutated throughout execution.

Local stack events, $\mathcal{S}_{\mathcal{L}}$, provide a fresh local environment for each function call. The $\texttt{LPush}^{()}$ event pushes a fresh local environment initialized with an association list of variables to $\mathcal{V}_u$s, the arguments passed to the function. The $\texttt{LPop}^{()}$ event pops the stack frame when a function returns. Separating $\mathcal{L}$ and $\mathcal{S}_{\mathcal{L}}$ into two distinct domains of events allows for the denotation of functions to be oblivious to the existence of this stack of states, as will become apparent in Section 3.3.3.

Memory events, $\mathcal{M}$, are richer. A program can $\texttt{MPush}^{()}$ or $\texttt{MPop}^{()}$ a (memory) frame within which new storage can be dynamically allocated via the $\texttt{Alloca}^{\mathcal{V}}(\tau)$ event. Memory cells can be accessed via $\texttt{Store}^{()}(a, dv)$ and $\texttt{Load}^{\mathcal{V}_u}(\tau, l)$. Note that our model stores *defined* values in memory, but loads may return *undefined* ones (e.g. if an allocated, but uninitialized cell is read). $\texttt{GEP}^{\mathcal{V}}(\tau, dv, dvs)$ computes a pointer within an aggregate structure. Finally, pointer–integer casts, $\texttt{PtoI}^{\mathcal{V}}(a)$, and, reciprocally, $\texttt{ItoP}^{\mathcal{V}}(i)$, are supported.

VIR supports internal calls, external calls, and calls to "intrinsics." Internal calls, C, should be the result of the denotation of the corresponding function: it can therefore return any $\mathcal{V}_u$. External calls, $C_{\mathcal{E}}$, are not resolved internally—they model invocations of OS or library code—and can be implemented by any external means: they only process and return defined values in $\mathcal{V}$. Intrinsics are LLVM's mechanism for lightweight language extensions: their names and semantics are standardized, but their addresses cannot be taken. VIR's semantics is parameterized by an extensible set of supported intrinsics modeled by events of type $\mathcal{I}$.

LLVM IR is a non-deterministic language. The VIR semantics implements the *undefined value* $\texttt{undef}_\tau$ (recall Section 3.2.2), by manipulating the symbolic under-defined values, $\mathcal{V}_u$, as long as possible. When the computation nonetheless reaches a point requiring a uniquely determined $\mathcal{V}$, an oracle, modeled by $\texttt{Pick}^{\mathcal{V}}(uv) \in \mathcal{P}$ events, is invoked to choose a defined value.

A second source of non-determinism comes from *undefined behaviors*, which represent exceptional circumstances. If execution leads to undefined behavior, the LLVM semantics says that *any* behavior may substitute for this execution. [10] Semantically, this means that we need an event to which we can give any meaning; this polymorphism is achieved through an event, $\texttt{UB}^{\emptyset} \in \mathcal{U}$, whose returned type is void. We write `raiseUB` for the polymorphic triggering of $\texttt{UB}^{\emptyset}$.

Finally, $\texttt{Throw}^{\emptyset} \in \mathcal{F}$ and $\texttt{Debug}^{()}(m) \in \mathcal{D}$ respectively express dynamic errors and dynamic debug messages. We write `fail` for the polymorphic triggering of $\texttt{Throw}^{\emptyset}$.

3.3.2 Representing VIR Programs as Interaction Trees

The second step of denotation consists of representing the *syntax* of VIR as an ITree acting over an interface built from the previously described events. More specifically, let us define the top-level interface for LLVM programs:

$$\texttt{virE} \triangleq C \oplus \mathcal{I} \oplus \mathcal{G} \oplus (\mathcal{S}_{\mathcal{L}} \oplus \mathcal{L}) \oplus \mathcal{M} \oplus \mathcal{P} \oplus \mathcal{U} \oplus \mathcal{D} \oplus \mathcal{F}$$

The main purpose of this section is hence to define a function

$$[\![p]\!]_{\text{mcfg}}\ (\tau : dtyp)\ (f : \mathcal{V})\ (args : \texttt{list}\ (\mathcal{V}))\ :\ \texttt{itree}\ \texttt{virE}\ \mathcal{V}_u$$

[10] The semantics may interpret an undefined behavior as any computation, but may not alter the past.

$$
\begin{array}{rl}
(\uparrow \texttt{poison}) \oplus \_ & = \uparrow \texttt{poison} \\
\_ \oplus (\uparrow \texttt{poison}) & = \uparrow \texttt{poison} \\
(\uparrow dv_1) \oplus (\uparrow dv_2) & = \uparrow (dv_1 +_{i64} dv_2) \\
uv_1 \oplus uv_2 & = uv_1 +_{i64} uv_2
\end{array}
\qquad\qquad
\begin{array}{rl}
uv_1 \oslash dv_2 & = \texttt{ret}\ (uv_1 /_{i64} (\uparrow dv_2)) \\
(\uparrow \texttt{poison}) \oslash \_ & = \texttt{ret}\ (\uparrow \texttt{poison}) \\
\_ \oslash \texttt{poison} & = \texttt{raiseUB} \\
(\uparrow dv_1) \oslash dv_2 & = \texttt{if}\ dv_2 =_{i64} 0 \\
& \quad \texttt{then raiseUB else ret}\ \uparrow (dv_1 /_{i64} dv_2)
\end{array}
$$

Figure 3.3: Binary operations on under-defined values

which, given a *mcfg* p, a return type τ, the address of the starting function f, and a list of arguments arg, internalizes the semantics into a single ITree over the `virE` interface.

The definition of $[\![\_]\!]_{\text{mcfg}}$ directly follows the structure of the syntax. In particular, our approach allows us to easily define the meaning of each syntactic sub-component in complete autonomy, which is a key feature to enable compositional reasoning about the resulting semantics.

Expressions Expressions are naturally represented as ITrees that return values $u \in \mathcal{V}_u$. The representation function, defined inductively over the syntax, is given by:

$$
\begin{aligned}
[\![\%i]\!]_e &= \texttt{trigger}\ (\texttt{LRd}^{\mathcal{V}_u}(i)) \\
[\![@i]\!]_e &= dv \leftarrow \texttt{trigger}\ (\texttt{GRd}^{\mathcal{V}}(i))\ ;;\ \texttt{ret}\ (\uparrow dv) \\
[\![e_1 + e_2]\!]_e &= uv_1 \leftarrow [\![e_1]\!]_e\ ;;\ uv_2 \leftarrow [\![e_2]\!]_e\ ;;\ \texttt{ret}\ (uv_1 \oplus uv_2) \\
[\![e_1 / e_2]\!]_e &= uv_1 \leftarrow [\![e_1]\!]_e\ ;;\ uv_2 \leftarrow [\![e_2]\!]_e\ ;; \\
&\qquad dv \leftarrow \texttt{concretize\_or\_pick}(uv_2)\ ;;\ uv_1 \oslash dv
\end{aligned}
$$

The meaning of a local variable $\%i$ is a computation with the effect of accessing the local environment to retrieve the value associated to i. Thus, at this stage, it is represented by triggering the $\texttt{LRd}^{\mathcal{V}_u}(i)$ event, whose return type is precisely $\mathcal{V}_u$: once interpreted, this small interaction tree will return a value of the correct type. A global variable $@i$ has a similar representation: it triggers the corresponding $\texttt{GRd}^{\mathcal{V}}(i)$ event, whose return type is statically guaranteed to contain defined values. We bind the triggered result to $dv \in \mathcal{V}$ and inject this bound value into the domain of under-defined values.

Binary operations, like the addition of integers, are represented by taking the ITree representation of each subexpression e_1 and e_2, binding the results of these computations to $uv_1, uv_2 \in \mathcal{V}_u$ respectively, and then performing the basic operation on uv_1 and uv_2 and returning the result. Division, however, is more complex because division by 0 is undefined behavior. If the denominator is an under-defined value, we will need to *pick* a valid concretization, $dv \in \mathcal{V}$. We use `concretize_or_pick` for this purpose, which either injects the denominator into $\mathcal{V}$ if it is already concrete, or triggers a $\texttt{Pick}^{\mathcal{V}}(uv_2)$ event that acts as an oracle for concretizing $\mathcal{V}_u$ values. Note that the basic operations must account for poison and trigger

undefined behavior via `raiseUB` when division by 0 occurs, as seen in Figure 3.3.

Instructions LLVM instructions are represented by a pair (id, ins) of a side-effectful instruction *ins* and an identifier *id* destined to receive the result of the operation. Their representation function builds upon $[\![\_]\!]_e$, as defined in Figure 3.4.

Representing an operation (id, e) reduces to calling $[\![e]\!]_e$ and binding its result with the trigger of the local write $\texttt{LWr}^{()}(id, uv)$. Memory operations require extra care. Consider `load` $(\tau,\ e)$, that reads from an address expression e of type τ. The address ua resulting from $[\![e]\!]_e$ should be used to trigger the appropriate memory event. However the memory model can be indexed only by defined memory addresses, and stores defined values. We therefore resolve any under-definedness in ua by *pick*ing a valid concretization, $da \in \mathcal{V}$, of the under-defined value. After getting the concrete address, we need to take care of one last subtlety: defined values can be poisoned, and attempting to load from such an address is an undefined behavior. This can be handled with a simple case analysis on the $\mathcal{V}$, which raises a $\mathcal{U}$ event if the $\mathcal{V}$ is poison. Stores and allocations follow a similar pattern.

We next turn to call instructions. The distinction between internal and external calls is a property of the ambient *mcfg*, and is not relevant to individual *cfg*s. They are hence both represented as a $\texttt{Call}^{\mathcal{V}_u}(\_, \_)$ event at the level of instructions, and will be distinguished at the level of *mcfg*s, as described at the end of this section. In contrast, the list of supported intrinsics is a parameter of our semantics; they can always be resolved statically. Hence, a `call` $(f,\ args)$ instruction is represented by first sequentially interpreting the list of arguments (*args*) using a monadic map, `mapm` . If the function is an intrinsic, arguments are concretized to defined values and passed to the dedicated $\mathcal{I}$ event. Otherwise, the address of the function is retrieved from its name and passed to a call event. In both cases, the resulting value is bound to the associated local variable *id*, as usual.

Denoting straight line code, $[\![\_]\!]_c$, simply sequences the denotation of its instructions using `mapm` .

Terminators Terminators either return the identity of the next block to be evaluated, or signal the end of the current function call by returning a value. This dichotomy is reflected in the ITree's return type, a disjoint sum of block identifiers and under-defined values (see below). The representation is otherwise as expected: `return` (e) evaluates e and returns its right injection. A `branch` (e, b_l, b_r) evaluates e and performs a case analysis on its result. In the first case, the result is a 1-bit integer, and the value is treated as a boolean

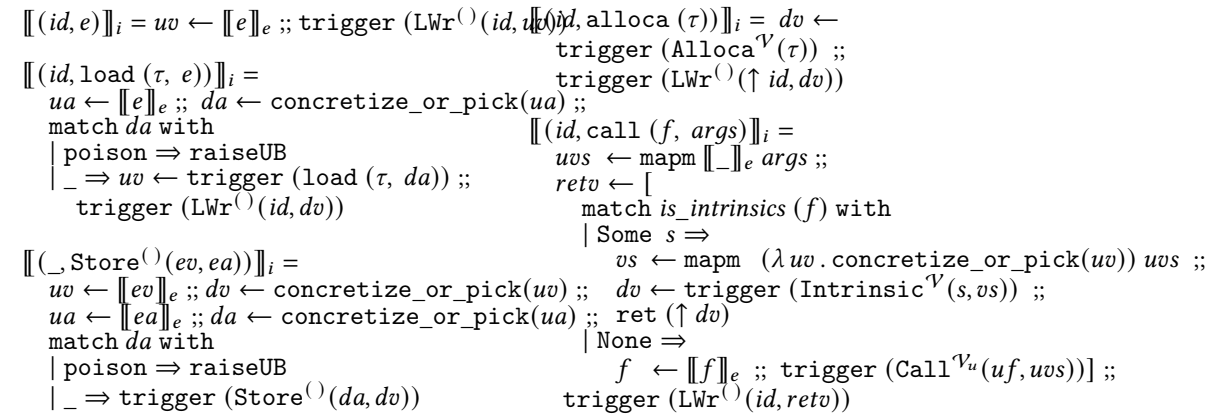

Figure 3.4: Denoting instructions as ITrees

to decide which branch to take and thus a block identifier is returned. Branching on a poisoned value is considered an undefined behavior, so a `raiseUB` is returned. All other cases are considered erroneous.

```
⟦return (e)⟧_t          =  uv ← ⟦e⟧_e ;; ret (inr uv)
⟦branch (e, b_l, b_r)⟧_t =
  uv ← ⟦e⟧_e ;; dv ← concretize_or_pick(ua) ;;
  match dv with
   | g       ⇒ if g =_1 1 then ret (inl b_l) else ret (inl b_r)
   | poison  ⇒ raiseUB
   | _       ⇒ fail
```

Control-flow graphs We next consider the representation of VIR functions, *i.e.*, of *cfg*s. More generally, we want to be able to denote *open* functions—a subgraph of mutually referential, labeled control-flow-graph blocks that might refer to block labels not in the subgraph—in order to reason compositionally about them. Therefore, we define the representation of a list of blocks: $[\![bks]\!]_{\text{bks}}\ (b_f, b_s)$ as a function that takes as an argument the label b_s of the source block at which to start the computation, as well as the label b_f of the block visited last. This function loops, using the `iter` operator (see Chapter 2) combinator to resolve the control flow of the mutual references among the blocks, until it either finds a return statement, or computes the label of a block that does not belong to the sub-control flow graph.

$$
\begin{aligned}
[\![bks]\!]_{\text{bks}} &= \texttt{iter}\ body \\
body\ (b_f, b_s) &= \texttt{try}\ bk_s \hookleftarrow bks\ [b_s]\ \texttt{with ret (inr (inl } (b_f, b_s))) \texttt{ in} \\
&\quad res \leftarrow ([\![bk_s.(\Phi)]\!]_{\Phi s}^{b_f} \ ;; \ [\![bk_s.(c)]\!]_c \ ;; \ [\![bk_s.(t)]\!]_t) \\
&\quad \texttt{match } res \texttt{ with} \\
&\quad \mid \texttt{inr } dv \Rightarrow \texttt{ret (inr (inr } dv)) \\
&\quad \mid \texttt{inl } b_t \Rightarrow \texttt{ret (inl } (b_s, b_t))
\end{aligned}
$$

Above, we write $\texttt{try}\ x \hookleftarrow mv\ \texttt{with}\ t\ \texttt{in}\ k$ to bind the content of an option value, mv to x in k if it is a `Some` constructor, and return t otherwise. When the partiality is simply internalized in the tree, we also abbreviate $x \hookleftarrow mv\ \texttt{in}\ k$ for $\texttt{try}\ x \hookleftarrow mv\ \texttt{with fail in}\ k$.

One wrinkle is that we need to account for Φ−nodes, which assign to local variables based on the label of the previously visited block. Additionally, all Φ−nodes need to be executed "in parallel", due to cycles in the control flow graph allowing for the right-hand side expressions to depend on the (previous) values of variables being assigned. Thus, given the label bid_f of the previously visited block, we can represent the computation returning the value to be bound at a given Φ−node:

$$
[\![(id, \Phi(args))]\!]_{\Phi}^{bid_f} = op \hookleftarrow args[bid_f]\ \texttt{in}\ uv \leftarrow [\![op]\!]_e \ ;; \ \texttt{ret}\ (id, uv)
$$

A list of Φ−nodes then retrieves the association list of identifiers to under-defined values, before performing the writes.

$$
\begin{aligned}
[\![\Phi s]\!]_{\Phi s}^{bid_f} = \quad & dvs \leftarrow \texttt{mapm}\ ([\![\_]\!]_{\Phi}^{bid_f})\ \Phi s \ ;; \\
& \texttt{mapm}\ (\lambda\ (id, dv)\ .\ \texttt{trigger}\ (\texttt{LWr}^{()}(id, dv)))\ dvs
\end{aligned}
$$

$$
\begin{aligned}
&[\![cfg]\!]_{\text{cfg}} = \\
&\quad r \leftarrow [\![cfg.(body)]\!]_{\text{bks}}\ (\cdot, cfg.(entry)) \ ;; \\
&\quad \texttt{match } r \texttt{ with} \\
&\quad \mid \texttt{inr } uv \ \Rightarrow \texttt{ret } uv \\
&\quad \mid \texttt{inl } bid \ \Rightarrow \texttt{fail}
\end{aligned}
$$

Defining the representation of a *closed cfg* (right) is simply a matter of representing its blocks and interpreting a final label as an error (an invalid jump).[11]

[11] Note that it is safe to provide a "dummy" origin block as LLVM IR explicitely prohibits entry blocks of functions to contain Φ−nodes.

Mutually Recursive Control-Flow Graphs

$$
\begin{aligned}
&[\![\, mcfg \,]\!]_{\text{mcfg}}\ fundefs\ f\ args = \texttt{mrec}\ body\ (\texttt{Call}^{\mathcal{V}_u}(f, args)) \\
&body\ (\texttt{Call}^{\mathcal{V}_u}(uf, args)) = \\
&\quad df \leftarrow \texttt{concretize\_or\_pick}(uf)\ ;; \\
&\quad \texttt{match}\ fundefs\ [df]\ \texttt{with} \\
&\quad \mid \texttt{Some}\ f\_den \Rightarrow f\_den\ (args) \\
&\quad \mid \texttt{None} \Rightarrow dargs \leftarrow \texttt{mapm}\ (\lambda v\,.\,\texttt{concretize\_or\_pick}(v))\ args\ ;; \\
&\qquad \texttt{trigger}\ (\texttt{Call}_{\texttt{E}}{}^{\mathcal{V}}(uf, dargs))
\end{aligned}
$$

Lastly, we represent *mcfg*s, i.e. sets of mutually recursive *cfg*s. The main task is tying the recursive knot of function calls, similar to the *cfg* blocks. However, the `iter` combinator falls short this time: calls are not necessarily tail recursive. We therefore rely on a more general `mrec` combinator, to tie the knot for us by dynamically unrolling function calls. Conveniently, LLVM IR is a first order language: all (internal) functions are defined at the top-level, as part of the *mcfg*. We can therefore statically know their global identifiers, and build an association list[12] of type $fundefs : \texttt{list}\ (\mathcal{V} * (\texttt{list}\ (\mathcal{V}_u) \rightarrow \texttt{itree}\ \texttt{virE}\ \mathcal{V}_u))$ mapping each function address to its ITree representation. As shown below, the body passed to `mrec` can therefore simply query this list to know if the function being called is internal, in which case it returns its representation. Otherwise, it triggers back the call, this time explicitly classified as external.

3.3.3 Handling Events

Section 3.3.2 introduced a compositional representation of VIR in terms of ITrees. The effects captured by the events contained in these trees do not have a presupposed implementation: we now define their meaning in a modular way through independent *handlers*.

As shown in Figure 3.5, the full VIR semantic model is given by a "tower of interpreters" which interpret events to different levels. Level 0 corresponds to the uninterpreted ITree. Each subsequent level handles some events using an appropriate instance of `interp`. For example, the interpreter from Level 0 to Level 1 handles intrinsic events only, whereas by Level 2 both intrinsic events and global events have been handled. We want to be able to establish that a program p_1 refines a program p_2 in the simplest monad allowing the refinement to be established.

[12] Constructing this list happens when initializing the global, top-level state. See Section 3.3.4.

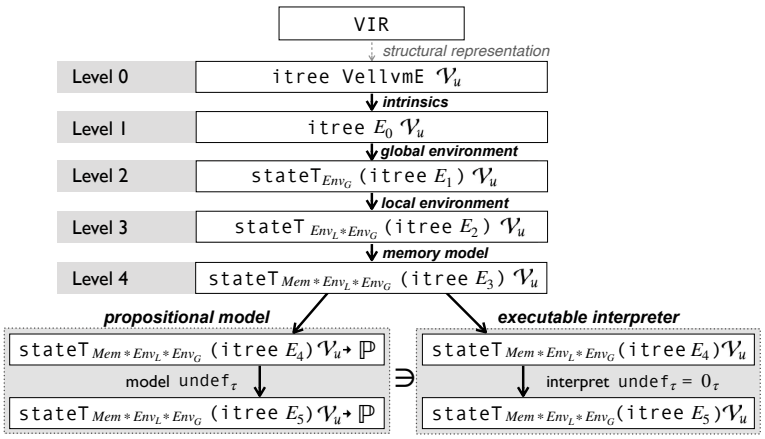

Figure 3.5: Levels of interpretation

A second major benefit of using handlers is the ability to use *different* handlers for the *same* events. This "plug-and-play" aspect makes it easier to experiment with semantic features, such as alternate memory models. We also make crucial use of this feature to define both the full VIR *semantic model* (the left path through Figure 3.5) and an *executable* VIR *interpreter* (the right path). As explained below, the model accounts for nondeterminism in the VIR semantics by interpreting some events *propositionally* (*i.e.*, into sets characterized by Coq predicates), making them suitable for specification but not extraction, whereas the executable interpreter concretizes the nondeterminism, which is useful for testing and debugging. The two semantics share most of the interpretation levels, allowing us to easily prove that the implementation refines the model (see Section 4.1).

The following subsections discuss the successive handlers for VIR's events. Most of them target state monads, of which the memory model is the most complex. The handlers for pick events $\mathcal{P}$ and undefined behaviors $\mathcal{U}$ target the $\texttt{propT}_E$ monad of "propositional sets of computations."

$\mathcal{I}$: Intrinsics VIR, like LLVM, supports *intrinsic functions* that extend its core semantics (for instance to allow for the implementation of new "primitive" arithmetic operations). Such intrinsics are defined by a map associating each name to a semantic function of type $\texttt{list}\ (\mathcal{V}) \to \mathcal{V} + \texttt{err}$, *i.e.*, a pure Coq function that takes a list of $\mathcal{V}$s and produces either an error or a $\mathcal{V}$ as a result. The handler for intrinsics looks up the name, and runs the semantic function on the arguments, returning the result (or raising an error if it

```
handle_I Intrinsic^V(f_name, args) : itree E_0 V =
  match is_intrinsics (f_name) with
  | Some f ⇒ v ⟷ f args in ret v
  | None   ⇒ trigger (Intrinsic^V(f_name, args))

handle_G e env : stateT_EnvG (itree E_1) _ =
  λ env. (match e with
  | GWr^()(l, v) ⇒ ret (Map.add l v env, tt)
  | GRd^V(l)     ⇒ v ⟷ Map.lookup l env in ret (env, v))
```

```
handle_S_L e : stateT_Frame * Stack (itree E_3) _ =
  λ (env, stack).
    (match e with
    | LPush^()(args) ⇒
      ret (foldr (λ (x, dv) . (Map.add x dv))
        Map.empty args, env :: stack), tt)
    | LPop^() ⇒
      match stack with
      | [] ⇒ fail
      | env' :: stack' ⇒ ret ((env', stack'), tt))
```

Figure 3.6: Handlers for Interpretation Levels

fails).[13]

$\mathcal{G}$: Globals Global variables in VIR are given by a state monad that acts on a map *env* of type *EnvG* from identifiers to pointers. Handling globals simply involves converting $\texttt{GRd}^{\mathcal{V}}(k)$ and $\texttt{GWr}^{()}(k, v,)$ events into lookups and insertions into this map, respectively. The map *env* is constructed at initialization time and is constant thereafter.

$\mathcal{L}$: Locals Local variables are handled analogously to globals, $\mathcal{L}$ events being implemented w.r.t. a map of type *EnvL*. The scope of local variables will be handled by $\mathcal{S}_{\mathcal{L}}$ events.

$\mathcal{S}_{\mathcal{L}}$: Stack $\mathcal{S}_{\mathcal{L}}$ stack events are triggered when calling a function and returning from a function. These $\mathcal{S}_{\mathcal{L}}$ events, $\texttt{LPush}^{()}(as)$ and $\texttt{LPop}^{()}$, set up the local environment containing the functions arguments and pop this environment on function return, respectively. Local variables from an enclosing scope in VIR are not accessible within the current scope, and so this stack of environments can simply be a list of unrelated mappings from identifiers to values.

$\mathcal{M}$: Memory The handler for VIR's memory events is far more complex than the handlers described above. The VIR implementation is closest to the *quasi-concrete* model proposed by Kang, *et al.* [KHM+15a]. Briefly, the quasi-concrete model has a "logical" memory, represented by an integer map to blocks, where each block is an integer map to *symbolic bytes* that contain actual bytes or representation information, including the possibility of undefinedness. Logical addresses are represented as a pair of integers; the first being the index in the map of blocks, and the second representing the offset of the first byte of the value within the block. $\mathcal{M}$ events are handled by interpreting them into a state monad containing this map of

[13] If the intrinsic function isn't handled here, the event is re-triggered, allowing downstream interpreters to handle it. For instance the memory handler handles the `memcpy` intrinsics.

logical blocks, as well as a list of stack frames.

$\texttt{Alloca}^{\mathcal{V}}(\tau)$ allocates a new empty block with a size matching τ to the current stack frame. $\texttt{Store}^{()}(a, v)$ serializes v into symbolic bytes, storing them at address a in memory, and triggering failure if a is not allocated. $\texttt{Load}^{\mathcal{V}_u}(\tau, a)$ deserializes the symbolic bytes stored at a in memory, also failing on unallocated addresses. The $\texttt{GEP}^{\mathcal{V}}(\tau, dv, vs)$ event implements LLVM's `getelementptr` instruction, which is used for indexing into aggregate data structures, where τ is the type of the structure, dv is the base address of the structure, and vs is a list of indices. The final two $\mathcal{M}$ events are $\texttt{PtoI}^{\mathcal{V}}(a)$ and $\texttt{ItoP}^{\mathcal{V}}(a)$, which represent pointer-to-integer and integer-to-pointer casts respectively. To properly handle these casts the model also contains a "concrete" memory, giving concretized blocks (i.e., blocks referenced by a pointer has been cast to an integer) a concrete address that can be converted to an integer. Pointer values remain "logical" until they participate in a cast instruction.

This memory model, though sound, is a source of misalignment between our semantics and LLVM IR's semantics. Indeed, as described previously, we have taken care of introducing under-defined values in order to make sure that successive reads to an instance of `undef` could lead to different results: it behaves as a random variable. However, this memory model is only able to store a defined value: it collapses the non-determinism via the $\texttt{Pick}^{\mathcal{V}}(\_)$ event when interacting with the memory. This behavior prevents proving the correctness of certain optimizations, such as store forwarding. Other proposed memory models, such as the "twin allocation semantics" by Lee *et al.* [LHJ+18b] permit store forwarding, but prohibit other desirable optimizations (such as dead allocation elimination). It remains an open research question how best to fully model the LLVM's complex memory semantics, but the modularity of our handlers should make it easier to adapt VIR as the technology improves.

$\mathcal{P}$: Pick When implementing the handlers for a $\texttt{Pick}^{\mathcal{V}}(u)$ event, which resolve nondeterminism, there is a bifurcation: The "true" semantic model, which aims to capture *all* the legal behaviors, uses a handler that interprets behaviors into a monad $\texttt{propT}_E A \triangleq \texttt{itree}\ E\ A \to \mathbb{P}$. This monad represents *sets* of ITrees as Coq predicates, allowing us to use logical quantifiers to express the allowable nondeterministic behaviors. On the other hand, for *executable* versions of the VIR semantics, we can use any handler that implements *one* of the allowable behaviors, but provides a way to run VIR programs. We will see in Section 4.1 that we can prove that a (good) executable interpreter refines the model. Here, we just define the handlers themselves.

The $\mathcal{P}$ handler for the semantic model is shown below:

$$\texttt{model\_handle}_{\mathcal{P}}\ \texttt{Pick}^{\mathcal{V}}(u) : \texttt{propT}_{E_5}\mathcal{V} \ = \ \{t \mid \begin{cases} t \approx \texttt{fail} & [\![u]\!]_C = \emptyset \\ t \approx \texttt{ret}\ v & dv \in [\![u]\!]_C \ \wedge \ dv \neq \texttt{poison} \\ t \approx \texttt{raiseUB} & dv \in [\![u]\!]_C \ \wedge \ dv = \texttt{poison} \end{cases} \}$$

Here, "$\approx$" stands for ITree equivalence. The set $[\![u]\!]_C$ denotes all possible defined values corresponding to u. For example, we have $[\![2/\texttt{undef}_{\texttt{i64}}]\!]_C = \{2, 1, 0, \texttt{poison}\}$ because $2/2 = 1$, $2/1 = 2$, $2/0 = \texttt{poison}$, and $2/\texttt{n} = 0$ for all other (unsigned) n. Thus, handling $\texttt{Pick}^{\mathcal{V}}(2/\texttt{undef}_{\texttt{i64}})$ might trigger undefined behavior or it might yield 0, 1, or 2, nondeterministically. If there are no concretizations of u, the semantics fails.

Many executable implementations are allowed by this model—they work by "picking" a default value (generally the equivalent of 0 for the given type) for each instance of $\texttt{undef}_\tau$ in the under-defined expression u and then evaluating the expression to obtain a defined value.

$$\texttt{exec\_handle}_{\mathcal{P}}\ \texttt{Pick}^{\mathcal{V}}(u) : \texttt{propT}_{E_5}\mathcal{V} \ = \ \texttt{ret default(u)}$$

$\mathcal{U}$: Undefined Behavior Vellvm represents undefined behavior through $\mathcal{U}$ events. A $\mathcal{U}$ event $\texttt{UB}^{\emptyset}$ is triggered whenever undefined behavior is encountered, either directly from the interpretation of the program, as in the case of a store to poison, or less directly through under-defined values and $\mathcal{P}$ events, such as a division by undef as described above. As with $\mathcal{P}$, there are both propositional and executable handlers.

The propT handler is trivial: it permits the set of *all* ITrees of the appropriate type:

$$\texttt{model\_handle}_{\mathcal{U}}\ \texttt{UB}^{\emptyset} : \texttt{propT}_{E_5}\mathcal{V} \ = \ \{t \mid t : \texttt{itree}\ E_5\ \mathcal{V}\}$$

An executable semantics is free to do anything at all upon encountering undefined behavior. To aid with debugging, our executable semantics simply fails:

$$\texttt{exec\_handle}_{\mathcal{U}}\ \texttt{UB}^{\emptyset} : \texttt{propT}_{E_5}\mathcal{V} \ = \ \texttt{fail}$$

3.3.4 Stitching the Semantics Together

Having represented our syntax as ITrees, and having defined handlers for each event type, we combine them with interp (see Chapter 2) to obtain intepreters over complete ITrees as depicted in Figure 3.5. The

order in which we compose these interpreters is chosen to keep "simpler" semantics (such as the pure intrinsics) earlier and delay as far down the chain as possible the introduction of the `prop` monad.

At the top-level, an LLVM program is parsed into a VIR representation containing the declarations of globals[14], the *mcfg*, and the name of the main from which to start the execution. The set of internal functions is fixed and known statically, which allows us to build the association list of function addresses to denotations required by $[\![\_]\!]_{\text{mcfg}}$:

$$
\begin{aligned}
&[\![prog]\!]_{\text{VIR}}\ main\ args\ mcfg = \\
&\quad genv \leftarrow \text{build\_global\_env}\ (prog)\ ;; \\
&\quad defns \leftarrow \texttt{mapm}\ (\ \lambda\ cfg\ .\ fv \leftarrow \texttt{trigger}\ (\texttt{GRd}^{\mathcal{V}}(cfg.(entry)))\ ;; \\
&\qquad\qquad\qquad \texttt{ret}\ (fv,\ [\![cfg]\!]_{\text{cfg}}))\ prog\ ;; \\
&\quad addr \leftarrow \texttt{trigger}\ (\texttt{GRd}^{\mathcal{V}}(main))\ ;; \\
&\quad [\![prog]\!]_{\text{mcfg}}\ defns\ (\uparrow addr)\ args
\end{aligned}
$$

Finally, we obtain the full semantic model for VIR, `model`, as $\texttt{interp\_vir}([\![\_]\!]_{\text{VIR}})$. If, rather than composing all the layers of intepretation, we instead define $\texttt{interp\_vir}_4$, stopping at the fourth level, we obtain a semantics that does not introduce the `prop` monad—we return to this idea in Section 4.1. Finally, we can also interpret all stages, but using different handlers: the left path on Figure 3.5 defines the *propositional model*, where the right path leads to an *executable interpreter* for VIR that we refer to as `interpreter`.

[14]We elide the details of the initialization of the global environment, keeping build_global_env opaque.

Chapter 4

VIR Metatheory

4.1 VIR Equivalences

One of Vellvm's primary goals is to serve as a formal semantics suitable for *reasoning about* LLVM IR code, for verifying optimization passes or the correctness of translations to/from it. We hence require a notion of what it means for an optimization to be correct: we need a *refinement relation* between LLVM programs. Due to the nondeterminism present in LLVM (e.g. for `undef` values and undefined behaviors), a single program fragment p may have a *set* of valid behaviors $[\![p]\!]$,and any p' such that $[\![p]\!] \supseteq [\![p']\!]$ is a valid *refinement* of p.

In this section, we define appropriate notions of refinement and prove that we can lift refinements at the ITree level to set inclusions at the propositional level. We also establish some powerful general-purpose machinery for working with these refinements, obtaining the correctness of VIR's executable interpreter with respect to the nondeterministic model as an easy corollary of the correctness of handlers for pick and undefined behaviors. The refinement theory is crucial for reasoning about VIR programs—by lifting the structural equational theory to VIR constructs, we obtain powerful relational reasoning principles suitable to prove correct program transformations and compilers targeting VIR in a compositional fashion.

4.1.1 ITree Equivalences and Refinement Relations

At the heart of the refinement relations for ITrees is the $t_1 \approx_R t_2$, or `eutt` relation, also known as "equivalence up to taus." Here $t_1 \approx_R t_2$ relates t_1 with t_2 if these itrees are weakly bisimilar (i.e. they produce the same tree of visible events, ignoring any finite number of `Tau`s) where all values returned along corresponding branches are related by R. We omit the definition of $\approx_R$ (see [XZH+20, ZHHZ20] for details), instead focusing on its relevant properties. Technically, $\approx_R$ is an equivalence relation only when R is; the usual notion of weak bisimulation is recovered as the instance $\approx_{\texttt{eq}}$, where the relation is chosen to be Coq's Leibnitz equality, `eq`, and we leave off the subscript in this case. The $\approx$ relation plays a particular role in

General Monad Laws

$$x \leftarrow \texttt{ret}\ v;;\ k\ x \approx k\ v$$
$$x \leftarrow t;;\ \texttt{ret}\ x \approx t$$
$$x \leftarrow (y \leftarrow s;;\ t\ y);;\ u\ x \approx (y \leftarrow s;;\ x \leftarrow t\ y;;\ u\ x)$$

General Interpreter Laws

$$\texttt{interp}\ h\ (\texttt{trigger}\ e) \approx h\ \_\ e$$
$$\texttt{interp}\ h\ (\texttt{ret}\ r) \approx \texttt{ret}\ r$$
$$\texttt{interp}\ h\ (x \leftarrow t;;\ k\ x) \approx x \leftarrow \texttt{interp}\ h\ t;;\ \texttt{interp}\ h\ (k\ x)$$

ITree-specific Structural Laws

$$\texttt{Tau}\ t \approx t$$
$$x \leftarrow \texttt{Tau}\ t;;\ k\ x \approx \texttt{Tau}\ (x \leftarrow t;;\ k\ x)$$
$$x \leftarrow \texttt{Vis}\ e\ k_1;;\ k_2\ x \approx \texttt{Vis}\ e\ (\lambda\ y.\ (x \leftarrow k_1\ y;;\ k_2\ x))$$

Figure 4.1: Core equational theory of ITrees.

that it can be used as a rewriting rule in any $\approx_R$ goal. When R is a preorder (i.e. reflexive and transitive), so is $\approx_R$, and we can think of this relation as a form of tree refinement; in this case we write $t_1 \gtrsim_R t_2$ to emphasize the (potential) asymmetry and think of t_2 as refining t_1.

The ITrees equational theory is defined in terms of $\approx$. Figure 4.1 shows the key equivalences that allow us to exploit the monadic structure and semantics of interpretations. The general interpreter laws hold for any monad that supports a suitable implementation of the `iter` combinator, which includes ITrees and many monads built from them—especially important for the VIR semantics are the state and propositional monad transformers.[15] The figure also shows laws specific to ITrees, which explain how `Tau` and `Vis` interact with `bind`. The first of these laws, $(\texttt{Tau}\ t) \approx t$, lets us ignore any (finite number) of `Tau`'s, which is where $\approx$ gets the name "equivalence up to taus" from.

Figure 4.2 shows (selected) *relational* reasoning principles that hold for $\approx_R$, for an arbitrary relation R. In the case of refinements, the ERet rule establishes the basic relation between values returned by the computation, and reflexivity of R ensures that the computation refines itself. In the ETrans rule, we write $R_1 \circ R_2$ for relation composition. For refinements, we have $R \circ R = R$ by transitivity, so indeed tree refinement is also transitive. Moreover, ETrans implies that rewriting with the monad and interpretation laws is sound for refinement: since $\texttt{eq} \circ R = R = R \circ \texttt{eq}$ for any relation R. This means that we can string refinements and equivalences together to reach a desired conclusion. For instance, from $t_1 \approx t_2 \gtrsim_R t_3 \gtrsim_R t_4 \approx t_5$ we can conclude $t_1 \gtrsim_R t_5$.

Rule EMon says that monotonicity allows us to prove a stronger refinement relation to establish a weaker one, and EInterp says that interpretation with respect to the same handler preserves any refinement relation

[15] The Coq code uses typeclasses to characterize such monads and to overload $\approx_R$ with suitable notions of refinement.

$$\frac{R(r_1, r_2)}{\texttt{ret}\ r_1 \approx_R \texttt{ret}\ r_2}\ \textsc{ERet} \qquad \frac{t_1 \approx_{R_1} t_2 \quad t_2 \approx_{R_2} t_3}{t_1 \approx_{R_1 \circ R_2} t_3}\ \textsc{ETrans}$$

$$\frac{t_1 \approx_U t_2 \qquad \forall\, u_1, u_2, U(u_1, u_2) \Rightarrow (k_1\ u_1) \approx_R (k_2\ u_2)}{(x \leftarrow t_1;; (k_1\ x)) \approx_R (x \leftarrow t_2;; (k_2\ x))}\ \textsc{ECloBind}$$

$$\frac{t_1 \approx_{R_1} t_2 \quad R_1 \subseteq R_2}{t_1 \approx_{R_2} t_2}\ \textsc{EMon} \qquad \frac{t_1 \approx_R t_2}{(\texttt{interp}\ h\ t_1) \approx_R (\texttt{interp}\ h\ t_2)}\ \textsc{EInterp}$$

Figure 4.2: Relational reasoning principles

(intuitively, since handlers affect only the visible events of the tree, the leaves remain in the refinement relation). Finally, ECloBind (for "relational closure under bind") says that, to prove that two trees both built from binds are related by refinement, it suffices to find some relation U (which is existentially quantified in this rule) that relates the results of the first parts of the computation and that for any answers related by U that they might produce, the continuations of the bind are in refinement. ECloBind plays a crucial role in reasoning about ITrees—we will see in more detail below how it is used.

4.1.2 Interpretation into $\mathbb{P}$

Recall that a predicate $S : A \to \mathbb{P}$ can be thought of as a (propositionally-defined) *set* of values of type A. We write $a \in S$ for the proposition $S\ a$, which indicates that a is an element of S. Similarly, the type $\texttt{propT}_E\texttt{A}$, defined as $\texttt{itree}\ E\ A \to \mathbb{P}$, represents a *set* of ITrees, where we additionally treat set membership modulo $\approx$. We use this type in the VIR semantics to model nondeterminism in the language definition. The type $\texttt{propT}_E$ is *nearly* a monad,[16] where, intuitively, $\texttt{ret}\ x$ is the singleton set $\{x\}$ corresponding to a deterministic result, but $\texttt{bind}\ spec\ k_{spec}$ must take the union over all possible nondeterministic behaviors allowed by *spec*, when each of those might itself continue via any one of a set of possible behaviors characterized by k_{spec}. The unions are implemented in Coq by existentially quantifying over the possibilities. Formally, we have:

Definition 1. $\texttt{propT}_E\texttt{A}$ operations

$$\texttt{ret}\ (x : A) : \texttt{propT}_E\ A = \lambda(t : \texttt{itree}\ E\ A)\ .\ t \approx \texttt{ret}\ x$$

$$\begin{aligned}\texttt{bind}(spec_A : \texttt{propT}_E\ A)\ (k_{spec} : A \to \texttt{propT}_F\ B) : \texttt{propT}_F\ B = \\ \lambda(t : \texttt{itree}\ E\ B)\ .\ \exists(t_a : \texttt{itree}\ E\ A)\ \exists(k : A \to \texttt{itree}\ E\ B))\ . \\ t \approx (x \leftarrow t_a;;(k\ x))\ \wedge\ t_a \in spec_A\ \wedge \\ \forall(a : A), (a \in \texttt{returns}\ t_a) \Rightarrow (k\ a) \in (k_{spec}\ a)\end{aligned}$$

Here, `ret` lifts a value into the singleton set containing the pure itree that simply returns the value. The `bind` operation is more interesting: the resulting set contains all trees that can be factored into a subtree t_a satisfying the predicate $spec_A$, bound to a continuation k that maps every answer a that might be returned by t_a to a tree satisfying $k_{spec}\ a$. The `returns` t_a predicate is an *inductively* defined characterization of the set of values that might be returned by the computation t_a, and it is given by the definition below.

Definition 2. `Returns` t

$(\texttt{Returns}\ t) : A \to \mathbb{P}$ is the smallest set such that

- $t \approx \texttt{ret}\ a \Rightarrow a \in (\texttt{Returns}\ t)$
- $t \approx \texttt{Tau}\ u \Rightarrow a \in (\texttt{Returns}\ u) \Rightarrow a \in (\texttt{Returns}\ t)$

[16] All of the expected monad laws hold with respect to equality defined as set equivalence (up to $\approx$), except one direction of bind associativity. This is expected in the presence of nondeterminism [MHRVM20].

- $t \approx \texttt{Vis}\ e\ k \Rightarrow \exists (b : B), a \in (\texttt{Returns}\ (k\ b)) \Rightarrow a \in (\texttt{Returns}\ t)$, where $e : EB$ is an event with response type B.

The key part of its definition says that a value a is in the set `returns(Vis e k)` if there exists a value b such that $a \in$ `returns(k b)`, in other words, if the continuation `k` can return a for some b. Crucially, `returns` t_a can be a strict subset of values of type A—for instance it is empty when t_a (always) diverges. Quantifying over all $a \in A$, rather than just those that t_a might yield, is too strong and breaks many expected monad law equivalences.

The semantics of nondeterministic events like `pick` are given by interpretation via a function `interp_prop` into $\texttt{propT}_E\text{A}$, which as we saw above, represents a set of ITrees. This is sufficient for the purposes of defining the semantics, but to prove a refinement relation between two such interpretations, it is convenient to also allow the sets produced by interpreter to be "saturated" by a relation, so we parameterize the type of `interp_prop` to include a relation R on the underlying ITree type and define it as follows:

Definition 3. `interp_prop` Let $h_{spec} : E \rightsquigarrow \texttt{propT}_F$ be a (propositional) handler and $R : A \to B \to \mathbb{P}$ be a relation, then $\texttt{interp\_prop}_R\ h_{spec}$ has type $\texttt{itree}\ E\ A \to \texttt{propT}_F\ B$ and is defined as a coinductive predicate satisfying:

- If $R(r_1, r_2)$ and $t_2 \approx \texttt{ret}\ r_2$ then $t_2 \in \texttt{interp\_prop}_R\ h_{spec}\ (\texttt{Ret}\ r_1)$
- If $t_2 \in \texttt{interp\_prop}_R\ h_{spec}\ t_1$ then $t_2 \in \texttt{interp\_prop}_R\ h_{spec}\ (\texttt{Tau}\ t_1)$
- If $t_2 \approx (\texttt{bind}\ t_c\ k_2)$ for some t_c and $k_2 : C \to \texttt{itree}\ E\ B$ such that $t_c \in (h_{spec}\ e)$ and
  $\forall (c : C), (\texttt{returns}\ t_c\ c) \Rightarrow (k_2\ c) \in \texttt{interp\_prop}_R\ h_{spec}\ (k_1\ a)$, then
  $t_2 \in \texttt{interp\_prop}_R\ h_{spec}\ (\texttt{Vis}\ e\ k_1)$

This definition of `interp_prop` satisfies the general interpreter laws in Figure 4.1.[17] More importantly for reasoning about sets of behaviors is that interpretation "lifts" handlers. First, let us define what it means for a handler h to satisfy some specification:

Definition 4. Handler Correctness. A handler $h : E \rightsquigarrow \texttt{itree}\ F$ is *correct* with respect to a specification $h_{spec} : E \rightsquigarrow \texttt{propT}F$, written as $h \in h_{spec}$, if and only if $\forall\ T\ e, (h\ T\ e) \in h_{spec}\ T\ e$.

[17] Except that, as for bind associativity, the bind law holds in only one direction, again due to nondeterminism.

Then we prove that interpretation of some tree by a handler h that is correct with respect to some specification h_{spec} yields a computation whose behaviors are among those allowed by the specification. The following lemma follows by straightforward coinduction.

Lemma 1. `interp_prop` correct. For any handler $h \in h_{spec}$, any reflexive relation R, and any tree t : `itree` $E\ A$ it is the case that (`interp` $h\ t$) $\in$ $\texttt{interp\_prop}_R\ h_{spec}\ t$.

A significantly less trivial property—the proof is fairly tricky and we refer interested readers to the Coq development for details—establishes that the analog of the EInterp rule from Figure 4.2 also holds when we interpret into the `PropT` monad.

Lemma 2. `interp_prop` respects refinement. For any h_{spec} and any partial order R, if $t_1 \gtrsim_R t_2$, then ($\texttt{interp\_prop}_R\ h_{spec}\ t_1$) $\supseteq$ ($\texttt{interp\_prop}_R\ h_{spec}\ t_2$).

With the above results established, our development uses $\texttt{interp\_prop}_R$ in two ways. In the *definition* of the VIR propositional model (see the left path of Figure 3.5), we use R = `eq` (Coq's equality), in which case $\texttt{interp\_prop}_{\texttt{eq}}$ gives us the desired "sets of ITrees" semantics for modeling nondeterminism. On the other hand, we use Lemma 2 to reason about that model—in particular, to establish refinement properties, where we pick R to be nontrivial (see Section 4.2.1).

4.1.3 Equational Theory for Vellvm

We use the ITrees equational theory described above to reason about VIR code. As simple examples, it is easy to prove that $[\![3 + 4]\!]_e \approx [\![7]\!]_e$, and, with a bit more work, that $\texttt{interp\_vir}_3[\![3 + \%x]\!]_e(l, g) \approx \texttt{interp\_vir}_3[\![7]\!]_e(l, g)$ whenever $l(\%x) = 4$ (we have to interpret to L_3 to reason about the local environment l). Equations of this form let us use rewriting to "execute" the VIR semantics in any refinement proof.

It is a common compiler optimization to perform systematic rewriting of equivalent expressions, often associated with clever mechanisms used to find the optimal sequence of rewriting according to some cost function. Since expressions depend on the state, but (most) do not cause side effects,[18] rewriting expression

[18] In our memory model, pointer-to-integer casts *do* have side effects.

$$\frac{\texttt{outputs}(cfg_2) \cap \texttt{inputs}(cfg_1) = \emptyset \quad to \notin \texttt{inputs}(cfg_1)}{[\![cfg_1 +\!\!+ cfg_2]\!]_{\text{bks}}\ (f, to) \approx [\![cfg_2]\!]_{\text{bks}}\ (f, to)}\ \textsc{SubCFG1}$$

$$\frac{\texttt{independent\_flows}\ cfg_1\ cfg_2 \quad to \in \texttt{inputs}(cfg_i)}{[\![cfg_1 +\!\!+ cfg_2]\!]_{\text{bks}}\ (f, to) \approx [\![cfg_i]\!]_{\text{bks}}\ (f, to)}\ \textsc{Flow}$$

$$\frac{}{[\![cfg_1 +\!\!+ cfg_2]\!]_{\text{bks}}\ (f, to) \approx x \leftarrow [\![cfg_1]\!]_{\text{bks}}\ (f, to)\ ;;\ \texttt{match}\ x\ \texttt{with} \mid \texttt{inl}\ fto \Rightarrow [\![cfg_1 +\!\!+ cfg_2]\!]_{\text{bks}}\ fto \mid \texttt{inr}\ v \Rightarrow \texttt{ret}\ v}\ \textsc{SubCFG2}$$

Figure 4.3: Structural VIR equations (excerpt)

e into f can usually be established sound with respect to a strong notion of equivalence: they are bisimilar, *and* compute exactly the same states, *i.e.*:

$$\forall g\ l\ m,\ \texttt{interp\_vir}_4 [\![e]\!]_e\ g\ l\ m \approx \texttt{interp\_vir}_4 [\![f]\!]_e\ g\ l\ m.$$

This equivalence, much stronger than the notion of refinement that completely disregards the computed states (see Section 4.2.1), can be easily lifted to all contexts without any syntactic conditions about variables in scope. Naturally, this strong equivalence also entails the refinement relation: substitution of f for e is always sound, in any piece of syntax.

While one could always unfold the denotation of a Vellvm program to systematically use the low level equational theory of the underlying monad, it would quickly be extremely tedious and impractical. Instead, we make all representation functions opaque and provide a high level equational theory to reason directly over the syntax of Vellvm programs.

The equations pertaining to the denotation of open *cfg*s are of particular interest: we highlight a couple of them in Figure 4.3. Suppose that we are interested in the semantics of a *cfg* composed of two components cfg_1 and cfg_2. Equation `SubCFG1` allows us to simply disregard cfg_1 granted that we *syntactically* check that we are jumping into cfg_2, and that it cannot jump back into cfg_1: we can reason about sequence. Similarly, equation `Flow` helps to reason about branches: if cfg_1 and cfg_2 are (syntactically) completely independent, the semantics of their union is simply the semantics of whichever subgraph we enter. Finally, equation `SubCFG2` states that we can always temporarily forget about cfg_2 by starting execution in cfg_1 (without cfg_2) and then proceeding afterward with the whole graph in scope.

Lifting expression equivalence The compositionality of the semantics allows us to lift equivalences of sub-components to the context. In general, this process is non-trivial: an optimization eliminating an instruction from a block can naturally only be lifted into a context where the assigned variable is dead. However, compositionality allows us to prove the syntactic conditions on the context under which the substitution is valid once and for all, and reason locally when proving optimizations.

Reasoning about the control flow in a stateless world: block fusion While substitution of equivalent expressions is the canonical example of a local reasoning enabled by compositionality, we prove a simple block fusion optimization to illustrate the benefits from the modularity of the semantics.

The optimization scans a *cfg* until it finds a block bk_s such that: (1) bk_s has a direct jump to some block bk_t, (2) bk_t admits only bk_s for predecessor, (3) bk_s and bk_t are distinct and (4) bk_t has no phi-node. If it finds such a couple, it removes them from the graph, adds their obvious sequential merge, and updates the phi-nodes of the successors of bk_t to expect instead a jump from bk_s.

This optimization only modifies the control flow of programs. As a consequence, we can establish the correctness of the transformation without interpreting any event in the graph. Assuming a *well-formed graph* G—all block identifiers are unique, and phi-nodes only expect jumps from predecessors—we establish:

$$[\![G]\!]_{\text{cfg}} \approx [\![\texttt{fusion\_block}\ G]\!]_{\text{cfg}}.$$

The result is established with no layer of interpretation, abstracting away from the state. Once again, the equivalence can be transported by interpretation all the way to the top-level semantics.

The proof of this result on closed *cfg*s derives from a bisimulation established on open *cfg*s. At that level, the post-condition established is not straightforward equality of computed results: the provenance of a jump out of the graph may have been changed by the transformation. This subtlety disappears when specialized over closed graphs, resulting in this simple $\approx$ relation.

It is worth noting that establishing the simulation for this optimization must be done using an explicit coinductive proof — in contrast to transformations such as loop unrolling for instance. To the best of our knowledge, the axiomatization of the ITree loop iterators is indeed not expressive enough to reason about such fusion because it requires matching two iterations of a body to a single iteration of the body for some

values of the accumulator.

4.2 VIR Refinement and Relational Reasoning

4.2.1 VIR Refinements

The refinement machinery defined in Sections 4.1.1 and 4.1.2 lets us give a clean semantics to VIR's underspecified values and undefined behaviors. Moreover, we can straightforwardly define appropriate refinement relations that work at any level of interpretation shown in Figure 3.5 such that refinement at one level implies refinement at the next. This arrangement means that we can prove the correctness of program transformations at whatever level is most suited to the task.

Uvalue refinements In order to prove refinements between programs we need to know what it means for a value to be a refinement of another in Vellvm. For concrete values, this is straightforward: refinement is reflexive and anything can refine `poison`. However, as we have established, LLVM makes use of (typed) under-defined values, which can represent arbitrary (typed) sets of concrete values. The refinement relation is thus given by inclusion between the sets of concrete values that can be represented by a $\mathcal{V}_u$. At the base case we have $[\![\texttt{undef}_\tau]\!]_C = \{v \mid v \textit{ is a concrete value of type } \tau\}$ where the notation $[\![x]\!]_C$ represents the set of concrete values of x. For instance $[\![\texttt{undef}_{i64}]\!]_C$ is the set of all 64-bit integers. Since $\mathcal{V}_u$ contains "delayed" computations like $\texttt{2} \times \texttt{undef}_{i64}$, the sets are nontrivial. In this case, we have that $[\![\texttt{2} \times \texttt{undef}_{i64}]\!]_C$ is the set of *even* 64-bit integers.

Definition 5. Uvalue refinement. We say that $a \in \mathcal{V}_u$ *refines* $u \in \mathcal{V}_u$ precisely when $[\![u]\!]_C \supseteq [\![a]\!]_C$ and we write $u \geq a$ for that relation.

Uvalue refinement, namely $t_1 \approx_{\geq} t_2$, gives us the base notion of what it means for VIR programs to be related at the structural level L_0 in which none of the LLVM events have yet been interpreted. At each subsequent layer of interpretation, we are free to choose what observations of the computation are considered relevant for program equivalence. Following usual practice, we want those observations to be as liberal as possible to permit as many program transformations as we can—ultimately, we choose to observe (for programs without undefined behavior) only the (possibly infinite) sequence of external function calls

and the value (if any) returned by the program (up to uvalue refinement). A program with undefined behavior after some finite sequence of external calls is refined by any program that exhibits the same series of calls and then can behave arbitrarily. In particular, all the state (the memory, stack, local and global environments) is irrelevant. We express this irrelevancy by using the total relation: $\mathtt{T} = \lambda x\ y\ .\ \top$, which relates all elements of its domain, for those components of the state. Thus, at L_1 we use the refinement $t_1 \approx_{\mathtt{T} \times \geq} t_2$, and each subsequent state interpretation adds another $\mathtt{T} \times -$ to the relation.

Levels 1-4 introduce pieces of state: the compiler has no responsibility to preserve them, as long as programs exhibit the same series of external calls and return values are related by $\geq$. Thus, at L_1 we use the refinement $t_1 \approx_{\mathtt{T} \times \geq} t_2$, where $\mathtt{T}$ is the total relation. Each subsequent state interpretation adds another $\mathtt{T} \times -$ to the relation.

A consequence of EInterp is that refinement at a level implies refinement at later levels. For instance, we have:

Lemma 3. L_0 to L_1 refinement
For any global state g, if $t_1 \approx_{\geq} t_2$ then $(\texttt{interp\_global}\ t_1\ g) \approx_{\mathtt{T} \times \geq} (\texttt{interp\_global}\ t_2\ g)$.

For $\mathcal{P}$ and $\mathcal{U}$, this approach falls short as they lift their events into the $\texttt{propT}_E$ monad. We instead use Lemma 2 to lift a tree refinement to set inclusion, which gives us the desired definition of top level refinement for VIR programs.

Soundness of the executable interpreter A pleasing—and very useful—consequence of the above refinement lemmas is that it is almost trivial to prove that the executable VIR interpreter's program behaviors are permitted by the VIR semantics. The following theorem follows directly from Lemma 1 by showing that the executable handlers for $\mathcal{P}$ and $\mathcal{U}$ are correct with respect to their propositional specifications (which is entirely straightforward, since for $\mathcal{P}$ the only requirement is that the handler choose a concrete value of the appropriate type and $\mathcal{U}$ allows any behavior at all).

Theorem 1. VIR interpreter soundness For any program p, $(\texttt{interpreter}\ p) \in \texttt{model}\ p$.

4.2.2 Floyd-Hoare-Style Forward Relational Reasoning

From the point of view of reasoning, we can think of the R of $\approx_R$ as a *relational postcondition* satisfied by two bisimilar computations. Heterogeneous relations $R : A \to B \to \mathbb{P}$, which relate ITrees of *different* return types, are useful when connecting the behaviors of two ITrees, as is typical when reasoning about a compiler's or program transformation's correctness.

In this case we prove $t_{src} \approx_{ST} t_{tgt}$ for source and target trees that encode their respective semantics. Relation ST establishes the connection between source and target states, which might be of different types. The general relational properties work together allowing a verification strategy following this recipe: (1) rewrite t_{src} and t_{tgt} using the monad laws to normalize the trees by unnesting `bind`s, eliminating `Tau`s and "bubbling" `trigger`s and variables to the top; (2) use ECloBind to break down the term into simpler pieces that use assumptions about the variable from the environment or lemmas about the triggered event's handler; (3) conclude by ERet or independent lemmas established over the correctness of these smaller pieces.

When working with VIR in particular, we apply the same recipe but change the granularity of our "atomic" computations: we do not bubble up `trigger`s but instead we bubble up denotations of individual instruction, keeping their representation opaque and relying on axiomatizations of their behaviors.

Here ECloBind is analogous to the usual "sequencing" rule from Hoare logic, stating that to establish a postcondition R, we need to find *some* intermediate relation U that acts as a postcondition for the first tree. While this relation shares some similarities with the more traditional simulation relation used in backward-simulation-based approaches, it does not have to be global: each application of the ECloBind rule may introduce a different relation, much in the style of Floyd-Hoare forward proof. EInterp allows commuting $\approx_R$ through interpreters.

4.2.3 Expressing Functional Properties of VIR: a Derived Unary Program Logic

Using `eutt`, we can express and prove in the same framework the equational theory of VIR, the correctness of VIR to VIR optimizations, or, when used heterogeneously, to prove transformations relating VIR to other languages. When conducting such proofs, $\approx_R$ can be thought as a termination, trace equivalence, sensitive

relational program logic, where ECloBind acts as a cut rule.

An important missing aspect is the ability to express (and use in refinement proofs) functional properties of specific programs. To this end, we introduce a unary interpretation of `eutt`: given t : `itree` $E\ X$ and $P : X \to \mathbb{P}$, we write $t \hookrightarrow\ P \triangleq t \approx_{(\lambda(x,y)=>P\ x)}\ t$.[19]

This unary relation inherits from `eutt` a sequencing rule, and allows for the combination of postconditions of a same computations with respect to the usual logical combinators. This program logic has a partial correctness interpretation: all finite branches of the tree lead to the postcondition, but some branches may diverge.

We prove that such unary judgments can be established independently and easily invoked during refinement proofs. We derive to this end a new version of the ECloBind rule:

$$\frac{t_1 \hookrightarrow Q_1 \quad t_2 \hookrightarrow Q_2 \quad t_1 \approx_U t_2 \quad \forall\ u_1, u_2, U(u_1, u_2) \Rightarrow Q_1\ u_1 \Rightarrow Q_2\ u_2 \Rightarrow (k_1\ u_1) \approx_R (k_2\ u_2)}{(x \leftarrow t_1;; (k_1\ x)) \approx_R (x \leftarrow t_2;; (k_2\ x))}$$

A semantically simple, but practically crucial example of application of this rule is during the proof of correctness of the block fusion optimization. During the simulation, the semantics of terminators of blocks that are not the fused one are matched one against another trivially—they are the same. However, the correctness of the transformation requires us to prove that we will not jump to the fused block. To do so, we use this unary predicate to prove as a property of the semantics that the denotation of blocks can only return labels that are syntactically in the successors of the block. While this fact is intuitively trivial, establishing it requires a case analysis on the terminator and an explicit processing of its semantics — something that one one does not want to inline in a refinement proof.

Indeed, in the special case where a tree t is related to itself, i.e. $t \approx_R t$, we have that any value r returned by the tree will satisfy $R(r, r)$. We can encode usual Floyd-Hoare-style postconditions as the "diagonal" of some appropriately chosen R. It's worth noting that R need not be total, and choosing partial relations can be useful. For example, one can prove that $t \approx_\emptyset t$ holds, if and only if t *never* returns, that is, every branch diverges or ends in a visible event with void response type (and so there can be no answer from the environment) We can also characterize the predicate `Returns` $t\ a$ as the smallest set such that the relation given by $R_t = (\lambda x\ y\ .\ x = y\ \wedge$ `Returns` $t\ x)$ satisfies $t \approx_{R_t} t$, or, equivalently `Returns` t is the domain of the smallest R such that $t \approx_R t$—facts that are needed to establish some of the equivalences on `interp_prop`.

[19] We show that the definition $t \approx_{(\lambda(x,y)=>P\ x \wedge x=y)}\ t$ is equivalent.

4.3 Related Work

There is a large literature on formal verification of software artifacts [RPS+19a]. Here we focus on the works most closely connected to the VIR development.

Verified compilers The CompCert [Ler09] C compiler was a pivotal development in the domain of verified compilation, tackling a real world programming language and nontrivial optimizations formally in the Coq proof assistant [Tea20]. CompCert's success has fueled numerous projects aiming to expand upon its results. Examples include the addition of concurrency [ŠevčíkVN+13], the support for linking open programs [SCK+19, PA19], or the preservation of security properties [BBG+20]. Others have developed their own infrastructure in order to tackle different languages: the CakeML [KMNO14] project has developed a complete verified chain of compilation for ML

Compositional verification CompCert's original theorem suffered the major restriction of applying only to *whole* programs, thereby disallowing linking. A rich line of works [KKH+16, NHK+15, SBCA15, WWS19, SCK+19] has sought to relax this restriction via compositional simulation techniques. These works have struck different balances between expressiveness and proof obligations. Patterson and Ahmed [PA19] have recently proposed a framework allowing to compare these result. Another point of comparison comes from CertiKOS'[GKR+15, GSK+18] certified (concurrent) abstraction layers. These layers share many properties with the relational reasoning techniques we describe in Section 4.1, albeit the connections among such techniques requires further investigations.

Non-small step approaches Interaction trees were developed as a general-purpose representation for effectful, interactive, and possibly-divergent code [XZH+20] and, besides programming language semantics, have been used for specifying network servers[KLL+19]. One of their distinguishing features is the pervasive use of coinduction, which is crucial to support recursion and iteration, but requires sophisticated proof techniques [HNDV13, ZHHZ20]. Leroy and Grall [LG09] have experimented with coinduction to model divergence in the operational semantics of a lambda calculus, proving type soundness and verifying a compiler.

Several other exceptions to using relational small-step semantics approach are notable. Chlipala [Chl10] verifies a compiler for a language shallowly-embedded in Coq. The language in question is total, and

hence does not require recursion combinators; nevertheless, this style of semantics admits modular and compositional proof techniques similar to ours. Owens *et al.* advocate for big-step semantics "akin" to an interpreter [OMKT16]—they use a "clock" for "fuel" to bound recursion, thereby sidestepping the need for coinduction, but requiring proofs to take the fuel into account via step-indexed logical relations. We take this idea a step further and use a true interpreter, embracing the coinductive structure directly. This means that we can more readily reason equationally about ITrees semantics. The tradeoffs between such step-indexed and coinductive approaches deserve more attention.

JSCert The JSCert project [BCF+14], which formalizes JavaScript semantics in Coq, uses Charguéraud's "pretty big step" semantics [Cha13]. This approach, like interaction trees, promotes compositionality by allowing the semantics to be defined inductively on the syntax; it also uses coinduction to handle diverging terms. Unlike interaction trees, however, "pretty big step" semantics are still defined relationally. The authors implemented a separate executable version of semantics, JSRef, that is intended to serve as a reference implementation. Nontrivial proof effort (we estimate that it takes several thousand lines of Coq code) is required to prove the correspondence of the JSCert pretty-big step relational specification with the JSRef executable version. The authors write: "We believe that both JSCert and JSRef are necessary: JSCert, unlike JSRef, is well-suited for developing inductive proofs about the semantics of JavaScript; JSRef, unlike JSCert, can be used to run JavaScript programs." In contrast, we have shown that interaction trees meet both desiderata: they are well suited both for inductive proofs and for excecutability.

LLVM and C Semantics The Vellvm project [ZNMZ12] has focused its attention on LLVM's intermediate representation and verified complex optimizations over it [ZNMZ13]. The subset of LLVM IR that Vellvm handles is fairly similar to VIR's, albeit marginally outdated and less rich in features. More importantly, their semantics are radically different: Vellvm relies on a traditional small step relation parameterized by the whole *mcfg* considered. Proving any transformation of programs changes the *mcfg* in play and therefore requires to relate two distinct semantics, which in turn requires heavy invariants. Our approach leads to a significantly cleaner semantics illustrated by the removal of the heavy notion of program counter that Vellvm manipulates.

A number of other projects have formalized various subsets of C [Ell12, KW15, MML+16, MGD+19], or LLVM IR—such as Crellvm [KKS+18], K-LLVM [LG20a], and the Alive [LMNR15, MN17] projects. The

Crellvm project uses the Vellvm semantics internally, so it inherits the same fundamental structure. The K-LLVM framework, implemented in $\mathbb{K}$ [Rc10], is perhaps the most complete executable semantics for the LLVM IR and has been used for extensive testing. There is some work connecting $\mathbb{K}$ specifications to Isabelle/HOL [LG18], but, to our knowledge, the viability of that approach for formal proofs of, *e.g.*, compiler correctness, remains to be demonstrated.

Alive [LMNR15], and its recent successor Alive2 [LLH+21], focus on finding bugs in the LLVM IR implementation by using translation validation to check for mis-optimizations. Alive2 is able to run directly on LLVM's source code, and has demonstrated an impressive efficacy. Although their objective, bug-finding, differs from ours, formal verification, both projects share the need for formalizing parts of LLVM IR's semantics. One significant difference from our approach is that Alive2 only formalizes LLVM's semantics *implicitly*, through the encoding its validator performs to check an optimized program. Moreover, the Alive2 semantics properly avoids collapsing `undef`'s non-determinism when interacting with the memory mode—contrary to our current memory model—but it under-approximates its semantics elsewhere (per the paper [LLH+21], they "only allow an argument to be either fully `undef` or not `undef` at all"). Moreover, as far as we can tell, Alive2 does not support pointer-to-integer casts. These approximations are sound for bug-finding, only straying Alive2 farther from completeness, but are incompatible with verification. Those differences aside, Alive2 could be a rich source of test cases for VIR. One challenge is that most of the Alive2 test cases aren't executable (they are open program fragments before/after optimization), so it is not clear how we can use them in conjunction with VIR semantics. One could state the expected refinement relation between such program fragments as theorems and try to prove them, but finding a more automatic way of using Alive2 tests, perhaps by making them executable by (randomly) instantiating their free variables, would be desirable.

Others have focused their attention more specifically on characterizing the LLVM's undefined behaviors [LKS+17] and its concurrency semantics [CV17]. Even more specifically, modeling realistic memory models for LLVM is an active area of research in itself [KHM+15a, LABS12, MGZ15, LHJ+18b], which is closely connected to similar efforts for low-level languages like C [MGD+19]. None of these works rely on a mechanized denotational semantics as we do, which constitutes the core of our contribution. Nonetheless, many of these works cover semantic features that VIR does not yet tackle, and as such are major sources of

inspirations for the future of VIR. In particular, improving the VIR memory model is an important next step. For example, the memory model presented here does not support storing `undef`, or, more generally, elements of $\mathcal{V}_u$ in the heap, a limitation shared with Vellvm and most of the prior work. This forces the semantics to use a $\texttt{Pick}^{\mathcal{V}}()$ event as part of a `store` operation, which in turn invalidates store-forwarding optimizations (where a `load` following a `store` to the same location is replaced by the stored value). There are similar issues with respect to the proper treatment of `poison` and `undef` with respect to intrinsics and external function calls, that remain to be resolved. VIR currently models the behaviors of `GEP` and pointer-to-integer casts via interactions with the memory model, a natural model for these constructions as their semantics depends on details about how data is laid out in the heap; however, this also means that proving correctness of program transformations that move or eliminate such operations is nontrivial. For instance, in addition to the usual requirements about the scopes of program identifiers, one would have to prove that a call to `alloca` doesn't affect the result of `GEP` in order to move a use of the `GEP` instruction around an `alloca`. The memory model provided in Juneyoung et al. [LHJ+18b] addresses this issue, ensuring that `GEP` is truly pure, and should hence be part of the design of the future rework for VIR's memory model. Despite such remaining challenges, we are optimistic that the design of VIR provides the necessary ingredients to model LLVM semantics with higher fidelity—ITrees provide the ability to introduce and handle nondeterministic events at various levels of interpretation, and the use of the $\texttt{propT}_E$ monad provides a rich semantic space for describing the allowed behaviors.

LLVM IR's semantics is complex, but also evolving: subtle interactions between `poison` and `undef` led to a proposal [LKS+17] to simplify the under-defined values semantics via a `freeze` instruction, which in VIR affects where $\texttt{Pick}^{\mathcal{V}}()$ events occur; it was extremely straightforward to add support for `freeze` to VIR. Similarly, there is ongoing work on a "provenance" mechanism for specifying which pointer-to-integer casts are allowed, which is also subject to change. Maintaining a formal development of the size of VIR with such evolutions is a major challenge that we believe can be mitigated by the modularity of its semantics.

Executability Our use of monadic interpreters based on interaction trees allowed us to get an executable VIR semantics with very little effort, which enabled testing of the semantics early on. As mentioned above, one main contribution of the JSCert project is the proof of correspondence between the specification and a reference implementation. Similarly, both the Vellvm and CompCert projects have spent substantial

efforts during their development to define interpreters and prove them equivalent to the relational semantics. Maintaining two artifacts incurs the cost of synchronizing them, which can become especially painful as a language evolves. Any change to the semantics has to be echoed in the interpreter and the proof fixed. With our approach, almost all of the semantics is shared with the interpreter, with the exception of the implementation of the non-deterministic effects of the language.

As a measure of the impact of this design on the part of the development related to the interpreter, we offer an (admittedly) rough comparison with Vellvm, which, as it also aims to formalize LLVM IR, is the Coq development most like VIR.

	Vellvm	VIR
(extra) lines of Coq code to define interpreter:	~500	~130
(extra) lines of Coq code to prove refinement:	~1000	~250

The overhead of verifying the VIR interpreter is significantly smaller; Vellvm supports significantly fewer LLVM types and operations than VIR, so the numbers in the Vellvm column would be somewhat larger for a "fairer" comparison. The Vellvm proof that the interpreter refines the semantics proved the result only for a single small step, eliding the coinductive outer reasoning to establish co-termination of the two semantics, something that we get for free. The VIR results are therefore simpler, shorter, and much stronger. With respect to the resulting interpreters, there are also significant differences: The Vellvm semantics (due to its propositional nature) axiomatized properties about global memory and state initialization and, consequently does not extract an executable memory model (it punts to an C implementation), whereas VIR extracts the memory model too—the Vellvm interpreter itself is much less trustworthy than VIR's.

Part II

A Layered Equational Framework

Chapter 5

Layered Monadic Interpreters

In Part I, we saw a modular and executable semantics for the sequential semantics of LLVM IR, based on layered monadic interpreters. This part discusses a general *metatheory* for reasoning about layered interpreters, giving an extensible theory for lifting interpreters and structural rules.

Monadic computations built by interpreting, or *handling*, operations of a free monad are a compelling formalism for modeling language semantics and defining the behaviors of effectful systems. The resulting layered semantics offer the promise of modular reasoning principles based on the equational theory of the underlying monads. However, there are a number of obstacles to using such layered interpreters in practice. With more layers comes more boilerplate and glue code needed to define the monads and interpreters involved. That overhead is compounded by the need to define and justify the relational reasoning principles that characterize the equivalences at each layer.

This chapter addresses these problems by significantly extending the capabilities of the Coq *interaction trees* (ITrees) library, which supports layered monadic interpreters. We characterize a rich class of *interpretable monads*—obtained by applying monad transformers to ITrees—and show how to generically lift interpreters through them. We also introduce a corresponding framework for relational reasoning about "equivalence of monads up to a relation R". This collection of typeclasses, instances, new reasoning principles, and tactics greatly generalizes the existing theory of the ITree library, eliminating large amounts of unwieldy boilerplate code and dramatically simplifying proofs.

Several features of ITrees make them well suited to defining semantics. First, event signatures can be combined through a disjoint union operation: an ITree of type `itree (E +' F)` can have events drawn from either `E` or `F`, allowing for a modular definition of computations. Second, it is possible to write interpreters for the events in an ITree. We write `h : E ~> M` to mean that `h` is an *event handler* that maps events in the signature `E` into computations in the monad `M`. Then `interp h : ∀ A, itree E A → M A` maps an ITree computation into a monadic computation by "handling" the events in `E`, which gives them a semantic definition according to the operations of `M`. For instance, one might interpret "read" and "write" events into

a state monad. Taken together, the signature of events and event handlers allows for the decomposition of effects into stages, modularizing the semantics in a style akin to *algebraic effects* [PP03b]. Third, by implementing the `itree E` monad coinductively, divergence is given native support, allowing for generic fixed-point combinators to be defined without compromising the (lazy) computability of the resulting definitional interpreters. The type `itree voidE` (where `voidE` is the empty event signature, and is the unit of `+'`) is isomorphic to Capretta's *delay monad* [Cap05], which was designed specifically to represent nonterminating computations in type theory. Finally—and perhaps most importantly for the purposes of formal verification—ITrees come equipped with a rich *relational theory*: `t` $\approx_R$ `u` means that ITrees `t` and `u` are *weakly bisimilar* and produce answers related by R. This relational theory generalizes the notion of program equivalence (given an equivalence relation R), and the relational theory can be lifted to the other monads obtained by interpretation of ITrees, yielding a way to reason formally about monadic semantics.

Past experience suggests that ITrees provide a very comfortable semantic toolbox, whether to reason about the functional correctness of programs or the correctness—in terms of equivalence or refinement—of program transformations. The original paper [XZH+20] illustrated the approach by proving the correctness of a toy compiler from IMP, an imperative language, to ASM, a simplified assembly language. The story is pleasantly clean: a termination-sensitive result is established without the need for any explicit coinduction; the compiler is proved in a compositional fashion (open pieces of programs can be related to their compilation in isolation); and the proof method is heavily equational, relying on $\approx_R$ to allow tedious but easy-to-produce proofs by rewriting.

The ITrees approach has also been shown to scale. The most ambitious project in this realm, to date, is VIR, as described in Part I. In VIR, the semantics of the sequential fragment of LLVM IR has been formalized using ITrees, leading to a remarkably simplified semantics when contrasted with the previous iteration of the project [ZNMZ12], which was based on a more traditional operational semantics. In particular, the program counter and the heavy invariants it entails have disappeared. When it comes to reasoning, the benefits of the ITree semantics have been evaluated in several dimensions. With respect to LLVM IR transformations, simple peephole optimizations admit local, simple proofs, and proving a block fusion transformation correct can be done at a high level of abstraction—verifying a transformation that only impacts control-flow does not depend on the implementation of the state.

While the results above are a promising testament to the viability of ITree-based monadic interpreters for formal verification, things do not remain as smooth at the scale of Vellvm as they were for IMP and ASM (and even at the small scale, the situation can be improved). In this dissertation, we identify and provide solutions to the following pain points that are impediments to using ITrees at scale:

1. As mentioned above, the primary feature of the free monad is its extensibility: there is a natural inclusion of type `itree E` into `itree (E +' F)` with an event signature enriched via a coproduct. That inclusion necessitates a renaming (to add the left injection), and such renamings can be discharged with a simple typeclass (provided by the ITree library). However, when it comes to using `interp`, which handles *all* of the events of an ITree, the existing typeclasses are inadequate: a handler `h : E ⤳ M` cannot readily be lifted to a handler `h' : (E +' F) ⤳ M` (and such a lifting may not always be possible), which means that the user has to hardcode the syntactic structure of the event signatures for handlers and interpreters, breaking modularity. Existing techniques, for instance the automatic injections used in Swierstra's data types à la carte [Swi08] (and re-implemented in the ITree library), do not provide an adequate solution.

2. When modularly structuring a semantics as complex as Vellvm's, interpretation takes place in layers: several interpreters are successively composed, each handling different events. However, while free monads, and ITrees in particular, give the freedom to interpret *into* any monad, we are now left to ponder how to interpret *from* other structures. For example, consider a handler `h : E ⤳ stateT S (itree F)` that interprets `E` events into the monad `stateT S (itree F)`, where `S` is the notion of state considered and `stateT S` is the state monad transformer defined as `fun M R ⇒ S → M (S * R)` and `interp h` is of type `itree E ⤳ stateT S (itree F)`. If we have another handler `f : F ⤳ M`, we would like to *compose* the interpreters: `(interp f) ∘ (interp h)` should have type `itree E ⤳ stateT S M`, but the left-hand use of `interp` works over a state-transformed ITree, not an ITree, so this code does not typecheck—we require a much more general notion of "interpreter" to build such interpretation stacks. With the existing ITree library, such compositions must be constructed painfully and repetitiously by hand.

3. Beyond these issues that arise when *building* complex interpreter stacks, we also need to be able to

reason about the resulting computations. Consequently, we need versions of the relational theories (monad laws, *etc.*) for every monad in sight! In practice, this means that we need mechanisms to *lift* the equational theory of one monad through a monad transformer to obtain a transformed theory. It is not enough to be able to construct proofs of equivalences or refinements, we also need *inversion principles* to extract information from such proofs. It is not at all obvious how to build such a relational reasoning framework generically.

Contributions In this part, we propose solutions to the problems described above. After giving the necessary background about the ITree library and its current pain points in Section 5.1, we tackle the problem of building layered monadic interpreters.

Section 5.2 introduces several novel typeclasses: `Trigger`, `Subevent`, and `Interp`, along with a generic operation called `over`, that collectively address the issues with modularity of event signatures and construction of layered interpreters. Along the way, we see how to define instances of our new typeclasses for a variety of frequently-used monads: `itree E`, `state`, `error`, and `Prop`.

Section 6.1 develops new tools for relational reasoning, centered on a typeclass `eqmR` (for "equivalence of monads up to `R`")—a generalization of ITrees weak bisimulation, $\approx_R$. We identify a suitable axiomatization of its properties and define operations that lift it through monad transformers. A key, and, we believe, novel idea here is the concept of the *image* of a monadic computation, which precisely characterizes its possible results. The image is defined purely in terms of `eqmR`, and it is a key ingredient needed to define the equational theory.

Section 6.2 uses `eqmR` to define the properties of the new typeclasses introduced in Section 5.2, providing a rich framework for reasoning about layered monadic computations.

Section 7 describes how this framework pans out in practice, where a key contribution is a collection of tactics that provides type instantiations to help disambiguate typeclass resolution. We evaluate the effectiveness of this new infrastructure by porting the IMP-to-ASM proofs from the original ITrees development—we find that the resulting proofs are substantially less ad hoc, more compositional, and considerably simpler.

Section 7.3 situates our contributions with respect to related work, and Section 7.4 concludes with a discussion about further techniques and future work.

Implementation The ideas in this part are packaged as a Coq development [YZZ22] and all of the properties presented here have been proved in Coq. Although some of our contributions are Coq-specific (*e.g.*, the need to deal with `Proper` instances for Coq's setoid rewriting and the details of our tactics), we believe that most of the typeclasses and constructs proposed here could be profitably implemented in other settings as well.

5.1 Interaction Trees and Monadic Interpreters: Background and Shortcomings

Interaction Trees [XZH+20] (ITrees) have recently emerged in the Coq ecosystem as a rich toolbox to build compositional and modular monadic interpreters. One of its main benefits comes from its equational framework that allows reasoning about equivalence and refinement of computations. Through this section, we introduce the necessary background information to understand the theoretical and practical limitations that arise when using the framework at scale.

5.1.1 Scaling Up: The Shortcomings of Layered Monadic Interpreters

This promise of modularity is deceitful when used at scale: layering monadic interpreters can become unwieldy.

Let's take a look at the VIR semantics to see what can go wrong. When dealing with large languages, the naïve interpretation scheme sketched above, which interprets *all* of its events at once, is undesirable for a couple reasons. First, some effects may be implemented in terms of others: memory operations, for instance, may introduce undefined behavior events. Second, decoupling the interpretation of different categories of events modularizes the monadic structure they introduce, improving the modularity of the semantics and the robustness of the

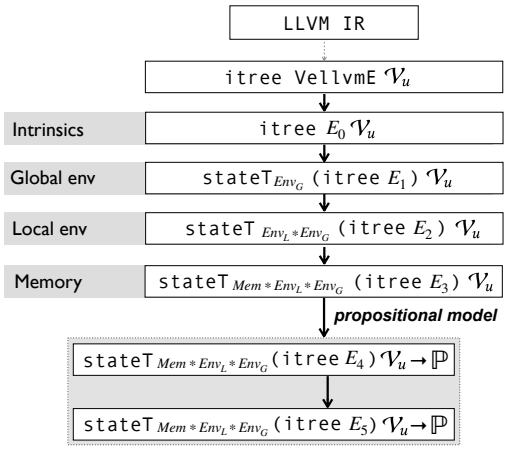

Figure 5.1: Vellvm's semantics: a stack of interpreters

```
Definition handle_state {E} :
  stateE ~> stateT S (itree E) :=
    fun _ e s => match e with
                 | Get    => Ret (s, s)
                 | Put s' => Ret (s', tt)
                 end.
```

```
Definition pure_state {S E} :
  E ~> stateT S (itree E) :=
    fun _ e s => Vis e (fun x => Ret (s, x)).

Definition interp_state {E} :
  itree (stateE +' E) ~> stateT S (itree E) :=
    interp (case_ handle_state pure_state).
```

Figure 5.2: State interpreter from the ITree library

formalization. Such decoupling leads to proof techniques allowing for some of the effects to remain *uninterpreted* during a proof.

Thus, it is desirable that complex monadic interpreters be organized as *layers* of interpretation. Figure 5.1 reproduces the structure of VIR's interpreter: a piece of LLVM IR syntax is first represented as an ITree over a rich signature before its effects are successively implemented, leading to a richer monadic structure at each layer. Intuitively, this sequence of interpreters implement: 1. calls to pure intrinsics interpreted as pure Coq functions, 2. the global state interpretation, 3. the local state interpretation, 4. the memory model interpretation, 5. the nondeterministic concretization of `undef` values, and 6. the nondeterministic refinement of undefined behaviors.

However, there are two glaring issues for both defining and working with layered interpreters, which are presented as follows.

5.1.1.1 Problem 1: Lifting Partial Handlers for Whole Signatures

The first difficulty is defining a handler for partial interpretations, *i.e.*, interpreting a particular effect out of a sum of events while leaving the others uninterpreted. The toy example that interprets away the `cellE` signature while preserving `printE` should be factored into a part that handles `cellE` in isolation and a generic part that injects the remaining events.

The ITree library provides some applicable tools. Figure 5.2 reproduces the standard interpreter for memory events, `stateE`, where `S` is the notion of state considered and `stateT S` is the state monad transformer defined as `fun M R ⇒ S → M (S * R)`. To be reusable, the `stateE` handler corresponding to our example `cellE` handler is defined in `handle_state`, which is parametric in the leftover ambient signature `E`. The branch implementing `printE` is captured generically in `pure_state`, since the implementation does not depend on the effect that remains uninterpreted. Finally, since `+'` forms a coproduct for an indexed function,

```
Definition handle_local {E} `{FailureE -< E} :
    (LocalE k v) ~> stateT map (itree E) :=
fun _ e env ⇒
  match e with
  | LocalWrite k v ⇒ ret (add k v env, tt)
  | LocalRead k    ⇒
    match lookup k env with
    | Some v ⇒ Ret (env, v)
    | None   ⇒ fail
    end
  end.

Variable (E F G H: Type → Type).
Context `{FailureE -< G}.
Notation Effin := (E +' F +' LocalE +' G).
Notation Effout := (E +' F +' G).
```

```
Definition E_trigger {M} : ∀R, E R → stateT
     M (itree Effout) R :=
  fun R e m ⇒ r ← trigger e ;; ret (m, r).

Definition F_trigger {M} : ∀R, F R → stateT
     M (itree Effout) R :=
  fun R e m ⇒ r ← trigger e ;; ret (m, r).

Definition G_trigger {M} : ∀R , G R →
    stateT M (itree Effout) R :=
  fun R e m ⇒ r ← trigger e ;; ret (m, r).

Definition interp_local_h := (case_
    E_trigger (case_ F_trigger (case_
    handle_local G_trigger))).

Definition interp_local : itree Effin ~>
    stateT map (itree Effout) :=
interp_state interp_local_h.
```

Figure 5.3: Interpreting Vellvm's register map

a generic `case_` combinator builds the handler used to interpret an arbitrary computation containing `stateE` events, as shown in `interp_state`.

So surely we should be able to happily simplify the definition of `handle_cell` by using the standard library's `stateE` events instead of specialized `cellE`, and directly defining `interp_cell` as `interp_state`? Unfortunately, we cannot! A slight mismatch creeps in: our previous computations have been defined over the `printE +' cellE` signature, while `interp_state` forces `stateE` to be in the head position: `stateE +' printE`. We are forced to either change our definitions to line up the signatures, or to duplicate the definitions.

These structural constraints add up to create bureaucratic clutter in large-scale developments. Figure 5.3 reproduces the Vellvm interpreter layer implementing the register map—this interpreter defines the translation from the Global env level to the Local env level of Figure 5.1. With this setup, events in the register map, defined in the `LocalE` signature, are structurally in the third position of the signature (see `Effin`) at the site where the interpreter of this handler is used, and three auxiliary definitions for triggering `E`, `F`, and `G` events along with fiddly uses of `case_` are needed to define `interp_local_h`.

Also, notice the constraint `Failure -< G` in the context, which informally represents "any event `G` that supports failure". This corresponds to the constraint mechanism introduced by Swierstra's Data Types à la Carte [Swi08] to automate the renaming of triggered events. We discuss its limitations and introduce a more expressive substitute in Section 5.2.3.

Simply reorganizing the shape of the signature is impossible due to constraints that do not compose in various places of the semantics, forcing developers to painstakingly handcraft special-case interpreters. Naturally, this definition is very fragile to the introduction of additional effects, in blatant contradiction to the modularity otherwise achieved.

5.1.1.2 Problem 2: Building Layered Interpreters

The second issue in building layered interpreters is the complete lack of a theory for interpreting computations from monads other than ITree. The existing `interp` implementation is parametric in its *target* monad, but it is not in its *source*: `interp (h : E ~> M) : itree E ~> M` is defined for any iterative monad `M`.

Sticking to the same implementation of the register map, we see in Figure 5.1 that it occurs *after* the implementation of the global state. As a consequence, the domain of computation manipulated at this stage is already not a plain ITree, but rather `stateT EnvG (itree E1)` for some signature `E1`. This problem reoccurs at each subsequent level.

The previous approach to this issue is defining ad hoc solutions for each situation. One can "interpret" a computation in `stateT EnvG (itree E)` by exposing the definition of the `stateT` transformer as a pure function of the initial state, and therefore interpreting the computation pointwise. Such construction breaks the abstraction of layered monadic interpretation.

In Section 5.2, we introduce the appropriate abstract structures necessary for the principled construction of layered interpreters, providing a general and clean solution to both problems from the programmatic perspective. Of course, *defining* monadic computations in this modular, layered way is only half of the problem. For formalization, we also need to develop the corresponding *metatheory* for reasoning equationally about these constructions. That is the subject of Chapter 6.

5.2 Building Layered Monadic Interpreters

In this section, we introduce (1) a novel `over` combinator for lifting partial handlers to whole signatures, (2) a general characterization of the `trigger` combinator, which interplays with a novel `interp` combinator for building layered interpreters. In addition, we propose a new kind of event constraint which characterizes

isomorphisms between *sums of events*, where we use typeclass resolution to infer the correct type injections for the `over` and `trigger` combinators.

The interpreter from Figure 5.2 is unsatisfactory because of the need for manual annotations of `inr1` and `inl1`. To simplify this interpreter, we define a generic function that injects `handle_state` into an arbitrary signature containing `stateE`. There are two main challenges in defining this automatic injection.

First, injecting the `handle_state` handler induces auxiliary handlers, such as `pure_state`, that perform no action on the events: these auxiliary handlers must be inferred and applied to the uninterpreted remainder of the sum. Second, when the injected handler `handle_state` acts on a signature containing `stateE` (where the signature may contain other events), it needs to return the remainder of the signature. This is trivially achieved when hard-coding the shape of the signature as `stateE +' E`, but cannot be captured by the current inclusion constraint `stateE -< F`.

These challenges motivate respectively the definition of *triggerable monads* (Section 5.2.1) and the generalization of the inclusion of signature into a decomposition of signatures (Section 5.2.2). These two building blocks are sufficient to define the *automatic injection of handlers* addressing Problem 1 (Section 5.2.3). Problem 2 finds its resolution by the additional introduction of *Interpretable monads* (Section 5.2.4). While motivated by concrete problems, `over`, `interp`, and `trigger` also form a cohesive equational theory: Section 6.1 and 6.2 describe the equational properties of these combinators.

5.2.1 Triggerable Monads

Recall the `trigger` combinator from Section 5.1. It is defined as `trigger e := Vis e (fun x ⇒ x)`, capturing the idea of a "minimal" impure computation performing an uninterpreted event—in this case, specialized to the ITree monad. The `pure_state` function from Figure 5.2 mirrors this intent, but in the monad `stateT S (itree E)`; it additionally makes explicit that this minimal computation does not affect the state.

We capture this notion into a `Trigger` typeclass, corresponding to an action (event) having a specific monad `M` as its domain of action.

```
Class Trigger (E: Type → Type) (M: Type → Type) := trigger: E ⇝ M.
```

On the implementation side, it is worth mentioning that this `trigger` operator does not explicitly mention the monadic structure of `M`. This is inspired by the "unbundled" approach of Spitters and van der

Weegen [SvdW11], that proves beneficial for mathematical formalizations in type theory. All typeclasses in our framework use this unbundled style.

Naturally, `ITree.trigger`[20] is an instance for `itree E` (where `E` is of type `Type → Type`), and `pure_state` for `stateT S (itree E)`. However, it is possible to capture a broader class of instances at once, as we will see shortly.

Since we also want to reason about the structures we introduce, this new definition raises the question of the axiomatization of the `trigger` operation. In the case of ITrees, there are two characteristic equations supported by `trigger`. First, when seen as a handler, its interpretation is the identity, *i.e.*, that `∀ t, interp trigger t ≈ t`. Second, when seen as an interpreted computation, it coincides with the effect of the handler on the event, i.e. `∀ h e, interp h (trigger e) ≈ h e`. At this point, we lack the tools to generalize these equations; in particular, we would need to be able to interpret a computation in the monad of interest, rather than an ITree specifically. Therefore, we delay the question of axiomatizing these properties until Section 6.2.

5.2.2 Automatic Injection and Decomposition of Signatures

Let us temporarily set aside triggerable monads to turn our attention to the second issue: how to remove the hard-coded shape of the source event signature, yet reconcile that with a requirement that the target signature removes the handled event.

We achieve this by first enriching the typeclass responsible for expressing that a signature is a super-set of another. The current constraint, `E -< F`, simply requires an embedding of `E` into `F`. Instead, we introduce a typeclass that explicitly computes the complement to `E` in `F`.

The resulting relation is therefore three-placed: the constraint `Subevent E F G`, written `E +? G -< F`, provides an isomorphism between the types `E +' G` and `F`:

```
Class Subevent {E F G : Type → Type} : Type := { split : F ⇝ E +' G ;
                                                 merge : E +' G ⇝ F }.
```

From a resolution standpoint, one should think of it as taking as input the ambient signature `F`—from the return type of the computation being built—and the sub-signature `E`—from the concrete object manipulated, typically a triggered event or a handler—and inferring from this information the complement `G`.

[20] This style of colored text corresponds to Coq syntax

```
Instance Subevent_Base {A B}: A +? B -< A +' B.
Instance Subevent_refl {A} : A +? void1 -< A.
Instance Subevent_Sum_In {A B C D}
  `{A +? D -< B} : (C +' A) +? D -< C +' B.
Instance Subevent_Sum_Out {A B C D}
  `{A +? D -< B} : A +? C +' D -< C +' B.
Instance Subevent_commute {A B C}
  {Sub: A +? B -< C} : B +? A -< C.


    Instance Trigger_ITree_base {E} : Trigger E (itree E) := fun _ e ⇒ trigger e.
    Instance Trigger_ITree {E F G} `{E +? F -< G}: Trigger E (itree G) :=
      fun _ e ⇒ trigger (merge (inl1 e)).
    Instance Trigger_MonadT {E F G} `{E +? F -< G} {T : (Type → Type) → Type → Type}
      {T_MonadT: MonadT T} : Trigger E (T (itree G)) :=
        fun _ e ⇒ lift (trigger e).
```

Figure 5.4: The trigger typeclass

Instance inference gets more involved with Swierstra's injection, especially when combining several constraints. We characterize to this end the algebraic properties of this abstract sum operation `+?` as instances. Essentially, we state that it extends `+'`, admits `void1` as a unit, allows for injections of `+'` on either side of the decomposition, and commutes. We constrain the use of these instances — commutation in particular — to prevent the inference mechanism to diverge: we refer the interested reader to our formal development. Each of these instances come with a proof of isomorphism, thus guaranteeing soundness of the inference.

With this definition of `Subevent` in place, we can use it to generically define, once and for all, `Trigger` instances as shown in Figure 5.4. Events of type `E` can be triggered into an ITree either at the same signature, or an extension of it. Moreover, rather than painfully (and manually) introducing ad hoc instances such as `pure_state`, we can transport `Trigger` instances through arbitrary monad transformers. The `lift` operator is defined on all monad transformers `T : (Type → Type) → Type → Type`, where `lift` transforms a monad `M : Type → Type` into a monad `T M`.

5.2.3 Automatic Injection for Handlers

We now have all the necessary tools to properly solve Problem 1 by injecting handlers of a restricted signature into a larger ambient one.

Consider a handler `h : E ⇝ M` implementing a set of effects described by `E` into an arbitrary monad `M`.

```
Definition interp_state {E F} `{stateE +? E -< F} : itree F ⤳ stateT S (itree E) :=
  interp (over handle_state).

Definition interp_local {E1 E2 F} `{LocalE +? E1 -< F} `{FailureE +? E2 -< E1}
: itree F ⤳ stateT map (itree E1) := interp (over handle_local).
```

Figure 5.5: State interpreter and Vellvm's register map interpreter using `over`

The function `over h` transports `h` into an implementation of a larger signature `F` into the same monad:

```
Definition over {E F G M : Type → Type} `{E +? G -< F} `{Trigger G M}
  (h : E ⤳ M) : F ⤳ M :=
  fun _ f ⇒ match split f with
  | inl1 e ⇒ h e
  | inr1 g ⇒ trigger g
  end.
```

The function relies on the constraint `E +? G -< F` to know how to case analyze on a `F` event whether it corresponds to the embedding of an `E` event or not. In the former case, it simply calls its implementation `h`. In the latter case, it relies on the `Trigger G M` constraint to know how to embed this event into `M`.

Figure 5.5 illustrates the cleaned-up definitions for the ITree's state interpreter and Vellvm's register map implementation. The `interp_state` definition is straightforwardly simplified: no explicit extension of the handler is needed, and the return type uses the complement specified in the typeclass constraint. In the second case, notice that we can easily enforce that the source signature contains both `LocalE` and `FailureE`, while the target signature is only stripped of the former. As intended, these interpreters can consequently be used regardless of the structural position of the interpreted signature in the ambient context. The equational theory of `over` is discussed in Section 6.2.

5.2.4 Interpretable Monads

We now focus on the second obstacle to the compositional definition of layered monadic interpreters: interpreting monadic structures other than pure ITrees. When staging a stack of interpreters, we end up having to interpret from monads such as "stateT S (itree E)": this is the motivation for a notion of *interpretable monads*, which are layered monads that satisfy the structural properties for interpretation. *Interpretable monads* must encompass monads built from ITree through layers of interpretation: they should essentially be iterative monads that can trigger events. Furthermore, interpretable monads must interpret into interpretable monads, as we aim to stack layers of interpretation. The typeclass we need to provide

generic constructions and axiomatizations over the computational structure suitable to be part of a monadic interpreter more specifically generalizes the `ITree.interp` function.

```
Class Interp (IM T: (Type → Type) → Type → Type) (M : Type → Type) :=
  interp : ∀(E : Type → Type) (h: E ⇝ M), T (IM E) ⇝ T M.
```

Intuitively, an instance of this class expresses that a structure akin to ITrees can lift handlers into a structure `M`. The shape of the source structure is, however, further specified. At its base, it should contain a family of monads `IM` indexed by signatures—`itree` is one example. Intuitively, this minimal structure is the one upon which the implementation of the effects will be lifted. To compose cleanly, a monad transformer `T` is assumed on the source, and preserved into the target: previously introduced effects are left untouched.[21]

The `interp` function provided by the ITree library is a particular instance of this typeclass, where `IM` is `itree`, `T` is the identity transformer, and `M` is an arbitrary iterative monad.

With this definition, very generic instances can be provided to build layered interpreters compositionally. Following [JG09], we make explicit the higher-order functorial structure of monad transformers: they directly transport indexed functions via `hfmap`, as well as functions through any functor, per the operations shown below.

```
Class HFunctor (f : (Type → Type) → Type → Type) :=
  { ffmap : ∀A B g, Functor g → (A → B) → f g A → f g B;
    hfmap : ∀g h, (g ⇝ h) → (f g ⇝ f h) }.
```

This functorial structure is sufficient to ensure that monad transformers preserve the fact that a structure is a valid source of interpretation, as is captured by the following instance:

```
Instance stack_interp {T IM : (Type → Type) → Type → Type} {M : Type → Type}
           {HFunctor T} {IterativeMonad M} {Interp IM Id M} : Interp IM T M :=
  fun E h R t ⇒ hfmap (interp h) t.
```

Requiring in the `Interp` assumption the transformer parameter to be `Id` forces the stack to be built by adding new effects at the bottom, eliminating ambiguity when inferring types — alternate usages can be manually recovered. In practice, we work with a more specialized instance, fixing `IM` to be `itree`, to lighten up the unification problems arising when using these highly overlapping and ambiguous typeclasses. Combined with `interp` as a base instance, we can build interpreters from any structure built by applying monad transformers atop of the ITree monad, resolving Problem 2.

[21] Because of the 'unbundled' approach, the structural constraints (such as well-formedness properties of their operators) on this multi-parameter typeclass are not apparent here.

Through this section, we have introduced principled tools that clean up the definitions of layered stacks of interpreters. To do so, we have identified general structures whose particular instances were used in the ITree library: triggerable monads, decomposition of signatures, and interpretable indexed monads. What remains to be defined is the infrastructure needed to reason about these definitions. In the following section, we start introducing their equational theory.

Chapter 6

Reasoning About Layered Monadic Interpreters

6.1 A Composable Equational Theory for Monads

Monadic interpreters come with an alluring promise: equivalences or refinements of computations can be established equationally. Part of this reasoning naturally relies on the domain-specific algebraic laws that a given monad satisfies. But a significant structural equational theory—relevant to essentially all of the monads in our layered interpretations—is equally necessary. These theories can be painfully (re-)discovered and manually implemented for each monad, but that approach does not scale in practice.

To alleviate this problem, we provide through this Section a rich equational axiomatization of the monadic structures that arise from the construction of layered monadic interpreters. This theory both refines the one provided by ITrees, and generalizes it greatly, notably by axiomatizing the new constructions introduced in Section 5.2. We present in Section 6.1.4 and 6.2 how these theories can be cleanly transported by monad transformers and interpretation, lightening the burden put on a user when building their own layered interpreter.

More specifically, Section 6.1.1 introduces eqmR, the family of relations over monadic computations we consider—one can think of it as a generalization of `eutt`. Section 6.1.2 defines the *image* of a monadic computation, allowing for the definition of the enriched set of monadic laws we axiomatize and prove to hold for standard monads in Section 6.1.3. Finally, we describe in Section 6.1.4 the transport of these structure through monad transformers to ease once and for all the construction of the structures used in layered interpreters, before discussing cross-monad relations.

6.1.1 Equivalence and Relations between Monadic Computations

A monad only deserves its name if it satisfies the well-known three monad laws. The `ret` operation should be a unit to the left of the `bind` ($x \leftarrow \texttt{ret}\ \ a\ ;;\ k\ x = k\ a$) and to its right ($x \leftarrow ma\ ;;\ \texttt{ret}\ x = ma$). The `bind` operation should additionally be associative ($b \leftarrow (a \leftarrow ma\ \ ;;\ f\ a)\ \ ;;\ g\ b = a \leftarrow ma\ \ ;;\ (b \leftarrow f\ a\ \ ;;\ g\ b)$).

$$\frac{Reflexive\ R}{Reflexive\ (\approx_R)}\ \textsc{Refl} \qquad \frac{PER\ R}{PER\ (\approx_R)}\ \text{PER}$$

$$\frac{ma \approx_{R_1} mb \quad mb \approx_{R_2} mc}{ma \approx_{R_2 \circ R_1} mc}\ \textsc{RelComp}$$

$$\frac{}{eqmR(\dagger R) \simeq \dagger(eqmR\ R)}\ \textsc{Transpose}$$

$$\frac{ma \approx_{R_1} mb \quad R_1 \subseteq R_2}{ma \approx_{R_2} mb}\ \textsc{Mono}$$

$$\frac{ma \approx_R mb \qquad ma \approx_{R'} mb}{ma \approx_{R \wedge R'} mb}\ \textsc{Conj}$$

$$\frac{m_1 \approx_{RA \otimes RB} m_2}{\text{fmap}\ fst\ m_1 \approx_{RA} \text{fmap}\ fst\ m_2}\ \textsc{ProdFst}$$

$$\frac{m_1 \approx_{RA \otimes RB} m_2}{\text{fmap}\ snd\ m_1 \approx_{RA} \text{fmap}\ snd\ m_2}\ \textsc{ProdSnd}$$

$$\frac{\text{fmap}\ fst\ m_1 \approx_{RA} \text{fmap}\ fst\ m_2 \quad \text{fmap}\ snd\ m_1 \approx_{RB} \text{fmap}\ snd\ m_2}{m_1 \approx_{RA \otimes RB} m_2}\ \textsc{Prod}$$

$$\frac{m_1 \approx_{RA} m_2}{\text{fmap}\ inl\ m_1 \approx_{RA \oplus RB} \text{fmap}\ inl\ m_2}\ \textsc{SumL1}$$

$$\frac{m_1 \approx_{RB} m_2}{\text{fmap}\ inr\ m_1 \approx_{RA \oplus RB} \text{fmap}\ inr\ m_2}\ \textsc{SumR1}$$

$$\frac{\begin{array}{c} m_1 \quad \approx_{RA \oplus RB} \quad m_2 \\ \forall\ a_1\ a_2,\ RA\ a_1\ a_2 \to RC\ (f_1\ a_1)\ (f_2\ a_2) \\ \forall\ b_1\ b_2,\ RB\ b_1\ b_2 \to RC\ (g_1\ b_1)\ (g_2\ b_2) \end{array}}{\text{fmap}\ (\text{case}\ f_1\ g_1)\ m_1 \approx_{RC} \text{fmap}\ (\text{case}\ f_2\ g_2)\ m_2}\ \textsc{Sum}$$

Figure 6.1: OK eqmR Laws (Well-formedness Laws of EqmR)

This statement is however too naive, hiding a major difficulty: these equations have no hope to hold up-to equality. In the case of a coinductive structure such as ITrees, for instance, even eta-laws do not hold with respect to `eq`, Coq's equality. One therefore needs to switch to a different notion of equivalence, namely, (strong) bisimulation, which is defined the the ITree library as `eq_itree`. While `eq_itree` can be used to prove the monad laws, it is *still* too strong for some iterative laws and interpretation laws; for these, we need weak bisimilarity, *i.e.*, `eutt`. The conclusion is not surprising: monads should come equipped with their own notion of equivalence of computations.

But monadic interpreters are used to prove more than program equivalence. For instance, the original ITree paper [XZH+20] establishes the correctness of a compiler. This is achieved by parameterizing `eutt` by a relation on computed values: `eutt` specifies that weak bisimilarity is the prime notion to compare computations, and lifts an arbitrary relation on the computed values in the process. This extension not only allows for relating heterogeneous computations, but provides the foundations for establishing bisimilarity results following a relational program logic style [BKBH09].

We therefore work with monads `M` equipped with an "equality of monads up to R", a family of relations: `eqmR A B (R : A → B → Prop) : M A → M B → Prop`. We write $\approx_R$ in lieu of `eqmR R`.

Figure 6.1 axiomatizes the required behavior for eqmR. Equivalences should be lifted into equivalences, ensuring, in particular, that $\approx_{eq}$ — written $\approx$ in the following — behaves as a suitable tightest notion of equality of computations. We derive this transport from the slightly stronger request that partial equivalence

relations (PERs) and reflexivity be preserved independently: PERs are used to define the notion of an *image* as introduced in Section 6.1.2. The `RelComp` and `Transpose` rules express the standard heterogeneous extensions of transitivity and symmetry (we write $\dagger R$ for the transposition of R and $\subseteq$ for inclusion of relations).

As mentioned, eqmR is meant to be thought of as the basis of a relational program logic. The indexed relation should therefore be monotone, providing a weakening rule, and ensuring compatibility with the equivalence of relations. From `Mono`, one trivially derives the usual disjunction rule, but the `Conj` rule must be additionally required.

Finally, anticipating Section 6.1.4, we wish to transport our equational theories via monad transformers. Since examples such as the state and error transformers expand the return type of the computation with respectively a product and a coproduct, eqmR should respect those too—the right column in Figure 6.1 specifies the necessary introduction and elimination rules.

We conclude this section by illustrating valid instances of eqmR over concrete monads.

Example 1. ITree.

At any signature `E`, `itree` E is known to be a monad. But more specifically, we prove that the strong (`eq_itree`) and weak bisimulation (`eutt`) are instances of eqmR and satisfy its laws.

Example 2. State.

Computations in the state monad are state-passing functions over a domain of states S: $(\texttt{stateM}_S\ X \triangleq S \to S * X)$. We consider the standard operations: pure computations leave the state untouched while sequencing threads the states.

$$\texttt{ret}^{St}\ v \triangleq \lambda\, s \Rightarrow (s, v) \qquad\qquad \texttt{bind}^{St}\ m\ k \triangleq \lambda\, s \Rightarrow \texttt{let}\ (s', a') := m\, s\ \texttt{in}\ f\ a'\ s'$$

Since computations in the state monad are functions, the family of relations we consider relaxes equivalence to functional extensionality, and further lifts the relation overt the returned values. The relation of state considered here is equality, but could be additionally relaxed to other equivalences.

$$m_a \approx_R m_b \triangleq \forall s,\ R\ (snd\ (m_a\ s))\ (snd\ (m_b\ s)) \wedge (fst\ (m_a\ s)) = (fst\ (m_b\ s))$$

We prove that this definition of eqmR satisfies the required laws.

Example 3. Error.

Potentially failing computations over a type of errors E can be implemented as a sum type $\texttt{errorM}_E\ R \triangleq E + R$ and equipped with the usual sequencing passing by valid computed value and propagating erroneous states, as depicted to the left in the following:

$$
\begin{array}{rl}
\texttt{ret}^{Err}\ v & \triangleq \mathsf{inr}\ v \\
\texttt{bind}^{Err}\ m\ k & \triangleq \mathsf{match}\ m\ \mathsf{with} \\
& \quad \mid \mathsf{inl}\ e \Rightarrow \mathsf{inl}\ e \\
& \quad \mid \mathsf{inr}\ v \Rightarrow k\ v \\
& \quad \mathsf{end}
\end{array}
\qquad\qquad
\begin{array}{rl}
x \approx_R y & \triangleq \mathsf{match}\ x, y\ \mathsf{with} \\
& \quad \mid \mathsf{inl}\ \_\ \mathsf{inl}\ \_ \Rightarrow \top \\
& \quad \mid \mathsf{inr}\ v_1, \mathsf{inr}\ v_2 \Rightarrow R\ v_1\ v_2 \\
& \quad \mid \_, \_ \Rightarrow \bot \\
& \quad \mathsf{end}
\end{array}
$$

To the right is an eqmR satisfying the required laws: it lifts the relation over valid related results, and accepts co-failure disregarding the value of the error.

Example 4. Nondeterminism.

Nondeterministic computations can be represented as sets of outcomes using `Prop`: $\texttt{prop}\ R \triangleq R \to \texttt{Prop}$. This propositional account of nondeterminism gives up its computational content, but is in exchange flexible to manipulate, allowing for modelling nondeterminism over infinite sets, as well as for specifying a computation.

The pure computation is a deterministic one, so it builds the singleton set: $\texttt{ret}^{ND} \triangleq \lambda\ a' \Rightarrow a = a'$. The bind should flatten into a single set all possible outcomes for each nondeterministically reachable branch of the computation, *i.e.*, $\texttt{bind}^{ND}\ m\ k \triangleq \exists\ a,\ m\ a \wedge k\ a\ b$. A notion of bijection up-to relation defines an eqmR satisfying *most of* the laws :

$$ma \approx_R mb \triangleq (\forall a, ma\ a \to \exists b, mb\ b \wedge R\ a\ b) \wedge (\forall b, mb\ b \to \exists a, ma\ a \wedge R\ a\ b)$$

We highlight the situation of `prop` due to a use of a related structure in Vellvm, as depicted at the bottom of Figure 5.1. However, it is instructive to notice that it does not quite satisfy the interface: the Conj rule and the rules related to the product of relations (see Figure 6.1) are invalid.

6.1.2 Image of Monadic Computations

The proposition $m_a \approx_R m_b$ intuitively asserts that m_a and m_b are compatible computations—weakly bisimilar in the case of ITrees, for instance—and that the relation R is a valid relational postcondition over returned values. The axioms from Figure 6.1 provide the basis to justify this interpretation of R as a postcondition. To conduct relational reasoning, one needs additional structural rules that compositionally relate computations built from combinators. For instance, the ITree library provides an equation relating sequences of computations: `ma` $\approx_S$ `mb` $\rightarrow$ `(∀ x y, S x y →` `ka x` $\approx_R$ `kb y) →` `ma >>= ka` $\approx_R$ `mb >>= kb`. This rule mirrors the familiar Hoare-style rule for sequence by quantifying existentially over `S`, the intermediate postcondition.

Monadic computations expressed as sequences have a more subtle structure, though. A typical monadic computation might return only a strict subset of the values of its return type, `R`, while its continuation is always defined over all of `R`. By way of illustration, consider the monad `itree ChoiceE` where the signature `ChoiceE` provides an event, `choice : ChoiceE bool`, encoding a binary branching indexed by a boolean. The computation `c ::= b ← trigger choice;; if b then ret 1 else ret 0` triggers this external choice event, and converts the returned boolean into a natural number. Because `c` has type `itree ChoiceE nat`, any continuation `k` bound to `c` will be indexed over *all* natural numbers, but the only relevant branches should be `(k 0)` and `(k 1)`. The proof rule above may lead to spurious proof obligations. While `S` can naturally always be taken sufficiently tight to rule out these spurious cases, there should be a systematic way to strengthen it.

Following this intuition, we introduce the notion of the *image* of a monadic computation: interpreting the diagonal of eqmR as a unary logic, we want to define uniformly a notion akin to a strongest postcondition of a computation. For an ITree, the *image* is, intuitively, the set of values that appear at its leaves, so the image of `c` will be the set $\{0, 1\}$.

6.1.2.1 Image: A Semantic Characterization

We wish to capture abstractly the set of values possibly returned by a monadic computation. Thus we seek to associate to each computation `m : M A` a predicate `image m : A → Prop` over its return type. Concretely, for an ITree `t`, `image t` should be the set of leaves of `t`; for a nondeterministic monad, the image should

be the set of values it may return. The challenge is defining the image *without* referring to the particular structure of a given monad, a necessity for incorporating it into general structural laws.

A first intuition on this path is to consider the diagonal of eqmR: suppose `m: M R` is a computation in some monad `M` and `R` is a relation at which `m` self-relates: $\texttt{m} \approx_R \texttt{m}$. Since R is a relational postcondition, all pairs of returned values should belong to `R`, and the square of the `image m`, *i.e.*, $\{ (\texttt{a}, \texttt{a}) \mid \texttt{image m a} \}$ should be included in `R`.

Following this idea, one could be tempted to carve out the extra junk in `R` by defining the image as the diagonal of the *intersection* of all relations at which `m` self relates:

$$\texttt{image}\ m\ v \stackrel{?}{=} \forall\ R, m \approx_R m \to R\ v\ v$$

Interestingly enough, this first attempt turns out to be too naïve when considering nondeterminism; it leads, in general, to a strict subset of the image we seek to capture. To see why, we focus our attention to the case of `propM`, with eqmR defined as in Example 4.

Let `(m : propM bool)` be the computation that nondeterministically returns a boolean, that is, `m = fun x` $\Rightarrow$ `x = true`$\vee$ `x = false`. We naturally expect the image of `m` to contain both booleans as well. However, by taking for relation `R` = `{(true,false);(false,true)}`, we easily see that $\texttt{m} \approx_R \texttt{m}$ holds despite `R`'s diagonal being empty!

Intuitively, considering all relations is inadequate: we should only consider those whose diagonal contains the elements it relates. To look on the side of equivalences would however be too drastic: all reflexive relations have their diagonal coincide with the whole return type. In particular, the ITree representing a silently diverging computation self-relates at all postconditions, we would fail to ascribe an empty image to it.

The right intuition echoes the idea of modeling the codomain of partial functions as *Partial Equivalence Relations* (PERs). PERs at which self-relation is possible always contain in their diagonal a superset of the image we seek to define, without being forced to contain the whole type. The image is precisely the diagonal of the *smallest* PER at which the computation self-relates:

$$\texttt{imageH}\ m\ a_1\ a_2 \triangleq \forall\ R, \mathit{PER}\ R \to m \approx_R m \to R\ a_1\ a_2 \qquad \texttt{image}\ m\ a \triangleq \texttt{imageH}\ m\ a\ a$$

We write $a \in m$ as a short-hand for `image` m a.

6.1.2.2 Image for `itree`: A Concrete Characterization

The semantic characterization of the image of a monadic computation only relies on eqmR, and can be leveraged to axiomatize the desired theory. However, this definition gives few reasoning principles. Therefore, we prove that in the case of the ITree monad, it coincides with the concrete original intuition: a predicate collecting the reachable leaves. This predicate is defined inductively over the structure of the tree, existentially collecting all branches:

```
Inductive Leaves {E} {A: Type} (a: A) : itree E A → Prop :=
| LeavesRet: ∀t, t ≈ Ret a → Leaves a t
| LeavesTau: ∀t u, t ≈ Tau u → Leaves a u → Leaves a t
| LeavesVis: ∀{X} (e: E X) (x: X) t k, t ≈ Vis e k → Leaves a (k x) → Leaves a t.
```

The structural and semantic definitions are proved equivalent, justifying the abstract definition, and providing inductive reasoning to establish membership to the image.

Lemma 4. $a \in ma \iff$ `Leaves` a ma

6.1.2.3 The Case of `stateM`

We prove that the image of a $\texttt{stateM}_S$ computation captures exactly the set of values that can be returned for *some* initial state, regardless of the final state.

Lemma 5. $v \in m \iff \exists\, s_i\ s_f, (m\ s_i) = (s_f, v)$

Notice the existential quantification on both the initial and final state.

Anticipating the axiomatization introduced in the following section, we consider how the image predicate and the bind construct interact. Following the analogy of a strongest postcondition, it could be hoped that the following rule universally hold:

$$\frac{u \in m \qquad v \in (k\ u)}{v \in (m\ ;;\ k)}\ \textsc{ImageBind}$$

However, this would be misunderstanding what the image expresses: while it universally captures the tightest postcondition over the set of returned value, it cannot do so w.r.t. the effects the computation

perform. Since the reachability of branches of the postcondition may depend on the history of effects, it is therefore expected that it will not behave as uniformly w.r.t. `bind`.

To illustrate more concretely this intuition, we build over this state monad instance a counter example to ImageBind. Fixing the state to `bool`, consider the following computation that returns the initial state as value, but always sets the final state to `true`.

```
m ::= fun b ⇒ if b then (true, true) else (true, false)
```

The image of `m` contains both booleans since either can be found as returned value for a certain initial state. In particular, `false ∈ m`. Now consider the continuation `k ::= fun v b ⇒ (v,b)` updating the state with its argument and returning the previous state. One can think of this computation as a balanced tree with four leaves, where each subtree admits both booleans in its image: in particular, `false ∈ k false`. The branches carrying `false` are, however, reachable only from a state set at `false`: since `m` does not return such state, they are unreachable branches. To sum up, `false ∈ m`, `false ∈ (k false)`, but yet `false ∉ (m >>= k)`, contradicting ImageBind.

Intuitively, the image characterizes what values a monadic computation might possibly return, but does so in a way that is parametric in the monad definition itself. As illustrated by this example, it does not account for information internalized by the monad, such as unreachable states. Nevertheless, the refined (if still approximate) reasoning enabled by the image is an essential ingredient for defining precise reasoning principles in the equational theory.

6.1.2.4 Image for `errorM`

As can be expected, the image over the error monad is much simpler to capture: it is either empty if the computation fails, or the singleton of the computed value otherwise.

Lemma 6. $a \in ma \iff ma = \mathsf{inr}\ a$.

6.1.2.5 Image for `prop`

For nondeterminism, the computation itself coincides with the image:

$$\frac{}{x \leftarrow \texttt{ret}\ a\ ;;\ f\ x \approx f\ a}\ \textsc{RetL} \qquad \frac{}{x \leftarrow ma\ ;;\ \texttt{ret}\ x \approx ma}\ \textsc{RetR} \qquad \frac{R_A\ a_1\ a_2}{\texttt{ret}\ a_1 \approx_{R_A} \texttt{ret}\ a_2}\ \textsc{Ret}$$

$$\frac{}{(a \leftarrow ma\ ;;\ b \leftarrow f\ a)\ ;;\ g\ b \approx a \leftarrow ma\ ;;\ (b \leftarrow f\ a\ ;;\ g\ b)}\ \textsc{BindAssoc}$$

$$\frac{ma_1 \approx_{R_A} ma_2 \qquad \forall a_1,\ a_2,\ R_A\ a_1\ a_2 \to k_1\ a_1 \approx_{R_B} k_2\ a_2}{x \leftarrow ma_1\ ;;\ k_1\ x \approx_{R_B} x \leftarrow ma_2\ ;;\ k_2\ x}\ \textsc{Bind} \qquad \frac{}{ma \approx_{(\texttt{imageH}\ ma)} ma}\ \textsc{ImageSelf}$$

$$\frac{ma_1 \approx_{R_A} ma_2 \qquad a_1 \in ma_1}{\exists\ a_2,\ R_A\ a_1\ a_2 \wedge a_2 \in ma_2}\ \textsc{ImageL} \qquad \frac{ma_1 \approx_{R_A} ma_2 \qquad a_2 \in ma_2}{\exists\ a_1,\ R_A\ a_1\ a_2 \wedge a_1 \in ma_1}\ \textsc{ImageR}$$

Figure 6.2: EqmRMonad Laws

Lemma 7. $a \in ma \iff ma\ a$.

6.1.3 Beyond Monadic Laws

We have all the tools required to define a first minimal axiomatization of the monads we accept to consider. This interface is described on Figure 6.2 and contains three kinds of properties. As expected, the three traditional monad laws are still required, but are expressed up-to $\approx$.

Next are three rules constraining properties of the image of a monadic computation. Rules ImageL and ImageR systematically link the images of two computations that can be proved to be related by eqmR: any point in one of the images can be related to the other via the postcondition. Rule ImageSelf ensures part of the intuition we started from: the image should capture all possibly returned values, it should therefore itself be a valid postcondition when self-relating. As illustrated in Section 6.1.2.3, the ImageBind rule should not be an axiom.

Finally come the two proof rules enriching our relational logic. They directly mirror the ones used in the ITree standard library, but generalized to work over arbitrary monads: the Ret rule describes how to relate pure computation; the Bind rule how to relate two sequences. We have mentioned at the beginning of this Section that the Bind rule puts all the stress of constraining the image of the computations being related on the choice of R_A. Using the ImageSelf and RelComp rules, we can now prove abstractly the following principle, systematically enriching R_A by intersecting it with both images.

$$\frac{ma_1 \approx_{R_A} ma_2 \qquad \forall a_1,\ a_2,\ a_1 \in ma_1 \to\ a_2 \in ma_2 \to\ R_A\ a_1\ a_2 \to k_1\ a_1 \approx_{R_B} k_2\ a_2}{x \leftarrow ma_1\ ;;\ k_1\ x \approx_{R_B} x \leftarrow ma_2\ ;;\ k_2\ x}\ \textsc{CloBindGen}$$

We illustrate the adequacy of this axiomatization by providing instances.

$$\frac{\texttt{ret}\ a_1 \approx_{RA} \texttt{ret}\ a_2}{RA\ a_1\ a_2}\ \textsc{RetInv} \qquad \frac{\mathsf{fmap}\ f\ ma \approx_R \mathsf{fmap}\ g\ mb}{ma \approx_{\lambda a\, b, R\ (f\ a)\ (g\ b)}\ mb.}\ \textsc{FmapInv} \qquad \frac{b \in x \leftarrow ma\ ;;\ k\ x}{\exists a, a \in ma \wedge b \in k\ a}\ \textsc{BindImageInv}$$

Figure 6.3: EqmRMonadInverses Laws

Lemma 8. The `itree`, $\texttt{stateM}_S$, $\texttt{errorM}_E$ and `prop` monads satisfy the eqmR laws.

Furthermore, we capture the fact that the sequencing operation over ITrees depends exclusively on the computed value:

Lemma 9. The `itree` monad additionally satisfies the ImageBind rule.

6.1.3.1 Inversion Laws

Figure 6.2 provides the core forward reasoning rules associated to eqmR, useful for establishing such relations. Reciprocally, backward reasoning is useful to derive information from an established relation between two computations. We capture in Figure 6.3 the inversion laws that hold true for all our structures of interest.

The two first rules derive information from an hypothesis that two computations of a certain shape are related. The RetInv rule ensures that related pure values are in the postcondition while the FmapInv rule expresses that when `fmap` computations are related, the mapped functions can be pushed down the postcondition.

Interestingly, while we have seen that forward compatibility of the image with `bind` is invalid in monads as common as $\texttt{stateM}_S$, backward compatibility is always valid. If we know that a value is in the image of a `bind`, BindInv decomposes this hypothesis by exhibiting a branch of the continuation whose image contains this same value. And indeed, we show:

Lemma 10. The `itree`, $\texttt{stateM}_S$, $\texttt{errorM}_E$ and `prop` monads satisfy the eqmR inversion laws.

Furthermore, in any monad satisfying both eqmR interfaces, the image of a pure computation is exactly the singleton:

Lemma 11. Over any eqmR monad satisfying all well-formedness laws, $a \in \texttt{ret}\ b \leftrightarrow a = b$.

We have carefully defined the semantic definition of eqmR as the smallest PER at which a computation self relates, as opposed to an arbitrary relation. This subtlety has been motivated by the case of the prop monad: over this instance, one can self relate at a relation whose diagonal does not contain the image. We show that this restriction can be dropped for the three other example considered:

Lemma 12. Over the `itree`, $\texttt{stateM}_S$ and $\texttt{errorM}_E$ monads, the diagonal of any self-relating postcondition contains the image:

$$\frac{ma \approx_R ma \quad a \in ma}{R\ a\ a}\ \textsc{BindImageInv}$$

We have not considered any backward reasoning principle for related `bind` computations. Indeed, no such rule hold in general, but the notion of image allows us to provide one in specific cases:

Lemma 13. Over the `itree` monad, the following rule, inverting relations between `bind`s sharing a common prefix, holds true:

$$\frac{t \gg= k_1 \approx_R t \gg= k_2 \quad r \in t}{k_1\ r \approx_R k_2\ r}\ \textsc{BindInv}$$

Such an inversion principle is in particular crucial when assuming an invariant of a computation expressed using the diagonal of eqmR.

6.1.4 Transporting eqmR via Monad Transformers

We have specified a minimal equational theory that any target domain for a monadic interpreter must satisfy, and illustrated that it holds on specific monads. These domains are, however, typically built by stacking successive monad transformers atop of a base triggerable monad—ITrees typically—as echoed by the structure of Vellvm's stack on Figure 5.1. In the absence of a clean abstract interface as the one we contribute here, users had no choice but to manually re-establish such a theory for each structure they consider. To alleviate this painful work, we provide the tools to systematically transport the eqmR structure and its theory via appropriate monad transformers. We spell out all the requirement through an additional series of typeclasses.

On the operational side first, a monad transformer $MT : (\mathsf{Type} \to \mathsf{Type}) \to \mathsf{Type} \to \mathsf{Type}$ must provide the traditional `lift` function embedding computations of an arbitrary monad into the transformed

$$\frac{R\ a\ b}{\texttt{lift}\ (\texttt{ret}\ a) \approx_R \texttt{ret}\ b}\ \textsc{LiftRet} \qquad \frac{ma \approx_{RA} ma' \qquad \forall a\ a', RA\ a\ a' \to k\ a \approx_{RB} k'\ a'}{\texttt{lift}\ (x \leftarrow ma;;k\ x) \approx_{RB} x \leftarrow \texttt{lift}\ ma';;\texttt{lift}\ (k'\ x)}\ \textsc{LiftBind}$$

$$\frac{ma \approx_R mb}{\texttt{lift}\ ma \approx_R \texttt{lift}\ mb}\ \textsc{LiftEqmR}$$

Figure 6.4: Monad morphism laws

$$\frac{\mathsf{EqmROK}\ M}{\mathsf{EqmROK}\ (\texttt{lift}\ M)}\ \mathrm{OK} \qquad \frac{\mathsf{EqmRMonad}\ M}{\mathsf{EqmRMonad}\ (\texttt{lift}\ M)}\ \textsc{OKMon} \qquad \frac{\mathsf{EqmRMonadInverses}\ M}{\mathsf{EqmRMonadInverses}\ (\texttt{lift}\ M)}\ \textsc{OKInv}$$

Figure 6.5: Monad transformer well-formedness conditions

structures:

$$\texttt{lift} : \forall (M : \mathsf{Type} \to \mathsf{Type})(A : \mathsf{Type}), M\ A \to MT\ M\ A$$

For any monad M, `lift` M must define a monad morphism, *i.e.*, an indexed function commuting with the `ret` *a*nd `bind` operations. While standard, these two laws, depicted on the upper part of Figure 6.4[22], must be stated with respect to eqmR in our setup. Furthermore, `lift` must itself respect eqmR, which we capture in the LiftEqmR rule. Monad transformers must additionally construct valid monads: lifted monads should satisfy the monad laws. Since we request a richer minimal equational theory of our monads, we spell out their preservation (Figure 6.5): the well-formedness of eqmR, its forward rules and its backward rules should all be preserved.

Example 5. `stateT` . The state monad can be generalized into an appropriate transformer $\texttt{stateT}_S\ M\ A \triangleq S \to M\ (S \times A)$. The return and bind definitions are standard, mirroring the ones from $\texttt{stateM}_S$, but leveraging the underlying monadic operations: $ret\ v \triangleq \lambda\ s \Rightarrow \texttt{ret}\ (v, s)$ and $bind\ c\ k \triangleq \lambda\ s \Rightarrow (v, s) \leftarrow c\ s\ ;;\ k\ v\ s$. The lift operator relies on the underlying fmap to reinject the unchanged state: $\texttt{lift}\ c \triangleq \lambda\ s \Rightarrow \mathsf{fmap}\ (\lambda\ x \Rightarrow (s, x))\ c$.

We parameterize the definition of eqmR at any monad transformed by the state transform by a relation on states R_{st}. The relation on computations is then obtained by lifting pointwise R_{st} and a relation on values through the underlying eqmR— it is reminiscent of Goubault-Larrecq's characterization [GLN08] of the

[22]We omit the implicit argument M in this Figure, writing `lift` in lieu of `lift` M.

state transformer logical relation, and Maillard's relational Dijkstra monads [MHRVM20]:

$$ma \approx_R mb \triangleq \forall s_1\ s_2,\ (s_1, s_2) \in R_{St} \to ma\ s_1 \approx_{R_{st} \otimes R} mb\ s_2$$

where $R \otimes R'$ is defined as $\lambda\ x\ y \Rightarrow R\ x\ y \wedge R'\ x\ y$. Assuming related input states, eqmR unfolds the definition of the state monad to see it explicitly as a computation in the underlying monad over the product type, and enforces both the state relation on output state and the relation on values by taking as a postcondition the product of both relations.

The image admits a similar characterisation as in the case of $\texttt{stateM}_S$, but expressed in terms of the underlying notion of image:

Lemma 14. $a \in ma \iff \exists\ s\ s',\ (ma\ s) \in (s',\ a)$

We prove that $\texttt{stateT}_S$ transports all interfaces, providing our equational theory for free for structures such as the ones introduced in the first three layers of Figure 5.1 or in both languages of the compiler described in the original ITree paper [XZH+20].

Example 6. `errorT` . Similarly, we support the standard generalization of `errorM` to a monad transformer: $\texttt{errorT}_E\ M\ A \triangleq M(E + A)$. The return and bind operators, omitted here, are standard. The lift simply injects the result of the underlying computation into a successful one: $\mathsf{lift} \triangleq \lambda\ ma \Rightarrow \mathsf{fmap}\ \mathsf{inr}\ ma$. Finally, eqmR is parameterized by an arbitrary relation over errors and feeds the coproduct of both relations to the underlying eqmR: $ma \approx_R mb \triangleq ma \approx_{R_{exn} \oplus R} mb$.

We prove that $\texttt{errorT}_E$ preserves all interfaces. The characterization of the image still holds: they are the successful elements of the underlying image.

Lemma 15. $a \in ma \iff (\texttt{inr}\ a \in ma)$

6.1.5 Relating Computations across Distinct Monads

For clarity of exposition, we have presented eqmR as a heterogeneous relation over the return type, but assuming the same monad on each side, which is the most direct analog to the ITree library's notion of `eutt`. However, in many situations—*e.g.*, when expressing the correctness of a pass of compilation, such as for the IMP-to-ASM compiler (see Section 7)—we need to work across languages and relate computations in distinct

$$\frac{}{\texttt{interp}\ h\ (\texttt{ret}\ x) \approx \texttt{ret}\ x}\ \textsc{InterpRet} \qquad \frac{}{\texttt{interp}\ h\ (x \leftarrow t\ ;;\ k\ x) \approx x \leftarrow \texttt{interp}\ h\ t;;\texttt{interp}\ h\ (k\ x)}\ \textsc{InterpBind}$$

$$\frac{}{\texttt{interp}\ h_E\ (\texttt{trigger}\ e_E) \approx \texttt{lift}\ (h_E\ e_E)}\ \textsc{InterpTrigger} \qquad \frac{\forall\ j,\ \texttt{interp}\ h\ (f\ j) \approx f'\ j}{\texttt{interp}\ h\ (\texttt{iter}\ f\ i) \approx \texttt{iter}\ f'\ i}\ \textsc{InterpIter}$$

$$\frac{F +? G \prec H \qquad E +? H \prec I}{\texttt{interp}_I\ (\texttt{over}\ h_E)\ (\texttt{trigger}\ e_F) \approx \texttt{lift}\ (\texttt{trigger}\ e_F)}\ \textsc{IgnoreTrigger}$$

Figure 6.6: Interpretation Laws (we write h_E for a handler of type $E \rightsquigarrow M$ and e_E for an event of E)

monads. We therefore also provide in our formal development a more general notion of family of relations, heqmR, parameterized by two monads M and N, and lifting relations at return types (`A → B → Prop`) to relations at computation-level across monads (`M A → N B → Prop`). Such relations are typically specific to the proof at hand: each monad still comes with its own eqmR that the cross-monad relation must be proved to respect.

6.2 Layering EqmR with Interpreters

In the previous section, we have discussed how to build equational theories for monads and monad transformers, where we exposed several semantic characterizations such as element inclusion (using `image`) and equational laws that certain monads satisfy.

How does this relate to building layered interpreters? The monadic equational framework is the basis for expressing structural properties for interpretation. When we layer the `interp` combinator (from Section 5.2), we will also like certain structural properties to hold at each layer of interpretation. For instance, `interp` should respect monadic operators such as `ret` and `bind`, and interact well with *iteration*. Now, given an interpretable monad (see Section 5.2.4), equipped with an appropriate instance of eqmR, we can state what laws the `trigger`, `over`, and `interp` functions should obey. These laws are structural properties for the `Interp` typeclass and are shown in Figure 6.6.

The InterpRet and InterpBind laws say that `interp h` is a monad homomorphism.

The InterpTrigger rule applies when the event being triggered belongs to the signature handled by the handler—it simply says that the interpretation (morally) *is* the result of the handler applied to the event. The `lift` on the right-hand-side is the one from the functor associated with the interpretable monad, and it

coerces the handler's output to the right form (see the type of `interp` in Section 5.2.4, and the definition of `lift` in Section 6.1.4). Note that the handler might itself contain `over`, which means that this rule can still apply for a function that injects a handler for a program with a larger signature.

IgnoreTrigger covers the case when the handler cannot act on the triggered event. For example, for `interp handle_mem (trigger Print)`, where `handle_mem` is a handler for memory operations, and `Print` is the event for I/O. `Print` cannot be handled by `handle_mem`, and thus is propagated by re-triggering the event in the target monad, in this case, as `lift (trigger Print)`.

The last rule, InterpIter formalizes the interaction between iteration and interpretation. It says that if the interpreter respects the loop body for every iteration j, then it commutes with `iter`. This provides a kind of lock-step simulation principle that is useful in practice when reasoning about the equivalence of two computations defined by iteration.

6.2.1 Higher Order Functors Lift Structural Properties : Interp Laws for any Stack

The key contribution in our framework is that these *interpretation laws* about monads need to be proven only once and for all, regardless of the stack of interpretation. Specifically, we have proved that the laws in Figure 6.6, which are specified via a typeclass in Coq, hold for ITrees as a base instance, with its standard definitions of `interp`, `iter`, *etc.*. To account for layered interpretations, we then define a set of well-formedness conditions on the monads and monad transformers, which let us *derive* further instances of the interpretation laws by applying monad transformers.

Recall the definition of stacking interpretation from Section 5.2.4—it derives instances of `Interp`.

```
Instance stack_interp {T IM : (Type → Type) → Type → Type} {M : Type → Type}
           {HFunctor T} {IterativeMonad M} {Interp IM Id M} : Interp IM T M :=
  fun E h R t ⇒ hfmap (interp h) t.
```

One constraint imposed by this definition is that `M` is an `IterativeMonad`, a property that must be lifted to `T M` for a correct interpretation to exist. The other key well-formedness conditions express the functorial properties of the higher-order functor, `hfmap`, used in this definition. Such structural properties of higher-order functors, and their use in nesting monadic types, is known in the literature (see [JG09]); and we adapt those definitions for use in our setting. Select higher-order functor laws are presented in Figure 6.7, which shows that `hfmap` transports identity, commutes with composition, and preserves

$$\frac{}{\texttt{hfmap}\ (\lambda x \Rightarrow x)t \approx t}\ \textsc{HfmapId} \qquad \frac{}{\texttt{hfmap}\ (f_1 \ggg f_2)\ t \approx (\texttt{hfmap}\ f_1 \ggg \texttt{hfmap}\ f_2)\ t}\ \textsc{HfmapComp}$$

$$\frac{\mathsf{MonadMorphism}f}{\mathsf{MonadMorphism}(\texttt{hfmap}f)}\ \textsc{HfmapNat}$$

Figure 6.7: Select HFunctor Laws

$$\frac{\mathsf{MonadT}\ S \qquad \mathsf{MonadT}\ T}{\mathsf{MonadT}\ (S \ggg T)}\ \textsc{ComposeMonadT} \qquad \frac{\begin{array}{c}\mathsf{IterativeMonadT}\ S\\ \mathsf{IterativeMonadT}\ T\end{array}}{\mathsf{IterativeMonadT}\ (S \ggg T)}\ \textsc{ComposeIterativeMonadT}$$

$$\frac{\mathsf{HFunctor}\ S \quad \mathsf{HFunctor}\ T}{\mathsf{HFunctor}\ (S \ggg T)}\ \textsc{ComposeHFunctor} \qquad \frac{\begin{array}{c}\mathsf{IterativeMonadTLaws}\ S\\ \mathsf{IterativeMonadTLaws}\ T\end{array}}{\mathsf{IterativeMonadTLaws}\ (S \ggg T)}\ \textsc{ComposeIterativeMonadTLaws}$$

$$\frac{\begin{array}{c}\mathsf{HFunctorLaws}\ S\\ \mathsf{HFunctorLaws}\ T\end{array}}{\mathsf{HFunctorLaws}\ (S \ggg T)}\ \textsc{ComposeHFunctorLaws}$$

Figure 6.8: Composable Structures and Laws

monad morphisms. These are unsurprising properties for higher-order functors, but they are useful in our setting when lifting `ret`, `bind`, `iter`, and `lift` combinators. The remaining requirements follow from general facts about compositionality: the function composition of monad transformers, iterative monad transformers, and higher-order functors preserve the operations and well-formedness properties, as shown in Figure 6.8, where $\ggg$ is function composition. The structural properties MonadT, IterativeMonadT, and so on, correspond to the typeclasses that capture the structures and well-formedness laws. The complete set of hfmap-related laws and the details of these typeclasses is included in the Coq development.

Although there is a fair amount of effort needed to instantiate these well-formedness requirements for a given monad transformer, that work needs to be done only once, after which we can build interpretable monads by applying the transformer. We have shown that the interpretation laws, iterative monad laws, and higher-order functor laws hold for the identity monad transformer, state monad transformer, and error monad transformer. These form a useful basis for building language semantics; we expect that additional monad transformers could also be verified to satisfy these requirements, but leave that to future work.

Chapter 7

EqmR in Practice : Implementation and Case Study

This section surveys our Coq formalization, briefly describing the typeclasses necessary for packaging the equational theory the custom tactics necessary for implicit type resolution, and a case study of an IMP to ASM compiler to see how eqmR can be used in practice.

7.1 Typeclasses for EQM and Interp Laws

Figure 7.1 summarizes the collection of typeclasses supported by our interpretation framework. The red nodes are the category theory-relevant typeclasses from the Interaction Trees library (most notably the theory of iterative monads and equivalence on Kleisli arrows), the yellow nodes are standard functional programming typeclasses (functor, monad, monad transformers, higher-order functors), and the green nodes represent the structural properties that we have formalized in this framework. A dotted line connects an "operational" typeclass to its corresponding laws, and solid arrows represent dependencies.

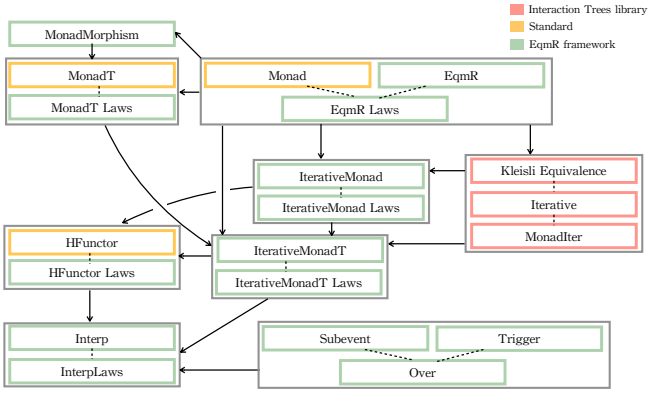

Figure 7.1: Typeclass dependencies in the EqmR framework

Custom Tactics for Extending Coq Typeclass Inference Without explicit support, the Coq typeclass inference algorithm often fails to infer the implicit arguments for `interp`, `over`, and `trigger` when using the `interp` laws. We solve this issue by providing custom tactics for each of the relevant lemmas for our `interp` laws: these are dubbed `i-tactics` for "interp law tactics". For each of the `interp` laws, there is a corresponding `i-tactic`, and we also have a specialized setoid-rewriting tactic `irewrite` for inferring the necessary setoid rewriting we need for the laws.

This is necessary because (1) there are multiple overlapping instances of equational laws (for instance,

both the strong and weak bisimulation in the ITree library satisfy eqmR rules), (2) the decomposition of stack of monad transformers is not unique (for instance, the identity monad transformer `fun x ⇒ x` is a trivial instance of a monad transformer), and (3) the type annotation for the sums of events of the `over` combinator in the IgnoreTrigger law (recall Figure 6.6) presents a difficult typeclass-resolution problem, and we would like the users to not use so many explicit type annotations while using these laws. We have developed a small custom tactic library for inferring the typeclass instances. Here we sketch (a simplified version of) the implementation of `ibind`, which corresponds to the application of the InterpBind law.

```
Ltac ibind_body TR h t k :=
  match type of h with
  | ∀ T : Type, ?E T → _ T ⇒
      let Hbind := fresh "Hbind" in
      pose proof
      (interp_bind (T := TR) (E := E)
         h k t) as Hbind;
      try (irewrite Hbind)
  end.
```

```
Ltac ibind_rec TR h x :=
  match x with
  | interp (T := ?TR) ?h ?x_
      ⇒
      ibind_rec TR h x_
  | bind ?t ?k ⇒
      ibind_body TR h t k
  end.
```

```
Ltac ibind :=
  match goal with
  | ⊢ eqmR _
      (interp (T := ?TR) ?h ?x)
      _ ⇒
      ibind_rec TR h x
  end.
```

Figure 7.2: Custom tactics for using the InterpBind rule

The custom tactics for using the InterpBind rule are shown in Figure 7.2. The base case is `ibind_body`, which instantiates the InterpBind law with explicit type arguments for the monad transformer `TR`, triggered event type `E`, handler `h`, continuation `k` and prefix `t` of the bind operator. In `ibind_body`, we must explicitly match the type of the handler, `h`, because, given a program `interp h (interp h' (x ← ma ;; k x))`, the typeclass inference in Coq cannot determine whether it should attempt to apply the commutation lemma to the inner or the outer `interp` function. The `irewrite` tactic infers the specific setoid instance needed in order to perform the rewriting. The `ibind_rec` recursively matches against the argument of `interp` to specify the innermost monad in the `interp` function, and the top-level tactic `ibind`, simply calls into `ibind_rec` to start the process.

7.2 Case Study : IMP to ASM Compiler

We have re-verified the correctness of a simple imperative language (IMP) to a simple assembly language (ASM) compiler, the case study presented in the original Interaction Trees paper [XZH+20], in this equational framework to illustrate how the equational laws are lifted in the stack of interpretation. In the following subsection we discuss the benefits to using the framework that we have observed.

7.2.1 Elegant Staged Interpretations : the ASM Example

Recall from Section 5.2.3 how the automatic injection of handlers eliminates the need for extra manual annotations. The benefit for our layered interpretation is larger when it comes to larger stacks of interpretations as presented in 5.1, but we present here a simplified version for expository purposes. We illustrate this point again in ASM, and illustrate another benefit: writing layered interpreters is more straightforward.

The ASM language has two event signatures: `Reg`, for registers, and `Memory`, for the heap, and has two handlers respectively, `h_reg` and `h_memory`. Let's look at the original definition of `interp_asm`, which uses `bimap` in order to manually inject the handlers for each of the events.

```
(* [interp_asm] definition without [over] and [interp] *)
Definition interp_asm {E A} (t : itree (Reg +' Memory +' E) A) :=
  let h := bimap h_reg (bimap h_memory (id_ _)) in
  let t' := interp h t in fun mem regs ⇒ interp_map (interp_map t' regs) mem.
```

The `bimap h_reg (bimap h_memory (id_ _))` manually injects the handlers for `Reg` and `Memory` for the signature of the program `itree (Reg +' Memory +' E) A`. The application of `interp_map` and `interp` does not accurately reflect the intuition that we are staging interpretation: in fact, the interpreters are being applied simultaneously. In addition, the event signature is very concrete, in the sense that the signature must provide the ordering of `Reg +' Memory +' E` for this `interp_asm` to apply.

The code below is the staged interpretation using `over` and the new `interp` combinators.

```
Definition interp_asm {D E F A} `{Reg +? D -< E} `{Memory +? E -< F} (t : itree F A) :=
  (interp (T := stateT memory) (over handle_reg) (interp (T := fun x ⇒ x) (over handle_mem) t)).
```

Observe the benefits in writing the interpretation in this manner: now, each stage of interpreting `Reg` and `Memory` are clearly distinct, and are compositional. The `over` annotation also eliminates the need for manual annotation. It is also extensible in the sense that adding another stage of interpretation is straightforward—the event signature is not rigid.

7.2.2 Structural Rules for Free

In the original IMP-to-ASM proof of correctness, one had to show that structural properties hold for each layer of interpretation. For instance, if one wanted to use the INTERPRET law with layers `stateT reg (stateT memory (itree E))`, an instance would need to be proven for each of `itree E`, `stateT memory (itree E)`,

and `stateT reg (stateT memory (itree E))` (or, less compositionally, one single, ad hoc instance that essentially combines all three proofs into one). This does not scale, especially when given a large stack interpreters with many laws. However, now we get these properties for free.

This illustrates how definitions can be simplified. What about proofs? Consider the following structural lemma stating how the interpretation function for ASM commutes with `bind`.

```
Lemma interp_asm_bind: ∀{R S} (t: itree E R) (k: R → itree E S),
    interp_asm (bind t k) ≈ (x ← interp_asm t ;; interp_asm (k x)).
```

Now compare the old proof to the new proof, as shown to the right. The old proof of the lemma had to refer explicitly to the bind commutation property at each layer (`interp_bind` and `interp_state_bind`), which are specific to the layers and cannot be composed with each other. In addition, it used the `eutt_clo_bind` lemma to perform rewriting under the monadic `bind`, while the intuitive reasoning principle should be "commute the bind operator under interp twice".

```
OLD PROOF:
  intros.
  unfold interp_asm, interp_map.
      cbn.
  repeat rewrite interp_bind.
  repeat rewrite interp_state_bind.
  repeat rewrite bind_bind.
  eapply eutt_clo_bind; [
      reflexivity | ..].
  intros. rewrite H.
  destruct u2 as [g' [l' x]].
  reflexivity.

NEW PROOF:
  intros; unfold interp_asm.
  do 2 ibind.
```

The new proof only has to apply the same bind commutation lemma using the `ibind` tactic, which essentially unfolds to invoking 'irewrite interp_bind' twice, while inferring the correct setoid instances for rewriting the INTERPBIND rule.

At each layer of interpretation, we have the same bind commutation property that holds, and we do not need to conjecture the structural property to hold or reprove it for each combination of layers.

7.2.3 Commuting Layers of Interpretation

To determine that we indeed have flexibly composable structural rules, we can modify the structure of an existing development and see how hard it is to "port" the proofs to the new structure. To that end, we modified the existing IMP to ASM compiler development by swapping the order in which registers and memory events are interpreted. Of course such a change necessitates certain modifications: for instance, we had to re-order the arguments to the relation connecting the IMP and ASM state monads. However, beyond those simple changes, most of the proofs go through with minimal differences—the most complicated

change being instances where the IgnoreTrigger rule applies at a different position in the stack.[23] The old proofs (like the one shown above) would be far less resilient to such changes, and therefore much more difficult to maintain than the new ones enabled by this framework.

7.3 Related Work

Monads, Monad Transformers, and Modular Interpreters. The approach of building modular interpreters from monads and monad transformers derives from an expansive literature. Moggi's seminal paper [Mog89a, Mog91] about using monads to characterize imperative features in a pure, functional setting was consequently popularized by Wadler [Wad90] and Peyton Jones [PJW93]. Monad transformers [Mog90] were then adopted as a way of composing various effects: notably Liang, *et al.* [LHJ95] showed how to build modular interpreters in that style, which we have adopted and formalized here. Swierstra [Swi08], Apfelmus [Apf10], Kiselyov, *et al.* [KSS13], and Kiselyov and Ishii [KI15] have showed how to use the *free* and *freer monads* to define modular monad instances. A related data structure to freer monads are Tlön Embeddings [LW22], which use *program adverbs* as a basis to allow more flexibility in computational modeling of effects. Interaction Trees [XZH+20] are a coinductive freer monad, and provide an instance where each interpretation layer forms an *iterative monad.* In our framework, we maintain that the range of interpretation is a stack of *iterative monads* and show how, in this practical setting, the laws for iteration can be composed and lifted through monadic interpreters.

Nesting interpreters are also a prominent feature of Johann and Ghani's work [JG09]. That work introduces two separate constructs for building layered monads: a base interpreter and a second interpreter for nesting. Our general `interp` definition subsumes both definitions, but uses a distinct base instance. Our eqmR framework shows how to build a formalized equational theory that works nicely with the lifting and plays well with Coq-style typeclasses, and so can be seen as a mechanized version of their results.

Algebraic Effects and Handlers. Interaction trees and languages with support for algebraic effects [PP03b, PP13, HPP06, PP09, BP14] share similar goals, namely flexible, programmable construction of semantics. As such, our work has taken inspiration from that literature. There are significant differences, however. Programming languages that implement algebraic effects, like Eff [BP15], are working in an ambient

[23] The supplementary Coq material contains the proofs.

environment that allows nontermination, whereas ITrees are crafted to fit with the total semantics of Coq. The handlers for algebraic effects are more general than those possible with ITrees. In particular, the ITree handlers do not have access to the continuation of the event, which makes them less expressive (giving the handler access to the continuation would be hard to realize in Coq because its termination checker would not be able to observe that manipulations of the continuation are sufficiently guarded to define valid cofixpoints).

The algebraic reasoning of algebraic effects is often used in that setting to characterize equations that hold for *particular* effects; for instance, for state effects the sequence `put s; put t` is equivalent to `put t` (since the second `put` overwrites the first). Such equivalences can be proven as theorems about particular monads used for interpreting events. In our context, such a theorem would justify a rewriting rule with respect to the state monad notion of eqmR. This thesis, however, focuses not on such monad-specific properties. Instead, we are looking at how to automatically construct the generic, structure-preserving, parts of equational theories compositionally.

Relational Logics. Benton's Relational Hoare Logic [Ben04] and Nanevski, *et al.*'s Relational Hoare Type Theory [NBG13] have been shown to be useful for reasoning about program transformations and properties such as information flow. The subsequent work on predicate transformers [SB19], Dijkstra Monads [AHM+17, MAA+19, MHRVM20], and F* [SHK+16] give a general framework for building program logics for arbitrary monadic effects. The Dijkstra Monad setting is especially relevant to our approach and is similar in that it builds a general logic for reasoning about monads, but is different in that it does not focus on composing reasoning about effects (*i.e.*, building a compositional theory using interpretation).

Logical Relations and Bisimulations for Monadic Types. There are many techniques for relational reasoning, including binary logical relations [DAB09, BKBH09, Ahm04] and bisimulations [LGL17, KW06, San12], and their combination [HDNV12, HRRS20]. The eqmR framework defines relations for monads that technically form logical relations. Logical relations over monadic types were developed by Goubault-Larrecq, *et al.* [GLN08], giving a sound basis for a monadic equivalence akin to a notion of bisimilarity. As in our approach, these techniques build an expressive basis for program logics and program verification [JSS+15].

PER Models. The notion of *image* is inspired from PER models of computation, as used in, *e.g.*, models of the lambda calculus with recursion and recursive types [Mit96, Chapter 5], for reasoning about operational

equivalences [AM01], or for giving different *views* of information [ABHR99]. In our setting, we use a PER model to gain set-theoretic reasoning about elements in a *monadic* types, as a way to refine our reasoning principles.

7.4 Discussion and Conclusion

We have presented a novel and principled approach to the construction of monadic interpreters built in layers from a free(r) monad structure such as ITrees. The tools we have introduced and formalized in Coq greatly reduce the boilerplate and glue code needed to construct such interpreters, and also provide for free the backbone of the equational theory necessary for any relational reasoning over the resulting structure. Our current implementation provides instances for the structures most commonly used in existing ITree projects, lifting the equational theory through state and error transformers. However, there is a zoo of monads: expanding the library to cover additional effects and monads would be a valuable extension.

Among those effects, nondeterminism is particularly worthy of attention. Indeed, we have presented in Section 6.1 the `prop` monad, but looking back at Figure 5.1, the Vellvm developers use a more general structure that models not just nondeterministic sets of *values*, but rather sets of *computations*. This approach relies on a hypothetical `prop` transformer, $\texttt{propT}_M\ A \triangleq M\ A \to \texttt{Prop}$. For Vellvm, the authors have fixed M to be `itree E`, defining a new ad hoc structure rather than a transformer. Interestingly, the reason for this was because they lacked a generic notion of image for a monadic computation, which we introduced through this dissertation. Indeed, they define the bind over this structure as: $\texttt{bind}\ P\ K\ t_b \triangleq \exists\ t_a,\ k,\ P\ t_a \wedge t_b \approx \texttt{bind}\ t_a\ k \wedge (\forall\ a,\ \texttt{Leaves}\ a\ t_a \to K\ a\ (k\ a))$. This should be read as judging whether an ITree t_b belongs to the bind: there should be a computation t_a in the prefix P and a continuation k such that *at any leaf of* t_a (*i.e.*, in the image of t_a to use our terminology), the continuation belongs to the nondeterministic set. Without the restriction to the image, this construction is completely ill-behaved. The reason we don't include `propT` here is that, even using the image, $\texttt{propT}_M$ doesn't lift the monad laws properly—this definition of `bind` does not associate to the left (an expected artifact of the nondeterminism [MHRVM20]). Thus, developing clean nondeterministic transformers for monadic interpreters remains an interesting prospect.

With these extensions to the ITree library in place, we lighten the bureaucratic boilerplate of structural rules when programming and reasoning about nested monadic interpreters. Of course, monadic reasoning

can be about more than its structural rules: the extensions in this dissertation express nothing about monad-specific algebras. This aspect of the reasoning is still fairly ad hoc in the ITree landscape; combining our contribution with other techniques, such as Dijkstra monads [MAA+19, MHRVM20], would likely further improve the viability of relational reasoning for monadic interpreters at scale.

Part III

A Separation Logic Framework

Chapter 8

Introduction to Separation Logic

LLVM IR transformations involve modifying code that contains stateful and effectful operations. However, in the previous chapters, there were no strong reasoning principles for stateful transformations. Its proofs of optimizations only reasoned about control flow and did not involve stateful computation. For larger pieces of code, proofs of optimizations become quickly excruciating without a method to support localized reasoning about state. Without a modular program logic, it is difficult to prove complex transformations correct. As a solution, we propose a *relational separation logic* framework for VIR, which provide useful abstractions for reasoning about LLVM IR transformations.

In this chapter, we will first define the basics of Hoare logic, explain the Hoare logic already present in VIR, and then introduce the concepts of separation logic and ghost resources. In fact, one can view the relational reasoning principles from Section 4.2.1 as relational Hoare triples for monadic programs. Chapter 9 will introduce the fundamentals for Velliris, our relational separation logic framework VIR, and the remainder of this dissertation will discuss the theory and model of this framework.

8.1 A Hoare logic for Interaction Trees

Hoare logic is an axiomatic semantics: it is a program logic that assigns meanings to programs. A set of rules are used to reason about programs, where the behavior of a program is described through the relationship between the properties of its input and output. A unary Hoare triple over a program e can be written as $\{P\}e\{Q\}$, which means that if P holds before the execution of e, and if e terminates, then Q holds after the execution of e. Note that Hoare triples state a *partial correctness* assertion over a program, where it checks that a program satisfies the postcondition when it terminates, but does not provide any guarantees about the termination of the program.

Relational Hoare logic (RHL) is a program logic for relating two programs. The relation between two programs may be about program equivalence or refinement, which is useful in compilation or showing similarity between two versions of the same program. Given a relation $\preceq$ that relates two programs e_t and

$$\frac{P \Rightarrow Q(r_1, r_2)}{\{P\}\ \texttt{ret}\ r_1 \approx \texttt{ret}\ r_2\ \{(v_1, v_2),\ Q\ (v_1, v_2)\}}\ \textsc{HRet} \qquad \frac{\{P_1\}\ t_1 \approx t_2\ \{(v_1, v_2),\ R_1\ (v_1, v_2)\} \quad \{P_2\}\ t_1 \approx t_2\ \{(v_1, v_2),\ R_2\ (v_1, v_2)\}}{\{P_1 \circ P_2\}\ t_1 \approx t_2\ \{(v_1, v_2),\ R_1 \circ R_2(v_1, v_2)\}}\ \textsc{HTrans}$$

$$\frac{\{P\}t_1 \approx t_2\ \{(v_1, v_2),\ U\ (v_1, v_2)\} \quad \forall u_1\ \ u_2,\ \{U(u_1, u_2)\}\ \ k_1\ \ u_1 \approx k_2\ \ u_2\ \ \{(v_1, v_2),\ \ R\ \ (v_1, v_2)\}}{\{P\}x \leftarrow t_1;;(k_1\ x) \approx x \leftarrow t_2;;(k_2\ x)\{(v_1, v_2),\ R\ (v_1, v_2)\}}\ \textsc{HCloBind}$$

$$\frac{\{P_1\}\ \ t_1 \ \approx\ t_2\{(v_1, v_2),\ \ R_1\ (r_1, r_2)\} \quad P_2 \subseteq P_1 \quad R_1 \subseteq R_2}{\{P_2\}\ t_1 \approx t_2\ \{(v_1, v_2),\ R_2\ (r_1, r_2)\}}\ \textsc{HMon} \qquad \frac{\{P\}\ t_1 \approx t_2\{(v_1, v_2),\ R\ (r_1, r_2)\}}{\{P\}\texttt{interp}\ h\ t_1 \approx \texttt{interp}\ h\ t_2\ \{(v_1, v_2),\ R\ (r_1, r_2)\}}\ \textsc{HInterp}$$

Figure 8.1: Relational reasoning principles over ITrees, Floyd-Hoare style

e_s, $\{P\}e_t \preceq e_s\{Q\}$ states a relational pre- and post-condition P and Q over the two programs.

An important liveness property for RHL is termination. In showing that two programs are similar, it is useful to show termination-preserving refinements between two related programs. A refinement $e_t \preceq e_s$ that does not preserve termination states that if a target program e_t terminates, then the source program e_s terminates. A termination-preserving refinement $e_t \preceq e_s$ states that (1) if a target program e_t terminates, then the source program e_s terminates and (2) if an execution of a target program e_t diverges, then there exists an execution of the source program e_s that diverges.

Now, we can observe that the bisimulation between ITrees we have seen in previous chapters can form a termination-preserving relational Hoare triple. $e_t \approx_R e_s$ can be seen as the relational Hoare triple $\{\top\}e_t \approx e_s\{(v_t,\ v_s),\ R\ v_t\ v_s\}$. Intuitively, $e_t \approx_R e_s$ means that two possibly diverging programs e_t and e_s either (1) both terminate and satisfy the postcondition R over the result of the computation (where v_t and v_s binds the resulting values of the computation), or (2) both diverge and the programs diverge in simulation with each other. Moreover, the relation $\approx_R$ is monotonic with respect to its parameterized relation R, so it is useful to use this relation for proving simulation assertions between two programs.

To capture a generic relational Hoare triple over ITrees, we can easily define $\{P\}e_t \approx e_s\{(v_t\ v_s), Q\ v_t\ v_s\}$ as $P \rightarrow e_t \approx_Q e_s$. Given this definition, it is easy to translate the relational reasoning principles from Section 4.2.1 as Hoare triples. The definition of relational Hoare triples over ITrees is shown in Figure 8.1. Using these rules, we can define properties about programs denoted with ITrees using Floyd-Hoare style reasoning.

8.2 Separation logic

Since the advent of Hoare logics, many variants have emerged to aid modular specification and verification of programs. A successor of Hoare logic is *separation logic* [Rey02, ORY01], which provides a structured logical framework for reasoning about stateful programs. It is particularly well-suited to programs that manipulate dynamic data structures and employ resource management. It offers a means to express and verify properties related to program state separation, ensuring that different parts of a program do not interfere with each other, even in a shared memory environment. This characteristic of separation logic makes it invaluable for reasoning about complex, concurrent, or dynamic programs. We begin by introducing the foundational concepts of separation logic in the following section.

Separation logic is a resource logic based on the logic of bunched implications (BI) [OP99]. Central to separation logic is the concept of *separation*, which allows the logic to distinguish between disjoint portions of the memory. Separation Logic introduces the *separating conjunction* operator, denoted as "$*$," to express the idea that two sets of memory regions do not overlap, thereby enabling local reasoning about program components. This concept of locality simplifies proofs, reduces the need for global invariants, and enhances the modularity of verification.

Assertions We present a separation logic with the following grammar.

$$\begin{aligned}\text{Propositions } P, Q ::= {} & \top \mid \bot \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid P \Leftrightarrow Q \mid \neg P \\ & \mid \exists x.P \mid \forall x.P \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \mapsto v\end{aligned}$$

The classical logical operators ($\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\neg$, $\forall$, $\exists$) have their expected definitions. The interesting part of this logic is the separation conjunction $*$ and the separating implication $\mathrel{-\!\!*}$.

The assertion $\ell \mapsto v$, read as ℓ points to v, states that the location ℓ in the heap stores the value v. The assertion $P * Q$ is the *separating conjunction* is similar to classical conjunction ($\wedge$), in that it states the assertion of both P and Q. However, it also asserts that P and Q are assertions on disjoint regions of the heap. The proposition $\ell \mapsto v * \ell \mapsto w$ is false for any ℓ, v, , and w. Similarly, the separating implication

$P \mathrel{-\!\!*} Q$ (the *magic wand* operator) is similar to the classical implication ($\Rightarrow$). It states Q holds under the assumption of ownership over the heap fragment described by assertion P.

Proof rules The most novel proof rule in separation logic is the *frame rule*.

$$\frac{\{P\}C\{Q\}}{\{P * R\}\ C\{Q * R\}}\ \textsc{SLFrame}$$

This rule means that if the program C executes with its initial heap satisfying P and after execution its heap satisfies Q, then it is also safe to add a disjoint resource R. Since C only needed to access a disjoint region of the memory than R, it will leave R untouched by the end of its execution.

8.3 Ghost resources

Ghost resources are auxiliary resources that are not present in the actual program execution but are used for reasoning and verification purposes. In the context of separation logic, *ghost resources* play a vital role. They allow us to attach additional information or invariants to memory locations without affecting the program's operational behavior.

So far, we have considered a concrete resource (heap) that is present in a program's execution. In general, separation logic does not need to be instantiated with a concrete resource. In fact, given a *resource algebra* that can state a notion of separation over its resources, we can instantiate a separation logic over it. Here, we define a simple resource algebra for separation logic.

8.3.1 A resource algebra: partial commutative monoid

We present a resource algebra $(R, (\odot), \epsilon)$ which is a partial commutative monoid. Intuitively, the laws state that resources can be added, but that certain resources cannot be added together.

The infix partial binary operator $(\odot) : R \times R \rightharpoonup R$ denotes the addition of two resources. The operator is partial because not all resources can be combined. In the case of the heap example earlier, two resources could be combined only if they are disjoint. The operator is a commutative monoid with an identity ϵ so that it can behave as an addition operator, as addition is typically associative and communitive and has an identity element.

$$m \odot \epsilon = m \qquad \text{(identity)}$$

$$m_1 \odot m_2 = m_2 \odot m_1 \qquad \text{(commutativity)}$$

$$(m_1 \odot m_2) \odot m_3 = m_1 \odot (m_2 \odot m_3) \qquad \text{(associativity)}$$

Example 7. We instantiate a resource algebra for a simple heap which is defined as a partial map from addresses to booleans. The carrier type R is the partial map from addresses to values, represented as a list of mapping to locations to values, $(\ell \longmapsto v; \ell' \longmapsto w)$, is a heap containing two locations, ℓ and ℓ') with an append function $^{++}$. The operator $\odot$ is defined as the following, and the unital element ϵ is an empty map ().

$$(\ell \longmapsto v) \odot (\ell \longmapsto w) := \bot$$

$$(\ell \longmapsto v) \odot (\ell' \longmapsto w) := (\ell \longmapsto v; \ell' \longmapsto w) \qquad \textit{where } \ell \neq \ell'$$

$$L \odot L' := (L \mathbin{^{++}} L') \qquad \textit{where } \mathsf{dom}(L) \cap \mathsf{dom}(L') = \emptyset$$

$$L \odot L' := \bot \qquad \textit{where } \mathsf{dom}(L) \cap \mathsf{dom}(L') \neq \emptyset$$

Often times, it is useful to have an abstract characterization of resources using a resource algebra, in order to offer user-defined ghost resources for a given separation logic. Characterizing resources with partial commutative monoids has been used in various separation logics [NLWSD14, TB21]. Modern separation logics, however, also use other variants of resource algebras to offer more expressive ghost resources for the user.

8.4 Iris as a program logic

In this thesis, we discuss a specific variant of separation logic, namely the bunched implication (BI) logic in the *Iris* framework. Iris [JSS+15, KJB+17] is a highly expressive separation logic framework for concurrent, higher-order programs. Iris was designed to consolidate the foundations of modern separation logics, and is used for higher-order imperative languages such as Rust and ML. The Iris framework is supported by a vibrant and active community of researchers and developers. The ecosystem around Iris includes libraries, tools, and case studies that further extend its applicability and utility.

We present in this section a simplified "core" of Iris 4.0 that is relevant for using Iris as a separation logic. We denote the BI logic of Iris as bi_{Iris}. The other features of Iris will be explained as necessary in the following chapters. In Chapter 10, we will further explain the ghost theory and resource algebra of Iris, as we define a ghost theory for LLVM IR in bi_{Iris}. In Chapter 11, we will give an overview of the standard model of Iris in order to explain the model constructed in this thesis.

8.4.1 Base logic: a bunched implication (BI) logic

We present a fragment of the bi_{Iris}logic that is relevant to this thesis. The fragment is Iris^{light}("Iris without the step-indexing"), where we keep the basic logic of BI and the persistence modality, but we ignore the step-indexing aspects of Iris. We can view this logic as an extension to the set of assertions introduced in 8.2.

Iris^{light} assertions, distilled. [24]

$$\text{Propositions } P, Q ::= \ulcorner \phi \urcorner \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \Box P \mid \mathsf{wp}\ e\ \{v.P\ (v)\}$$

$\mathsf{Iris}^{light}$is an embedded logic within the meta-logic of Coq. We can state logical assertions from the meta-logic using the notation $\ulcorner \phi \urcorner$ to state a Coq proposition ϕ.

As a separation logic, it has separating conjuction $*$ and the separating implication (magic wand) operator $\mathrel{-\!\!*}$. The ghost resources in the logic can be user-defined, and can be instantiated to any resource

[24] Note that this simplified presentation of Iris^{light}ignores the discussion around namespaces and ghost updates.

that satisfies the algebraic properties of a *camera* (a generalization of a partial commutative monoid).

The judgment $P \vdash Q$ denotes *entailment*, and is a statement in the meta-logic of Coq. The introduction and elimination of the magic wand operator with respect to entailment is as follows.

$$\frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}\ \textsc{WandIntro} \qquad \frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}\ \textsc{WandElim}$$

Iris^{light} is an *affine* logic, which means that propositions which state ownership over a resource cannot be duplicated. More precisely, the structural rule of *contraction* is rejected in this logic.

$$\frac{P * P \vdash Q}{P \vdash Q}\ \textsc{RejectedContraction} \quad \frac{P \vdash Q}{P * R \vdash Q}\ \textsc{AcceptedWeakening}$$

Unlike a linear logic, it does not reject the weakening rule as well. In practice, linear logic "must use a resource *exactly* once," whereas affine logics "must use a resouce *at most* once." Affine logics are often useful for reasoning about memory management, as in practice allocated addresses do not necessarily need to be used, but should not be used more than once as it may lead to data races or the double-freeing pointers.

8.4.2 Weakest preconditions, and the persistent modality

The weakest precondition statement $\mathsf{wp}\ e\ \{v.Q\ (v)\}$ is the core to using Iris^{light} as a Hoare logic. The proposition $\mathsf{wp}\ e\ \{v.Q\ (v)\}$ asserts that if e terminates, the resulting value v satisfies the postcondition Q. Hoare triples $\{P\}e\{v.Q\ (v)\}$ are defined as $P \vdash \mathsf{wp}\ e\ \{v.Q\ (v)\}$. The definition of wp involves the model of Iris^{light}, which we will discuss further in Section 11.2.

Iris^{light} also has a *persistent modality* $\square\ P$ in order to use previously-proved Hoare triples. Hoare triples $\{P\}e\{v.Q\ (v)\}$ can be embedded in Iris^{light} with $P \mathrel{-\!\!*} \mathsf{wp}\ e\ \{v.Q\ (v)\}$, and these assertions must be *duplicable* in order to be reused.

The main rules for persistence is as follows.

$$\textsc{PersDup}\quad \square P \vdash (\square P) * (\square P) \qquad\qquad \textsc{PersElim}\quad \square P \vdash P$$

The rule PersDup states that any persistent statement can be duplicated, and PersElim states that any persistent statement can eliminate its modality.

With the weakest precondition and persistent modality, Iris^{light} can be used as a Floyd-Hoare style separation logic. In the next chapter, we will see the relational variant of this logic, along with a construction of a ghost theory for LLVM IR. For more details about the foundation of the Iris framework, we refer the readers to Jung et al [JKJ+18].

Chapter 9

Velliris: A Relational Separation Logic for LLVM IR

In the previous chapter, we have introduced the basics of separation logic and how to use them to reason about the correctness of programs. This chapter introduces Velliris, the first relational separation logic framework for LLVM IR, integrated with the VIR formal semantics. The logical framework allows expressive reasoning principles about LLVM IR programs, with a practical ghost theory over LLVM IR resources. Formally, Velliris is implemented by constructing, within the Iris separation logic framework, a novel and general coinductive simulation over Interaction Trees with interpretations into state monads. We prove the logic sound by establishing a contextual adequacy theorem that lifts simulation results from the embedded logic to the ambient Coq logic. To demonstrate Velliris' utility, we focus on verifying optimizations that exploit LLVM IR's *memory attributes* and also prove correct a loop invariant code motion example, both of which were inexpressible in prior versions of VIR.

9.1 Introduction

There are at least two key components to any formally verified software effort (besides the software itself): (1) a *formal semantics* that characterizes the meaning of the constructs of the programming language in which the software is written, and (2) some kind of *program logic* in which the developer expresses the specification of the program and builds a proof that the program meets that specification. These two pieces must fit together: the constructs of the logic must accord with the behaviors that they purport to describe, i.e., the logic should be *sound* with respect to the semantics.

One influential and prominent example of such a framework is given by CompCert [Ler09] and the Verified Software Toolchain (VST) [App11, CBG+18], which has been used to verify various data structures and applications implemented in C [KLL+19, ZHK+21, MAN17, BPYA15]. CompCert's success has ushered in a golden age of software formalization, and there are now many efforts to produce similar results for a wide variety of programming languages and systems (see Ringer, et al's survey [RPS+19b])—research on both formal language semantics and program logics is flourishing!

Nevertheless, formal verification is still typically expensive in terms of both time and effort, and technically challenging, often requiring significant experience with complicated tools such as Coq [Tea20] and the facility with mathematical logic to correctly specify and prove the desired metatheoretic and system properties. For that reason, there is incentive to try to develop frameworks that can lighten the load of formal development, by allowing work on language semantics and program logics to be re-used and shared.

On the language semantics side, one promising approach is that of *interaction trees* (ITrees) [XZH+20], a data structure for representing computations as potentially infinite trees of "uninterpreted" events. Rich computational behaviors can be modeled by interpreting those events into different *monads* [Mog89b] that realize different effects, such as state, errors, I/O, or nondeterminism [YZZ22]. Such layered monadic interpreters [Ste94] offer a modular and compositional approach to building language semantics. Since their introduction, ITrees and related variants have been used in many contexts [FHW21, SZ21, LXK+22, SHC+23, SWYS23, SSS+23, SCL+23]. Most relevant to this work is their use in Vellvm [ZBY+21]—a Coq-based semantics for a large, practical subset of LLVM IR [LA04]. We refer to this formalized subset of LLVM IR as VIR, following Zakowski, et al.

On the program logic side, Iris [JSS+15] stands out as a flexible way to construct language-specific (concurrent) separation logics, the state-of-the-art way to reason about complex software. Iris is a Coq-based framework that provides infrastructure and general-purpose metatheory for creating language-specific program logics. Key to its design is a notion of "ghost resources" and "ownership" that can be used to soundly model various aspects of program semantics. It too has seen widespread adoption, and so Iris has been used to develop program logics for reasoning about Rust [JJKD17, MDJD22], web assembly [RGL+23], ARM [SHL+22], data structures [CJS+23, SLK+21], distributed systems [SJT+23, KJTO+20], and, recently, CompCert semantics [MD24] (among many other applications).

A natural idea is to try to marry these two approaches, that is, to define a sound program logic within Iris for reasoning about ITrees-defined semantics such as those found in Vellvm, and there have indeed been efforts along these lines [SSS+23, SCL+23]. However, these prior approaches have concentrated on core calculi and comparatively simple languages. Scaling those techniques up in a way that is suitable for reasoning about Vellvm's VIR semantics is the main contribution of this thesis, but achieving it is not trivial.

The Iris infrastructure was designed to work with *operational* semantics, but those properties are

at odds with the approach of *layered, monadic interpreters* over *coinductively-defined trees* used by VIR, and the differences are significant. To reconcile them, we build the machinery needed to reason about ITrees in a program logic, following the example of Simuliris [GSS+22]. More specifically, we provide a novel coinductive simulation of Interaction Trees as a weakest-precondition model in Iris. The weakest-precondition model is the basis for a relational Hoare "quadruple" that relates a source and target program to given pre- and post-conditions. Whereas typical models built in Iris use a weakest-precondition model based on *step-indexing* and operational semantics, our construction provides a coinductive simulation over ITrees by building a Knaster-Tarski mixed inductive-coinductive fixpoint in the logic of Iris. On top of this model, we provide expressive coinductive reasoning principles over fixpoint combinators for iteration and mutual recursion, features necessary to model VIR. (Notably, Simuliris also eschews step-indexing in favor of coinduction principles, but the logics built with it are tailored for reasoning about concurrent programs represented using small-step operational semantics and it isn't applicable "out of the box" to VIR's semantics.)

To demonstrate the utility of this approach, we then build a program logic, Velliris, for reasoning about VIR code in Iris and show that it is adequate for reasoning about (a large subset of) Vellvm's VIR semantics. The key definition is a notion of refinement between two VIR programs, and the key result is a proof of contextual refinement, which can be used to (among other things) justify the correctness of program transformations. Importantly, and novel to this work, Velliris supports LLVM IR's *attribute specifications*, which record the compiler's assumptions about (potentially external) function calls—attributes like *readonly* or *argmemonly* affect which optimizations are applicable because they constrain the assumed behavior of the functions. Such specifications are easy to encode in Velliris, but to implement them, we had to rectify some deficiencies in Vellvm's handling of state and external calls.

Defining separation logic reasoning principles in Velliris gives an elegant specification to side-effects in LLVM IR and helps us prove optimizations in a modular way. We develop a robust theory of "ghost resources" and reasoning principles for VIR semantics in Velliris. Importantly, this separation logic greatly streamlines proofs: prior to this development, reasoning about VIR code had to be done over the entire program state, which includes the global and local environment, stack, and the entire memory. For larger pieces of code, proofs of optimizations quickly became excruciatingly painful due to the lack of localized

reasoning about state. For this reason, prior VIR verification efforts concentrated mainly on control-flow optimizations. The new Velliris logic is useful for proving properties of VIR code, and we illustrate that by proving the correctness of an instance of loop-invariant code motion in the presence of external calls.

To summarize, the contributions of this part are the following:

- We develop an instance of a relational, coinductive weakest precondition model of Iris which supports a *monadic semantics* based on the Interaction Trees framework. Section 11.2 describes the model and the proof of adequacy. The proof of adequacy lifts results from the embedded logic to the ambient Coq logic, and also demonstrates that the embedded simulation is compatible with the equational theory already present in the metatheory of VIR.
- We develop a separation logic for Vellvm's VIR, with a relational Hoare quadruple defined with the ITree-based weakest precondition, by defining a ghost theory for VIR resources. The relational reasoning techniques and core properties are described in Section 9.2.
- We enable reasoning about function calls that have memory-relevant *attribute specifications*. The simulation in Velliris is aware of a logical interpretation for *memory attributes*, and controls permissions based on the type of attribute decorated on a function call. This is discussed in Section 9.3.
- We develop a formalization and contextual refinement proof for our logic, which is presented in Section 11.3. This involves performing *coinductive proofs* about iteration and mutual recursion in Velliris (Section 9.4). We use these reasoning principles to reason about a simple loop invariant code motion algorithm in Section 9.4.2. The class of transformations based on loops and reordering calls demonstrated in this thesis are optimizations which are either too painful, or impossible, to prove with prior versions of VIR.

9.2 The Program Logic of Velliris

Velliris is a *relational separation logic* framework for VIR, allowing for modular reasoning about program transformations. Instead of building a separation logic from scratch, we build on Iris's base logic and leverage its expressive support for separation logic resources.

$$\frac{\{P\}\ e_s\ \{v_s.\ \Psi\ v_s\}^{\mathsf{src}} \quad \forall v_s.\ \Psi\ v_s \mathrel{-\!\!*}\ e_t \preceq k_s\ v_s\ \{\Phi\}}{\{P\}\ e_t \preceq x \leftarrow e_s\ ;;\ k_s\ x\ \{\Phi\}}\ \textsc{SourceFocus}$$

$$\frac{\{P\}\ e_t\ \{v_t.\ \Psi\ v_t\}^{\mathsf{tgt}} \quad \forall v_t.\ \Psi\ v_t \mathrel{-\!\!*}\ k_t\ v_t \preceq e_s\ \{\Phi\}}{\{P\}\ x \leftarrow e_t\ ;;\ k_t\ x \preceq e_s\ \{\Phi\}}\ \textsc{TargetFocus}$$

$$\frac{\{P\}\ e_t \preceq e_s\ \{v_t\ v_s.\ \Psi\ v_t\ v_s\} \qquad \forall v_t\ v_s.\ \Psi\ v_t\ v_s \mathrel{-\!\!*} k_t\ v_t \preceq k_s\ v_s\ \{\Phi\}}{\{P\}\ (x \leftarrow e_t\ ;;\ k_t\ x) \preceq (x \leftarrow e_s\ ;;\ k_s\ x)\ \{\Phi\}}\ \textsc{SimBind}$$

$$\{\Phi\ v_t\ v_s\}\ \mathsf{ret}\ v_t \preceq \mathsf{ret}\ v_s\ \{\Phi\}\ \textsc{SimValue} \qquad \frac{\{P\}\ e_t \preceq e_s\ \{\Phi\}}{\{P * R\}\ e_t \preceq e_s \{v_t\ v_s.\ \Phi\ v_t\ v_s * R\}}\ \textsc{SimFrame}$$

Figure 9.1: Benton-style relational reasoning (+ frame rule)

While the underlying model for Velliris is itself novel and interesting—no existing instantiation of Iris concerned simulations over coinductive datatypes such as ITrees—we defer its discussion to 11.2 and focus on the user-interface of Velliris here.

9.2.1 Benton-style relational reasoning (+ frame rule)

In this section, we explain the Benton-style relational Hoare reasoning supported by Velliris. Specifically, we build a relational Hoare quadruple that allow local reasoning for the relational simulation.

The Hoare quadruple $\{P\}\ e_t \preceq e_s\ \{v_t\ v_s.\Phi\ v_t\ v_s\}$ denotes that under the relational precondition P (typically about the resources of both source and target programs), the source expression e_s simulates the target expression e_t. If the programs both terminate, the returned values (bound by v_t and v_s) must be related by the postcondition Φ, otherwise they diverge in simulation. In this sense, this relational Hoare quadruple is also a termination-preserving relation akin to the Hoare quadruple derived from the eutt relation.

The relational reasoning rules for this quadruple are given in Figure 9.1.[25] The logic provides unary reasoning principles through source- and target- focus rules (SourceFocus, TargetFocus), and a cut principle for the bind operator (SimBind). These rules are standard, and mimic the already-present reasoning principles on VIR.

Velliris also supports the well-known "frame rule" of separation logic via the SimFrame rule. It enables local reasoning through the *separating conjunction*, $P * Q$, which asserts that P and Q are assertions about *disjoint* resources. Then, the SimFrame rule states that we can *frame* a resource R around a simulation

[25] All of the rules are proved as lemmas in Coq, using the weakest-precondition model defined in Section 11.2.

$$
\begin{aligned}
\mathit{Global} &::= \mathit{id} \hookrightarrow \mathcal{V} \\
\mathit{Local} &::= \mathit{id} \hookrightarrow \mathcal{V} \\
\mathit{Frame} &::= \mathsf{list}\ \mathit{Addr} \\
\mathit{LocalStack} &::= \mathsf{list}\ \mathit{Local} \\
\mathit{id} &::= \mathit{string} \quad \mathit{Addr} ::= \mathcal{Z} * \mathcal{Z}
\end{aligned}
\qquad
\begin{aligned}
\mathit{FrameStack} &::= \mathsf{list}\ \mathit{Frame} \\
\mathit{Allocated} &::= \mathsf{list}\ \mathit{Addr} \\
\mathit{Mem} &::= \mathit{Allocated} * (\mathit{Addr} \hookrightarrow \mathsf{list}\ \mathit{byte}) \\
\mathit{byte} &::= \mathsf{SUndef} \mid \mathsf{Byte}\ \mathcal{Z} \mid \mathsf{Ptr}\ \mathit{Addr} \mid \mathsf{PtrFrag}
\end{aligned}
$$

Figure 9.2: VIR state

proof. Intuitively, a simulation proof with precondition P accesses just the resources specified by P, and thus preserves any other resources that could be specified as R.

9.2.2 VIR Resources

A crucial part of developing a program logic for a language in Iris is coming up with ghost theory for its state. For VIR, the state involves a global environment, local environment, memory, and the stack which keeps track of both the (1) local registers and (2) the stack-allocated variables. We first explain the VIR state, and then introduce the ghost resources via the rules of the program logic and examples. The full-fledged ghost theory of Velliris is elided in this thesis due to space constraints, and is available in the technical appendix.

9.2.2.1 VIR State

VIR's state is described in Figure 9.2. The global state is a read-only environment, a map that is instatiated only once at initialization, mapping *id*s to values. The local state corresponds to the stateful component of resources that live within the scope of a function. Because LLVM IR automatically deallocates all locally allocated addresses upon function return, the local state must keep track of the list of addresses allocated at the current frame. Thus, the local state consists of a local environment (*Local*) that maps local *id*s to uvalues, and also a list of addresses (*Frame*) allocated at the current frame. Because a program consists of multiple functions and thus will have multiple frames throughout its execution, it keeps track of a stack of local environments (*LocalStack*) and local frames (*FrameStack*).

We work with a logical view of VIR memory: a map between addresses with an offset (a pair of integers $\mathcal{Z}$), which then points to a list of bytes. The representation of bytes comprises an undefined byte (SUndef) for uninitialized bytes in memory, literal bytes (Byte), a pointer byte for locations that store an address

(Ptr), and a pointer fragment that is used for alignment of pointer bytes in memory (PtrFrag). The memory also keeps track of a monotonically increasing set of addresses, in order to prevent reuse of addresses (an address from a new allocation should be guaranteed to be fresh).

Remark. Note that we discard the physical view of VIR memory (as seen in the quasi-concrete memory model [KHM+15b]). The study of robust memory models of LLVM IR is a subject of ongoing research. The VIR memory presented in Zakowski et al. [ZBY+21] is a quasi-concrete memory model that supports casting between integers and pointers. Supporting integer to pointer casting in Velliris remains as future work, along with adapting it to other advanced memory models of LLVM IR, such as the twin-memory model [LHJ+18a], and on-going work on robust support of out-of-memory errors.

9.2.3 Velliris event laws and instruction laws

VIR formalizes categories of LLVM *events*, building a denotational domain over LLVM IR. The categories of events include global state, local state, (function) calls, memory, undefined behavior, and failure. Thanks to its *event*-based semantics, it is possible to give modular specifications to VIR. Each VIR *instruction* is composed of several side effects. Figure 9.1 shows the memory-related primitive event laws, and a simplified set of instruction laws that use them as their building blocks.

$$\begin{aligned}&[\![(id, \texttt{alloca}\ (\tau))]\!]_i = \ dv \leftarrow \\ &\quad \texttt{trigger}\ (\texttt{Alloca}^{\mathcal{V}}(\tau))\ ;; \\ &\quad \texttt{trigger}\ (\texttt{LWr}^{()}(\uparrow id, dv))\end{aligned}$$

As an example, lets us consider LLVM's `alloca` τ instruction, which allocates a new block of memory that can store values of type τ, and returns the address of the new memory block. The corresponding denotation is given on the left. LLVM instructions are represented by a pair $(id,\ ins)$ of a side-effectful instruction ins and an identifier id destined to receive the result of the operation. Representing an operation (id, e) reduces to calling $[\![e]\!]_e$, the denotation of the expression being bound to the identifier, and binding its result with the trigger of the local write $\texttt{LWr}^{()}(id, uv)$ (the superscript denotes the *return type* of the event, which is unit for local writes). The `alloca` instruction performs two side-effects: $\texttt{Alloca}^{\mathcal{V}}(\tau)$, which allocates a new block in memory with size corresponding to the type τ, and $\texttt{LWr}^{()}(\_, \_)$ which writes to the local environment.

Memory-relevant event rules

SourceAlloca
$\{\mathsf{Alloca}_i^{\mathsf{src}}\ S * \mathsf{Frame}^{\mathsf{src}}\ i\}$
  $\texttt{trigger}\ (\texttt{Alloca}^{\mathcal{V}}(\tau))$
$\{v_s'.\ \exists \ell_s.v_s = \ell_s * \ell_s \mapsto^{\mathsf{src}} \mathsf{new\_block}^{\tau} * \mathsf{Alloca}_i^{\mathsf{src}}\ \{z\} \cup S * \mathsf{Frame}^{\mathsf{src}}\ i\}^{\mathsf{src}}$

SourceLoad
$\{(v \in \tau) * \ell_s \mapsto^{\mathsf{src}} v\}$
  $\texttt{trigger}\ (\texttt{Load}^{\mathcal{V}_u}(\tau, \mathsf{Addr}(\ell)))$
$\{v_s'.\ v_s' = v * \ell_s \mapsto^{\mathsf{src}} v\}^{\mathsf{src}}$

SourceStore
$\{(v \in \tau) * \ell_s \mapsto^{\mathsf{src}} v\}$
  $\texttt{trigger}\ (\texttt{Store}^{()}(\mathsf{Addr}(\ell), v'))$
$\{v_s'.\ \ell_s \mapsto^{\mathsf{src}} v'\}^{\mathsf{src}}$

UB and Exception event rules

SimUB $e \preceq \texttt{trigger}\ (\texttt{UB}^{\emptyset})\{\Phi\}$
SimExc $e \preceq \texttt{trigger}\ (\texttt{Throw}^{\emptyset})\{\Phi\}$

Local environment rules

LocalWrite
$\{(id \notin L) * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Local}_i^{\mathsf{src}}\ L\}$
  $\texttt{trigger}\ (\texttt{LWr}^{()}(\uparrow id, v))$
$\{v_s'.\ \langle \mathsf{id} := v\rangle_i^{\mathsf{src}} * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Local}_i^{\mathsf{src}}\ (\{[\%id]\} \cup L)\}^{\mathsf{src}}$

LocalRead
$\{\langle \mathsf{id} := v\rangle_i^{\mathsf{src}} * \mathsf{Frame}^{\mathsf{src}}\ i\}$
  $\texttt{trigger}\ (\texttt{LRd}^{\mathcal{V}_u}(\uparrow id))$
$\{v_s'.\ v_s' = v * \langle \mathsf{id} := v\rangle_i^{\mathsf{src}}\}^{\mathsf{src}}$

Memory-relevant instruction rules

SourceInstrAlloca
$\{(x \notin L) * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Alloca}_i^{\mathsf{src}}\ S * \mathsf{Local}_i^{\mathsf{src}}\ L\}$
  $\%x^{\mathsf{Id}} = \mathsf{alloca}\ \tau$
$\{\exists \ell_s.\ \ell \mapsto^{\mathsf{src}} \mathsf{new\_block}^{\tau} * \langle \mathsf{x} := \mathsf{Addr}(\ell)\rangle_i^{\mathsf{src}} * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Alloca}_i^{\mathsf{src}}\ S * \mathsf{Local}_i^{\mathsf{src}}\ (\{[\%x]\} \cup L)\}^{\mathsf{src}}$

SourceInstrLoad
$\{(v \in \tau) * (x \notin L) * \ell \mapsto_q^{\mathsf{src}} v * \langle \mathsf{ptr} := \mathsf{Addr}(\ell)\rangle_i^{\mathsf{src}} * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Local}_i^{\mathsf{src}}\ L\}$
  $\%x^{\mathsf{Id}} = \mathsf{load}\ \tau,\ \tau*\ \%\mathsf{ptr}$
$\{\ell \mapsto_q^{\mathsf{src}} v * \langle \mathsf{ptr} := \mathsf{Addr}(\ell)\rangle_i^{\mathsf{src}} * \langle \mathsf{x} := \mathsf{Addr}(v)\rangle_i^{\mathsf{src}} * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Local}_i^{\mathsf{src}}\ (\{[\%x]\} \cup L)\}^{\mathsf{src}}$

SourceInstrStore
$\{(v \in \tau) * (v' \in \tau) * (x \notin L) * \ell \mapsto^{\mathsf{src}} v * \langle \mathsf{ptr} := \mathsf{Addr}(\ell)\rangle_i^{\mathsf{src}} * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Local}_i^{\mathsf{src}}\ L\}$
  $\%x^{\mathsf{Void}} = \mathsf{store}\ \tau\ v',\ \tau*\ \%\mathsf{ptr}$
$\{\ell \mapsto_q^{\mathsf{src}} v * \langle \mathsf{ptr} := \mathsf{Addr}(\ell)\rangle_i^{\mathsf{src}} * \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Local}_i^{\mathsf{src}}\ (\{[\%x]\} \cup L)\}^{\mathsf{src}}$

Figure 9.3: Excerpt of Velliris event and instruction rules (source triple rules are symmetric for target triple)

9.2.4 Example: A load-elimination optimization

In order to demonstrate how to use ownership reasoning to prove properties in Velliris, we prove a simple load-elimination example.[26] In the following, we aim to remove a redundant load that loads the value stored in the newly allocated address.

$\{\mathsf{True}\}$

```
%l = alloca i32
%x = store i32 42, i32* %l
ret i32 42
```

$\preceq$

```
%l = alloca i32
%x = store i32 42, i32* %l
%y = load i32, i32* %l
ret i32 %y
```

$\{v_t\ v_s.\ v_t = v_s\}$

[26] To ease the presentation, we elide information about alignment and debugging information in LLVM IR syntax.

The above Hoare quadruple states that under a trivial precondition, we can execute the source program (right) and the optimized target program (left) and get the same return value.

In order to prove the above quadruple, we *focus* on the target and source programs by switching to source and target *triples*, using SourceFocus and TargetFocus (see Figure 9.1). The source and target triples each have unary separation logic rules and are mostly similar, as resource-specific rules are mainly agnostic to whether the resource is used for the source or target program. Focusing is used in conjunction with the sequencing rule (SimBind), so that we can focus on the subexpressions of sequenced programs.

With the focusing triples, we can prove the above example. SourceFocus focuses the quadruple into a source triple, where we can use SourceInstrAlloca to get an allocation at a new address with a block of memory that can fit a value of type `i32`. For now, let us ignore the $\mathsf{Alloca}^{\mathsf{src}}$, $\mathsf{Local}^{\mathsf{src}}$, and $\mathsf{Frame}^{\mathsf{src}}$ predicates. After execution, the source program has full ownership over this location, i.e. we have $\ell_s \mapsto^{\mathsf{src}} \mathsf{new\_block}^{\tau}$. As Velliris is a relational separation logic [Yan07], there are two points-to connectives, each for the source or target program where $\ell_s \mapsto^{\mathsf{src}} v$ refers to the ownership of location ℓ_s at source memory, and $\ell_t \mapsto^{\mathsf{tgt}} v$ for target memory, respectively. The superscript notation src and tgt are used throughout this thesis to distinguish between the source and target resources. $\langle \mathsf{l} := \mathsf{Addr}(\ell)\rangle_i^{\mathsf{src}}$ is an assertion about the local environment, where it denotes that the local id l stores the address value ℓ (ignore the subscript i for now). Then, we can use the SourceInstrStore rule to update the points-to with the value 42 of type `i32`. Subsequently, the SourceInstrLoad rule can be used to get the value 42 onto the location `%y` and the source program finally returns the value 42.

Then, we can use symmetric reasoning for the target program, resulting in $\{\ell_t \mapsto^{\mathsf{tgt}} 42 * \ell_s \mapsto^{\mathsf{src}} 42\}$ `ret i32 42` $\preceq$ `ret i32 42` $\{v_t\ v_s.\ v_t = v_s\}$. We conclude the proof with SimValue with the trivial obligation that 42 is equal to itself.

9.2.5 Stack-local resources in Velliris

In fact, for simplicity, we have omitted one key component in the above verification: the "frame resources" responsible for frame-local states. These resources play an interesting role at function entry and exit and will be discussed further in Section 9.3. For now, we only give their basic ideas needed for a thorough understanding of Figure 9.3.

One key feature of LLVM IR is the stack-allocated memory obtained through alloca instructions. The instruction allocates memory on the stack frame of the currently executing function, which will be automatically freed when this function returns to its caller. In VIR, at function entry, a fresh stack frame containing only the passed arguments in its local environments is created, and at function exit, the local environment and stack-allocated memories are removed. The ghost theory in Velliris deals with this automatic deallocation by keeping track of the stack frame and the associated set of stack-allocated locations.

Frame resources For the source program, the $\mathsf{Frame}^{\mathsf{src}}\ i$ denotes the current frame with index i, $\mathsf{Alloca}_i^{\mathsf{src}}\ S$ is the set of locations S that is stack-allocated at frame i, and $\mathsf{Local}_i^{\mathsf{src}}\ L$ is the local environment L at frame i. For local environment assertions $\langle \mathsf{x} := v \rangle_i^{\mathsf{src}}$ the subscript i denotes the frame that the local stack is associated with. The target program has corresponding resources ($\mathsf{Frame}^{\mathsf{tgt}}\ i$, $\mathsf{Alloca}_i^{\mathsf{tgt}}\ S$, and $\mathsf{Local}_i^{\mathsf{tgt}}\ L$) as well. We use a bundled notation for the frame-relevant resources, with the following definition: $\mathsf{FrameRes}_i^{\mathsf{src}}(A, L) \triangleq \mathsf{Frame}^{\mathsf{src}}\ i * \mathsf{Alloca}_i^{\mathsf{tgt}}\ A * \mathsf{Local}_i^{\mathsf{src}}\ L$ and $\mathsf{FrameRes}_i^{\mathsf{tgt}}(A, L) \triangleq \mathsf{Frame}^{\mathsf{tgt}}\ i * \mathsf{Alloca}_i^{\mathsf{tgt}}\ A * \mathsf{Local}_i^{\mathsf{tgt}}\ L$.

9.3 Relaxed call simulation and semantics for VIR

Velliris supports LLVM IR's attribute specifications, which record the compiler's assumptions about (potentially external) function calls—attributes like *readonly* or *argmemonly* affect which optimizations are applicable because they constrain the assumed behavior of the functions. The specifications are straightforward to encode in Velliris due to its relaxed *call simulation* that we introduce in this section. To implement the relaxed call simulation, we also rectify in the way some deficiencies in Vellvm's handling of external state.

9.3.1 Velliris call simulation: overview

Unlike the prior VIR simulation, the Velliris calls simulation offers the following new features: (1) calls in simulation do not need to be exactly equal, such that related arguments can be passed around to function calls, (2) the simulation allows callees to (mutably, or immutably) borrow resources from the caller, keeping

track of "checked-out" resources, and (3) the simulation is aware of a logical interpretation for *memory attributes*, and controls permission based on the type of attribute decorated on a function call.

The call simulation in Velliris is like a librarian: when a patron (i.e. *a function call*) checks out a book (i.e. *acquiring ownership of a resource*), the librarian updates the system to indicate which book has been borrowed, checks whether the patron should have full access to the material (or a partial scan) (i.e. *permission-based ownership*), and makes sure that all resources have been safely returned.

Pointers can be leaked to function calls through *ownership transfer*. The ownership transfer can be exclusive (i.e. the call has mutable access to the pointer) or fractional (i.e. the call only has read-only access to the pointer). In order to transfer ownership, we leak the resources as a *public bijection* between pointers to express that a location ℓ_t from the target program and ℓ_s from the source program, which stores related bytes to memory. The bureaucratic resource that keeps track of which resource has been leaked to the public is the *checkout set*, which describes how much of a pair of publicly related resources has been "checked out" for use.

9.3.2 Value and Memory Relation

The Velliris call simulation allows *passing related pointers* to function calls, where the pointer address being passed on to the calls on source and target programs may not be exactly equal, but are related by a certain value relation $\mathcal{V}$. In the case of VIR, there are two notions of values (under-defined values and dynamic values), and the values stored in memory is the serialized bytes of the values. [27] Because of this, we need to craft a value relation that will be stable over memory operations, such as reads and writes, along with respecting the notion of defined and under-defined values that are specific to the VIR language.

Dynamic values are the domain of dynamic values that the language can manipulate include 1, 8, 32 and 64 bit integers, memory addresses ($\mathtt{Addr}(a)$), arrays and structs, and poison values , which denote deferred undefined behavior. [28] Under-defined values contain *undef* values, a set semantics model for deferred undefined behaviors. Because we work in a deterministic setting, we show relations between concretized results of under-defined values.

[27] The value relation without subscript is an overloaded notation for both dynamic values and under-defined values.

[28] The memory model used in Velliris does not support storing poison values to memory, which is why do not present a refinement relation over poison values in the value relation. Adapting Velliris to be equipped with a more robust memory model with poison support is left for future work.

Byte relation

$$\mathcal{V}_{\mathsf{SByte}}(\mathsf{Byte}(b_1), \mathsf{Byte}(b_2)) \triangleq b1 = b2$$
$$\mathcal{V}_{\mathsf{SByte}}(\mathsf{Ptr}(a_1), \mathsf{Ptr}(a_2)) \triangleq a_1 \leftrightarrow_h a_2$$
$$\mathcal{V}_{\mathsf{SByte}}(\mathsf{PtrFrag}, \mathsf{PtrFrag}) \triangleq \top$$
$$\mathcal{V}_{\mathsf{SByte}}(\mathsf{SUndef}, \mathsf{SUndef}) \triangleq \top$$

Dynamic value relation

$$\mathcal{V}_{\mathsf{Dyn}}(\mathsf{Addr}(a_1), \mathsf{Addr}(a_2)) \triangleq a_1 \leftrightarrow_h a_2$$
$$\mathcal{V}_{\mathsf{Dyn}}(\texttt{poison}, \texttt{poison}) \triangleq \top$$
$$\mathcal{V}_{\mathsf{Dyn}}(\mathsf{Int}_{sz}(i_1), \mathsf{Int}_{sz}(i_2)) \triangleq i_1 = i_2$$
$$\mathcal{V}_{\mathsf{Dyn}}(\mathsf{Float}(f_1), \mathsf{Float}(f_2)) \triangleq f_1 = f_2$$
$$\mathcal{V}_{\mathsf{Dyn}}(\mathsf{Double}(d_1), \mathsf{Double}(d_2)) \triangleq d_1 = d_2$$
$$\mathcal{V}_{\mathsf{Dyn}}(\mathsf{Array}(l_1), \mathsf{Array}(l_2) \triangleq \forall i, \mathcal{V}_{\mathsf{Dyn}}(l_1[i], l_2[i])$$
$$\mathcal{V}_{\mathsf{Dyn}}(\mathsf{Struct}(l_1), \mathsf{Struct}(l_2) \triangleq \forall i, \mathcal{V}_{\mathsf{Dyn}}(l_1[i], l_2[i])$$

Under-defined value relation

$$\mathcal{V}_{\mathsf{U}}(u_1, u_2) \triangleq \forall dv_2, [\![u_2]\!]_C = \texttt{ret}\ dv_2 \mathrel{-\!\!*} \exists dv_1, [\![u_1]\!]_C = \texttt{ret}\ dv_1 * \mathcal{V}_{\mathsf{Dyn}}(dv_1, dv_2)$$

Memory block relation

$$\mathcal{V}_{\mathsf{Blk}}(m_1, m_2) \triangleq (\forall kv, k[m_1] = v \mathrel{-\!\!*} \exists v', k[m_2] = v' * \mathcal{V}_{\mathsf{SByte}}(v, v')) * (\forall k, k[m_1] = \bot \Rightarrow k[m_2] = \bot)$$

Memory read relation

$$\mathcal{V}^{\rho}_{\mathsf{Blk}}(m_1, m_2) \triangleq (\forall \tau\ o, \mathsf{WF}_{\mathsf{Dyn}\tau}(\tau) \mathrel{-\!\!*} \mathcal{V}_{\mathsf{U}}(m_1[o]_\tau, m_2[o]_\tau))$$

Figure 9.4: Value and Memory relation

Figure 9.4 shows the value relation for bytes, dynamic values, and under-defined values. The byte relation relates literal bytes with exact equality, and address bytes using the bijection relation. Pointer fragments used for alignment and undefined bytes are always related. The dynamic value relation definition is also straightforward; it will relate two dynamic values if they are of the same type and their corresponding elements are related.

For the under-defined values, we should note that the setting of Velliris is deterministic. Thus, the under-defined values at concretization ($[\![-]\!]_C$) will either translate an already concrete value, or return the default value for its given type. Given these definitions, we can define a relation over two logical blocks. The following derived properties of value relations show that the byte relation, uvalue relation, and the dvalue relation are stable over serialization ($-^{\uparrow_{\mathsf{byte}}}$), uvalue-casting ($\uparrow -$), and concretization ($[\![-]\!]_C$).

$$\frac{[\![uv]\!]_C = \texttt{ret}\ dv \quad [\![uv']\!]_C = \texttt{ret}\ dv' \quad \mathcal{V}_{\mathsf{Dyn}}(dv, dv')}{\mathcal{V}_{\mathsf{U}}(uv, uv')}\ \textsc{ValConc} \qquad \frac{\mathcal{V}_{\mathsf{Dyn}}(dv, dv')}{\mathcal{V}_{\mathsf{U}}(\uparrow dv, \uparrow dv')}\ \textsc{UrelLift}$$

$$\frac{\mathcal{V}_{\mathsf{Dyn}}(dv, dv')}{|dv^{\uparrow_{\mathsf{byte}}}| = |dv'^{\uparrow_{\mathsf{byte}}}|}\ \textsc{DrelSerial} \qquad \frac{\mathcal{V}_{\mathsf{Dyn}}(dv, dv')}{\forall n, \mathcal{V}_{\mathsf{SByte}}(dv^{\uparrow_{\mathsf{byte}}}[n], dv'^{\uparrow_{\mathsf{byte}}}[n])}\ \textsc{ByteRelLift}$$

$$\frac{}{\mathcal{V}_{\mathsf{Blk}}(\emptyset, \emptyset)}\ \textsc{Empty} \qquad \frac{}{\mathcal{V}_{\mathsf{Blk}}(\mathsf{NewBlock}_\tau, \mathsf{NewBlock}_\tau)}\ \textsc{NewBlock}$$

$$\frac{\mathcal{V}_{\mathsf{Blk}}(m_1, m_2)}{\mathcal{V}^{\rho}_{\mathsf{Blk}}(m_1, m_2)}\ \textsc{MemRead} \qquad \frac{\mathcal{V}_{\mathsf{Blk}}(b_1, b_s) \quad \mathcal{V}_{\mathsf{Dyn}}(dv_1, dv_2)}{\mathcal{V}_{\mathsf{Blk}}(b_1[o \mapsto dv_1^{\uparrow_{\mathsf{byte}}}], b_2[o \mapsto dv_2^{\uparrow_{\mathsf{byte}}}])}\ \textsc{WriteStable}$$

Figure 9.5: Value and memory relation properties

9.3.2.1 Memory relation

Figure 9.4 shows the memory-relevant relations. The $\mathcal{V}_{\mathsf{Blk}}$ relation relates each byte in memory between source and target. If there is a byte in the source memory, then there must be a corresponding byte in the target memory that is related by the byte relation $\mathcal{V}_{\mathsf{SByte}}$. The $\mathcal{V}^{\rho}_{\mathsf{Blk}}$ relation is the memory *read* relation. LLVM IR is peculiar in that it can read from memory at not only any *offset*, but also at any well-formed *type*. The definition $\mathsf{WF}_{\mathsf{Dyn}\tau}(-)$ formalizes the definition of a well-formed type (a non-empty type), and the memory read relation $\mathcal{V}^{\rho}_{\mathsf{Blk}}$ relates two logical blocks if for all well formed types and offset, the corresponding reads are related by the under-defined value relation $\mathcal{V}_{\mathsf{U}}$.

The value relation $\mathcal{V}$ used to relate two logical blocks is defined as $\mathcal{V}(m_1, m_2) := \mathcal{V}_{\mathsf{Blk}}(m_1^{\mathsf{bytes}}, m_2^{\mathsf{bytes}})$. Now having a well-behaved relation for memory blocks (a list of bytes), we can define the value relation over logical blocks (which is a tuple that keeps track of a memory block and its size).

The derived properties of the block-related relations are described in Figure 9.5. Empty states that the empty memory blocks are related. NewBlock states that a new block with a given type is related to itself. The MemRead rule states that if two memory blocks are related bytewise (by $\mathcal{V}_{\mathsf{Blk}}$), then the corresponding reads are related (by $\mathcal{V}^{\rho}_{\mathsf{Blk}}$). The WriteStable rule states that if two blocks are related bytewise, then writing a related value to the same offset will also be related.

9.3.2.2 Leaking pointers and the location bijection

The notion of *similarity* in values is captured by the value relation $\mathcal{V}$, as described in the previous section. In the case of pointers, the locations must not be exclusively owned so that the callee may mutate memory. We introduce the *public bijection* assertion $\ell_t \leftrightarrow_h \ell_s$ which states that target location ℓ_t and source location ℓ_s are an escaped pair of related locations.

When pointers are passed along a function call, we *leak* the ownership into the public so that the call can modify the location. Depending on the function attribute, we have different specifications on how much information can be leaked, so that we can limit control over how much information the function call has access to.

LocEscape

$$\frac{\{\ell_t \leftrightarrow_h \ell_s * P\}\ e_t \preceq e_s\ \{\Phi\}}{\{\ell_t \mapsto^{\mathsf{tgt}} v_t * \ell_s \mapsto^{\mathsf{src}} v_s * \mathcal{V}(v_t, v_s) * P\}\ e_t \preceq e_s\ \{\Phi\}}$$

SimLoad

$$\{\ell_t \leftrightarrow_h \ell_s * \mathsf{checkout}\ C * ((\ell_t, \ell_s) \notin C \vee C(\ell_t, \ell_s) < 1)\}$$
$$\mathtt{trigger}\ (\mathtt{Load}^{\mathcal{V}_u}(\tau, \mathsf{Addr}(\ell_t))) \preceq \mathtt{trigger}\ (\mathtt{Load}^{\mathcal{V}_u}(\tau, \mathsf{Addr}(\ell_s)))$$
$$\{v_t, v_s. \mathcal{V}(v_t, v_s) * \mathsf{checkout}\ C\}$$

SimStore

$$\{\ell_t \leftrightarrow_h \ell_s * \mathcal{V}(v_t, v_s) * \mathsf{checkout}\ C * (\ell_t, \ell_s) \notin C\}$$
$$\mathtt{trigger}\ (\mathtt{Store}^{()}(\mathsf{Addr}(\ell_t), v_t)) \preceq \mathtt{trigger}\ (\mathtt{Store}^{()}(\mathsf{Addr}(\ell_s), v_s))$$
$$\{v_t, v_s. \mathcal{V}(v_t, v_s) * \mathsf{checkout}\ C\}$$

Figure 9.6: Bijection laws (excerpt.)

9.3.2.3 Checkout set

The *checkout set* checkout C controls how much information has been leaked to the public. It keeps track of all currently leaked (public) locations, and the amount of information leaked through ownership. For instance, if there is a pair of locations (ℓ_t, ℓ_s) in C, and $C(\ell_t, \ell_s) = 1$, the ownership for this pair of locations "has been fully checked out", meaning that the location cannot be accessed through ownership until the ownership is returned to the checkout set. On the other hand, if there is pair of locations (ℓ_t, ℓ_s) in C, and $(\ell_t, \ell_s) \notin C$, the locations are available for checkout (the range of values used in the checkout set is $(0, 1]$, as is standard in fractional ownership). To illustrate the behavior of how loads and stores interact with the checkout set, the excerpt of the rules are shown in Figure 9.7.

The bijection $\ell_t \leftrightarrow_h \ell_s$ is *public* knowledge, and thus it is a duplicable resource. Thus we are able to have knowledge about the locations ℓ_t, ℓ_s after the call. The rest of the local resources can also be retained through using the frame rule (since we are in a separation logic, all proof rules are compatible with framing). In order to take advantage of the framing rule along with the public bijection, we must show the adequacy of our function calls. The simulation relation is an *open* simulation, which allows skipping over calls to the same function in source and target programs. This would only be sound if the function that is skipped over respects the ownership. To ensure the soundness of this rule, the top-level adequacy proof in Section 11.2 assumes that all functions in the program satisfy the simulation relation.

READONLY-CALL

$\{\mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s A_s L_s * \mathsf{checkout}\ C * \overrightarrow{\mathcal{V}}(args_t, args_s) * (\forall(\ell_t, \ell_s) \in C.C(\ell_t, \ell_s) = q \land q < 1)\}$

$\quad \mathsf{call}\ \tau\ f(args_t)\ \mathsf{readonly}\ \preceq\ \mathsf{call}\ \tau\ f(args_s)\ \mathsf{readonly}$

$\{v_t, v_s.\mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s * \mathcal{V}(v_t, v_s) * \mathsf{checkout}\ C\}$

ARGMEMONLY-CALL

$\{\mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s * \mathsf{checkout}\ C * \overrightarrow{\mathcal{V}}(args_t, args_s) * (\forall \ell_t, \ell_s.(\mathsf{Addr}(\ell_t), \mathsf{Addr}(\ell_s)) \in \mathsf{zip}\ args_t\ args_s \mathrel{-\!\!*} (\ell_t, \ell_s) \notin C)\}$

$\quad \mathsf{call}\ \tau\ f(args_t)\ \mathsf{argmemonly}\ \preceq\ \mathsf{call}\ \tau\ f(args_s)\ \mathsf{argmemonly}$

$\{v_t, v_s.\mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s * \mathcal{V}(v_t, v_s) * \mathsf{checkout}\ C\}$

ARGMEMONLY-READONLY-CALL

$\{\mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s * \mathsf{checkout}\ C * \overrightarrow{\mathcal{V}}(args_t, args_s) * (\forall \ell_t, \ell_s, (\mathsf{Addr}(\ell_t), \mathsf{Addr}(\ell_s)) \in \mathsf{zip}\ args_t\ args_s \mathrel{-\!\!*} C(\ell_t, \ell_s) < 1)\}$

$\quad \mathsf{call}\ \tau\ f(args_t)\ \mathsf{argmemonly, readonly}\ \preceq\ \mathsf{call}\ \tau\ f(args_s)\ \mathsf{argmemonly, readonly}$

$\{v_t, v_s.\mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s * \mathcal{V}(v_t, v_s) * \mathsf{checkout}\ C\}$

Figure 9.7: Readonly and argmemonly attribute laws

9.3.3 Call simulation

SIMCALL

$\{\mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s * \mathsf{checkout}\ \emptyset * \overrightarrow{\mathcal{V}}(args_t, args_s)\}$

$\quad \mathsf{call}\ \tau\ f(args_t)\ \preceq\ \mathsf{call}\ \tau\ f(args_s)$

$\{v_t, v_s.\mathsf{Frame}^{\mathsf{tgt}}\ i_t * \mathsf{Frame}^{\mathsf{src}}\ i_s * \mathsf{checkout}\ \emptyset * \mathcal{V}(v_t, v_s)\}$

With the definition of the value relation and location bijection, we can explain the call simulation rule (SIMCALL) of Velliris. Given that we have two calls to the same functions where (1) frame index on the source and target program (2) arguments are pointwise related by the value relation ($\overrightarrow{\mathcal{V}}(args_t, args_s)$, and (3) all public locations in bijection are available for checkout (checkout $\emptyset$), the calls are in simulation. The callee does not affect the caller's frame index, and returns any resources that it might have used from the public (so that the checkout set is $\emptyset$ after the call, again). The resulting values from the call is related to the value relation, as we have called the same function with related arguments.

9.3.3.1 Logical interpretation of memory attributes

In LLVM IR, *function attributes* are behavioral specifications about a function. They are are used to enable certain transformations over functions, where it is assumed to throw undefined behavior if its specification is not met. The usage of attributes can be at the call-site of the function, or can be part of the declaration of the function. Many transformation passes in LLVM IR rely on the attributes to perform certain transformations.

Velliris gives logical specifications for reasoning about memory-relevant attributes. Thanks to the separation logic resources we have in the framework, it is possible and straightforward to define specification over these attributes. Figure 9.7 describes the memory-attribute relevant bijection laws. The readonly attribute is a specification that a function call does not affect memory, thus we can specify this using our checkout set that all public locations have already been at least partially checked out ($\forall(\ell_t, \ell_s) \in C.C(\ell_t, \ell_s) = q \land q < 1$). The argmemonly attribute is a specification that a function call only affects any pointers that were passed as an argument to the function, thus specifying that those locations have not been checked out by any resource. Memory-relevant attributes may also be combined, as in the ARGMEMONLY-READONLY-CALL. Using these attribute specifications, it is possible to prove transformations that use memory-relevant attributes correct.

9.3.4 Example: Store-forwarding across calls

To illustrate an example of a transformation that can occur under a call with function attributes, we show a store-forwarding transformation across a call. Other memory attributes allow for different types of transformations as well, such as load forwarding across calls under readonly attributes.

$$\{\mathsf{FrameRes}^{\mathsf{tgt}}_{i_t}(A_t, L_t) * \mathsf{FrameRes}^{\mathsf{src}}_{i_s}(A_s, L_s) * \mathsf{checkout}\ \emptyset\}$$

```
%l = alloca i32                                  %l = alloca i32
store i32 42, i32* %l                            %m = alloca i32
%m = alloca i32                                  store i32 2, i32* %m
store i32 2, i32* %m                      ⪯      call void @foo(i32* %m) argmemonly
call void @foo(i32* %m) argmemonly               store i32 42, i32* %l
ret i32 %l                                       ret i32 %l
```

$$\{v_t\ v_s.\, v_t = v_s * \mathsf{FrameRes}^{\mathsf{tgt}}_{i_t}(A_t \cup \{[\ell_t, m_t]\}, L_t \cup \{[\%m, \%l]\}) * \mathsf{FrameRes}^{\mathsf{src}}_{i_s}(A_s \cup \{[\ell_s, m_s]\}, L_s \cup \{[\%m, \%l]\}) * \mathsf{checkout}\ \emptyset\}$$

Consider the above example, where a pointer that is stored in local id %m is passed to the function call @foo. The argmemonly attribute specifies that the call should only access the pointer being stored at %m. Thus, its logical specification guarantees that the value allocated at %m will not be affected by the call, and thus the store-forwarding optimization is sound. For proving the above example, we can use the argmemonly-specific call rule (ARGMEMONLY-CALL) to step on both the source and target, and because the checkout set is empty for the location in bijection, we can safely use this rule. We can conclude with our

postcondition because the load resulting from the source program will be the same as the target, since for the target side, the value 42 stored in `%l` remained the same after the call. More precisely, the points-to ownership stating $\ell_t \mapsto^{\text{tgt}} 42$ and $\ell_s \mapsto^{\text{src}} \text{new\_block}^{\text{i32}}$ (where ℓ_t and ℓ_s are the pointers stored at local id `%l` on target and source, respectively) can be framed around the call to `%foo` in conjunction with the ARGMEMONLY-CALL rule. Finally, the value stored in ℓ_s will be updated to 42 using the SOURCEFOCUS and SOURCEINSTRSTORE rule, so both programs return related results.

9.3.5 Extended External Call Semantics for VIR

Here, we present the extension of the original call semantics of VIR that was necessary in order to support expressive reasoning about function calls.

Recall that in VIR, each event is given a semantics through an *event handler*. The type signature of the events constrain the types of the handlers that will concretely implement their semantics. An event handler is a function that takes an input a signature of events and outputs its interpretation into a semantic domain (i.e., a specific monad). For VIR, its semantics is given with event handlers and is given in layers. This is convenient because some effects are implemented in terms of others: memory operations, for instance, may introduce undefined behavior events. However, there is a constraint to forming layers of interpretation: the order of interpretation matters. In general, it is a well-known result that such layered interpretation for monads do not commute, i.e. a state event interpreted before an exception event has a different meaning than an exception event interpreted before state. This comes with a specific limitation with how call events are implemented in the VIR semantics. `VellvmE`, the top-level type of events for LLVM IR programs, denotes the entire sum of events supported by VIR, including exceptions, UB, and calls. Each arrow corresponds to the folding of the relevant event handler (e.g. global event handler) over the program structure. Because the implementation of external calls can be left unknown at the time of interpretation, its interpretation is *delayed* to the later layers of interpretation. Thus, E_3 has remaining syntactic call events in the program. Unfortunately, this naïve interpretation in

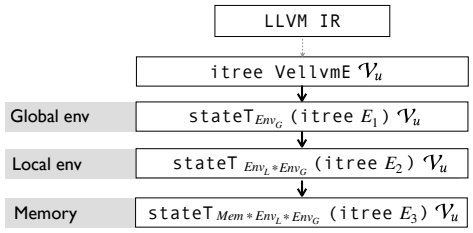

Figure 9.8: Stateful fragment of VIR interpretation

```
Variant stateEff (E : Type -> Type) : Type -> Type :=
| StateEff {X} : S * E X -> stateEff (S * X).

Definition handle_state_eff (L: language) {E F : Type -> Type} `{stateEff E -< F}:
  forall A, E A -> stateT S (itree F) A :=
  fun A (e : E A) (env : state L) => trigger (StateEff (env, e)).

Definition interp_call_state {E F} :
  itree (@stateE S +' E +' F) ~> stateT S (itree (@stateEff E +' F)) :=
  interp_state (case_ h_state (case_ handle_stateEff pure_state)).

Definition CallEvents L := stateEff (state L) (call_events L).

(* Layer(s) of interpretation *)
Definition L0 L := state_events L +' call_events L +' E L.
Definition L1 L := CallEvents L +' E L.
```

Figure 9.9: Event transformer for stateful external call events

the original VIR semantics gives an overly conservative interpretation over calls, where calls to external functions is not aware of the stateful interpretation.

Event transformers: controlling delayed interpretation Using the prior interpretation scheme of VIR, any arbitrary event `E` passed onto `interp_state` would not be able to affect the state. However, realistic calls may affect memory and thus we need a new method of interpretation. An *event transformer* can transform a call event into a new event type that modifies a given state, by expecting an input state and returning an output state. Figure 9.9 shows the implementation of an event transformer `handle_state_eff`. Given an event `E` which may affect state, the event can be transformed into a `stateEff`. Intuitively, `stateEff` interprets away the state-affecting semantics of `E`, while keeping the syntactic marker for the original event. Concretely, it is an event that takes in as input a pair of state and `E` event and outputs a new state and a value that has the same return type as the original `E` event. Note that ITree events are of kind `Type` $\rightarrow$ `Type`, where `e : E A` is a visible external event with an expected output of type `A`. The annotation `stateEff E -< F` indicates that `F` is a sum of events that contains the event `stateEff E`. Using this event transformer, we can directly thread through the state to the external call *without fully interpreting* the external call event.

With this event transformation scheme, we can define a new interpretation layer where call events can affect memory. The first layer `L0` is the uninterpreted layer, while layer `L1` interprets away state events while transforming the call events into a `stateEff` and keeping the remainder of events `E` uninterpreted. This interpretation scheme allows us to define a simulation relation over a partially interpreted language,

$$\frac{e_t \preceq e_s \ \{\Phi\}}{e_t \preceq e_s \ [\ \Phi\ ]}\ \textsc{SimLift} \qquad \frac{e_t \preceq e_s \ [\ \Phi^{\uparrow}\ ]}{e_t \preceq e_s \ \{\Phi\}}\ \textsc{SimRet}$$

$$\frac{\begin{array}{r}\square(\forall \Phi, e_t, e_s. F\ \Phi\ e_t\ e_s \mathrel{-\!\!*} (e_t = \mathsf{tau}\ e_t' * e_s = \mathsf{tau}\ e_s * e_t' \preceq e_s' \ \{((F\ \Phi) \vee_2 \Phi))\} \vee \\ (e_t = \mathsf{vis}\ ev_t\ k_t * e_s = \mathsf{vis}\ ev_s\ k_s * \mathsf{handleEvent}\ (\mathsf{isim}\ ((F\ \Phi) \vee_2 \Phi))\ k_t\ k_s\ ev_t\ ev_s))\end{array}}{F\ \Phi\ e_t\ e_s \mathrel{-\!\!*} e_t \preceq e_s\ [\ \Phi\ ]}\ \textsc{SimPaco}$$

$$\frac{\begin{array}{l}F\ i_t\ i_s \\ \square(\forall i_t, i_s.\ F\ i_t\ i_s \ \mathrel{-\!\!*} \\ \quad f_t\ \ i_t \ \preceq\ f_s\ \ i_s\ \ [\ \lambda e_t, e_s. \\ \qquad (\exists r_t, r_s. e_t = \mathsf{ret}\ (\mathtt{inl}\ r_t) * e_s = \mathsf{ret}\ (\mathtt{inl}\ r_s) * F\ \ i_t\ \ i_s) \\ \qquad \vee (\exists r_t, r_s. e_t = \mathsf{ret}\ (\mathtt{inr}\ r_t) * e_s = \mathsf{ret}\ (\mathtt{inr}\ r_s) * \Phi\ i_t\ \ i_s)\ ])\end{array}}{\mathsf{iter}(f_t, i_t) \preceq \mathsf{iter}(f_s, i_s)\ [\ \Phi\ ]}\ \textsc{SimIter}$$

$$\frac{\begin{array}{c}F\ \ i_t\ \ i_s \qquad\qquad \square(\forall i_t, i_s.\ F\ \ i_t\ \ i_s\ \ \mathrel{-\!\!*}\ x \preceq y\ [\ \lambda e_t, e_s.\ \Phi\ x\ y * \ F\ \ i_t\ \ i_s\ ] \\ \square(\forall i_t, i_s. \vec{\mathcal{V}}(args_t, args_s) * \mathcal{V}(f_t, f_s) \mathrel{-\!\!*} f\ (\mathtt{Call}^{\mathcal{V}_u}(f_t, args_t)) \preceq g\ (\mathtt{Call}^{\mathcal{V}_u}(f_s, args_s))\ [\ \mathcal{V}\ ])\end{array}}{\mathsf{interp\_mrec}\ f\ x \preceq \mathsf{interp\_mrec}\ g\ y\ [\ \Phi\ ]}\ \textsc{SimMrec}$$

Figure 9.10: Coinductive principles in Velliris

and we discuss this simulation in the following section.

9.4 Coinductive Reasoning

The denotation of VIR programs uses ITree-based *combinators* for iteration such as the iteration operator iter and the mutual recursion operator mrec. The top-level mutually-recursive control flow graph, for instance, is defined using the mrec combinator. In this section, we provide the coinductive reasoning principles for these combinators in Velliris, which are used for the proof of the fundamental theorem, contextual refinement theorem (stated in the following section), and the example for loop invariant code motion. Because we have defined our simulation as a Knaster-Tarski fixpoint in the previous section, it is possible to define proof laws about *parametric coinduction* for these combinators within the Velliris logic.

9.4.1 Coinductive principles in Velliris

The coinductive principles in Velliris is described in Figure 11.4. The SimLift and SimRet rules describe the interaction between the two simulation definitions defined in the prior section. SimPaco describes the *parametric coinduction* principle, which states a standard parametric coinduction principle over lock-stepping programs. The SimIter and SimMrec rule allows you to reason about looping programs using standard loop invariant reasoning, instead of writing a manual coinductive proof.

$\mathsf{iter}(f, i)$ denotes an iterating program, where f is the function body and i is the initial argument over

the function body. Given that an initial relational loop invariant F holds over the initial arguments, two iterating bodies are equivalent if they preserve the loop invariant for the next iteration of the loop (the inl case denotes the "continue with the loop" case, while the inr case denotes the "loop exit" case), or satisfy the postcondition if the iteration terminates.

While the SimPaco and SimIter rules are language-independent, the mutual recursion principle is not, as it must be aware of the interpretation of call events. The mutual recursion law SimMrec states that a mutually recursive function over the bodies f and g with initial values x and y can be shown equivalent if an invariant F is preserved over the function bodies. The proof of the mutual recursion law involves a coinductive proof language-specific inversion laws over interpreted results of a given expression.

9.4.2 Example: Reasoning about Loop Invariant Code Motion

We prove a simple loop invariant code motion optimization to illustrate the benefits of the coinductive reasoning principles and separation logic accessible in Velliris. Loop invariant code motion is an optimization algorithm for hoisting expressions out of a loop body if the expression is invariant across the execution of the loop body.

Our simple optimization hoist a load instruction out of a body if the location it is trying to access is disjoint from any of the locations being written to during the execution of the loop body. The specification is straightforward to write as a separation logic statement. Given a denotation function that takes in the predecessor block for the loop (pre) and the loop body block for the loop (body), and if the optimization returns the instruction hoisted and the resulting loop body without it, body′, we show that:

$$[\![(\mathsf{pre},\ \mathsf{body})]\!]_{\mathsf{ocfg}} \preceq [\![(\mathsf{pre} +\!\!+ [\mathsf{hoisted}],\ \mathsf{body}')]\!]_{\mathsf{ocfg}}$$

Establishing the simulation for this optimization is straightforward thanks to the frame rule and the SimIter rule, and requires no explicit coinductive proof.

Chapter 10

Velliris Ghost Theory

In this chapter, we explain the ghost theory in Velliris. Section 10.2.1 explains the VIR state used for Velliris, which maintains a *logical* view of memory. The later sections define the ghost theory for each VIR resource, with a particular focus on how to manage frame-relevant resources. Next, we will explain a notion of stateful *invariants* that is necessary to build an Iris weakest-precondition model. Finally, we will define the invariant for the Velliris using the ghost theory that we have defined.

10.1 Iris resource algebras

In this section, we will introduce the notation and definitions that are used in the rest of the chapter.[29] To use the BI logic of Iris, a resource algebra with a suitable ghost theory must be defined. In Section 8.3.1, we showed an example of how to instantiate a heap as a partial commutative monoid (PCM). Iris provides a library for a more generic notion of resource algebras named *camera*s. We will use generic constructions such as *authoritative* and *agreement* algebras, and *ghost maps* to define the Velliris ghost theory.

10.1.1 Resource algebra

Resource algebras in Iris form a generalized variant of a partial commutative monoid, with a carrier set that is equipped with an step-indexing aware equivalence. In Iris^{light}, this step-indexing metric is not of concern, so we omit the discussion about the carrier element being a metric-aware set (more precisely, a ordered family of equivalences). The presentation here follows the definition of resource algebras in the Semantics lecture notes by Dreyer et al[DSG+22].

A resource algebra M is a tuple $(A, \odot : A \times A \to A, \bar{V} : A \to \mathit{Prop}, |-| : A \to A^?)$. The carrier type A is a set that carries elements of the resource type. The operation $\odot$ is a partial commutative monoid operation over A. The predicate $\bar{V}$ is a meta-level *validity* proposition that describes the well-formedness of the resource. The operation $|-|$ maps resources to its duplicable fragment.

[29] For a more detailed explanation of Iris resource algebras, refer to Jung et al [JKJ+18].

Definition 6. A resource algebra is a tuple $(A, \odot : A \times A \to A, \bar{V} : A \to \mathit{Prop}, |-| : A \to A^?)$ satisfying:

$$
\begin{aligned}
m_1 \odot m_2 &= m_2 \odot m_1 && \text{(commutativity)}\\
(m_1 \odot m_2) \odot m_3 &= m_1 \odot (m_2 \odot m_3) && \text{(associativity)}\\
|m| \in A &\Rightarrow |m| \odot m = m && \text{(core–identity)}\\
|m| \in A &\Rightarrow ||m|| = |m| && \text{(core–idem)}\\
|m| \in A \land m \leq n &\Rightarrow |n| \in A \land |m| = |n| && \text{(core–mono)}\\
m \odot n \in \bar{V} &\Rightarrow a \in \bar{V} && \text{(valid–op)}\\
\textit{where} \quad A^? &:= A \uplus \{\bot\} \qquad x \odot \bot := \bot \odot x := x\\
m \leq n &:= \exists c \in A. n = m \odot c && \text{(incl)}
\end{aligned}
$$

Definition 7. A frame-preserving update $a \rightsquigarrow B$ is an operation where given $a \in A$ and $B \subseteq A$, if $\forall x_f \in A^?. a \odot x_f \in \bar{V} \Rightarrow \exists b \in B. b \odot x_f \in \bar{V}$. An update from an element a to b is defined as $a \rightsquigarrow b := a \rightsquigarrow \{b\}$.

Given this definition of a resource algebra and frame-preserving updates, here are some common resource algebras that are useful.

10.1.2 Exclusive algebra

Exclusive algebras represents a resource that can only be owned by one party. Its carrier type is $Ex(X) := \mathsf{ex}(x : X) \mid \lightning$, where it denotes the exclusive elements ex or the invalid element $\lightning$. The operations are defined as:

$$
a \odot b := \lightning \qquad a \in \bar{\mathcal{V}} := a \neq \lightning \qquad |a| := \bot
$$

$$
\forall x, y. \mathsf{ex}(x : X) \rightsquigarrow \mathsf{ex}(y : X) \qquad\qquad a \leq b \Leftrightarrow b = \lightning
$$

This resource is similar to the resource algebra on the heap $\ell \mapsto v$ described earlier, as the points-to assertions can be updated freely and cannot be owned by multiple entities.

10.1.3 Agreement algebra

The agreement algebra is useful for describing a read-only resource that can be shared by multiple parties. Its carrier type are finite, non-empty sets of elements,

$$Ag(X) := \{A \in 2^X | A \textit{ finite, non-empty}\} \qquad \mathrm{ag}(x) := \{x\}.$$

The operations are:

$$A \odot B := A \cup B \qquad A \in \bar{V} := \exists x. A = \{x\} \qquad |A| := A$$

$$\mathrm{ag}(x) \rightsquigarrow \mathrm{ag}(y) \Leftrightarrow x = y \qquad A \leq B \Leftrightarrow x = y$$

Agreement algebras can be duplicated freely, but its state cannot be changed.

10.1.4 Authoritative algebra

The authoritative resource algebra $Auth(M)$ is one of the most common resource algebras. This algebra contains authoritative $\bullet(a)$ and fragment $\circ(b)$ elements.

Its carrier type and operations are defined by :

$$Auth(M) := Ex(M)^? \times M \qquad \bullet a := (\mathrm{ex}(a), \epsilon_M) \qquad \circ b := (\bot, b)$$
$$(x, a) \odot (y, b) := (x \odot y, a \odot b) \qquad \bar{V} := \{(\bot, b) | b \in \bar{V}_M\} \cup \{(\mathrm{ex}(a), b) | b \leq_M a \wedge a \in \bar{V}_M\}$$
$$\epsilon_M := (\bot, \epsilon) \qquad |(x, a)| := (\bot, |a|)$$

The authoritative element has control over the fragment elements; at any point, fragment elements must be included in the authoritative element. Any updates must be done through the authoritative element. One can view fragment elements as read-only elements of a resource, where the read-only view can be changed only if the authoritative portion is updated alongside it. Given these definitions, we can now define the Velliris resource algebra.

10.2 Velliris resource algebra

The novel part of Velliris's ghost theory come from its (1) frame resources, which must manage automatic deallocations of frame-allocated locations and maintain an invariant over frames, and (2) serialized dynamic values, which incur a particular value relation that is stable over the memory relation invariant that we have seen in Section 9.3.2. For the bijection between source and target resources (in Section), the Velliris ghost theory follows the style of resource algebra in Simuliris[GSS+22], an Iris model for program simulations.

10.2.1 VIR State

The following constructs define the types needed to represent the VIR state.

$$
\begin{aligned}
\mathit{Global} &::= \mathit{id} \hookrightarrow \mathrm{dvalue} \\
\mathit{Local} &::= \mathit{id} \hookrightarrow \mathrm{uvalue} \\
\mathit{Frame} &::= \mathsf{list}\ \mathrm{Addr} \\
\mathit{LocalStack} &::= \mathsf{list}\ \mathit{Local} \\
\mathit{FrameStack} &::= \mathsf{list}\ \mathit{Frame} \\
\mathrm{Allocated} &::= \mathsf{list}\ \mathrm{Addr} \\
\mathit{Mem} &::= \mathrm{Allocated} * (\mathrm{Addr} \hookrightarrow \mathsf{list}\ \mathrm{byte}) \\
\mathrm{byte} &::= \mathsf{SUndef} \mid \mathsf{Byte}\ \mathcal{Z} \mid \mathsf{Ptr}\ \mathrm{Addr} \mid \mathsf{PtrFrag} \\
\\
\mathit{id} &::= \mathit{string} \\
\mathrm{Addr} &::= \mathcal{Z} * \mathcal{Z}
\end{aligned}
$$

The global state is a read-only environment, a map that is instatiated only once at initialization, mapping *id*s to dvalues. The local state corresponds to the stateful component of resources that live within the scope of a function. Because LLVM IR automatically deallocates all locally allocated addresses upon function return, the local state must keep track of the list of addresses allocated at the current frame. Thus, the local state consists of a local environment (*Local*) that maps local *id*s to uvalues, and also a list of addresses (*Frame*) allocated at the current frame. Because a program consists of multiple functions and thus will have multiple frames throughout its execution, it keeps track of a stack of local environments (*LocalStack*) and local frames (*FrameStack*).

We work with a logical view of VIR memory: a map between addresses with an offset (a pair of integers $\mathcal{Z}$), which then points to a list of bytes. The representation of bytes contains an undefined byte (SUndef) for initialized bytes in memory, literal bytes (Byte), a pointer byte for locations that store an address (Ptr), and a pointer fragment that is used for alignment of pointer bytes in memory (PtrFrag). The memory also keeps track of a monotonically increasing set of addresses, in order to prevent reuse of addresses (e.g. an address cannot be used for a new allocation if it has been allocated before, even if the byte in the address has been freed).

Remark. Note that we discard the physical view of VIR memory in this section. The study of robust memory models of LLVM IR is a subject of ongoing research. The VIR memory presented in Section 3.3.3 is a quasi-concrete memory model that supports casting between integers and pointers. Supporting integer to pointer casting in Velliris remains as future work, along with adapting it to other advanced memory models of LLVM IR, such as the twin-memory model[LHJ+18a], and on-going work on robust support of out-of-memory errors.

10.2.2 Resource algebra

The resource algebra of Velliris contains seven main algebras, which are split into frame-relevant and frame-irrelevant resources. Frame-irrelevant resources (i.e. "global" resources) include *Heap*, *Alloc*, and *Global*.

$$\begin{aligned} \mathit{Heap} &:= \mathit{Auth}(\mathit{Loc} \xrightarrow{\text{fin}} (\mathit{Frac} \times \mathit{Ag}(\mathit{Val}))) \\ \mathit{Alloc} &:= \mathit{Auth}(\mathit{Loc} \xrightarrow{\text{fin}} (\mathit{Ag}(\mathcal{N}^?)) \\ \mathit{Global} &:= \mathit{Ag}(\mathit{id} \xrightarrow{\text{fin}} \mathcal{V}) \end{aligned}$$

Heap controls the contents of the memory, with support for fractional ownership. *Alloc* controls the size of the allocated blocks: it tracks either the size of the block, or $\bot$ if the block has been deallocated. *Global* controls the environment of global variables.

The frame-relevant (i.e. "local" resources) are *FrameStack*, *FrameAlloc*, *LocalEnv*, *LocalDomain*, and *LocalStack*. Keeping track of the frame-relevant resources and the allocated set of addresses at the current frame is important to facilitate automatic deallocation of frame-allocated resources at function return.

$$\begin{aligned}
\mathit{FrameStack} &:= \mathit{Auth}(\wp^{\mathsf{fin}}(\mathsf{frame}^{\Gamma})) \\
\mathit{LocalStack} &:= \mathit{Auth}(\wp^{\mathsf{fin}}(\wp^{\mathsf{fin}}(\mathit{id} \xrightarrow{\mathsf{fin}} \mathcal{V}_u))) \\
\mathit{FrameAlloc} &:= \mathit{Auth}(\wp^{\mathsf{fin}}(\mathcal{Z})) \\
\mathit{LocalEnv} &:= \mathit{Auth}(\mathit{id} \xrightarrow{\mathsf{fin}} \mathcal{V}_u) \\
\mathit{LocalDomain} &:= \mathit{Auth}(\wp^{\mathsf{fin}}(\mathit{id}))
\end{aligned}$$

First are the resources that keep track of the entire stack of frames. *FrameStack* keeps track of the stack of frames, and the names associated with each frame on the stack. Each frame keeps track of the index, name of the local environment, local domain, and the frame-allocated set. The *LocalStack* keeps track of the stack of local environments corresponding to each stack frame. Then, there are resources that pertain to a particular frame. The current frame is defined as the head of the frame stack. *FrameAlloc* controls the allocated locations at the current frame. *LocalEnv* controls the current local environment, along with *LocalDomain* keeping track of the domain of the local environment at the current frame.

Instantiating the global resource functor, and ghost namespaces. Using Iris's global resource functor, we register these algebras. Instances of these algebras are need for both the source and target programs, and thus instances are parameterized by the corresponding tuple of ghost names $\gamma_{\mathsf{heap}} = (\gamma_{\mathsf{mem}}, \gamma_{\mathsf{blocksize}}, \gamma_{\mathsf{globals}}, \gamma_{\mathsf{stack}})$. Each of the frames have an associated natural number index and tuple of names as well, expressed through frame^{Γ}, which is a tuple $(\mathcal{N}, \gamma_{\mathsf{local}}, \gamma_{\mathsf{domain}}, \gamma_{\mathsf{alloca}})$, keeping track of the index and the name for the local environment, local domain, and frame-allocated addresses.

With these ingredients ready, we can define the logical heap interpretation.

$$\begin{aligned}
&\mathsf{heapCtx}_{\gamma_{\mathsf{heap}}}(\sigma) := \\
&\quad \exists i\ F, \boxed{\bullet(\sigma^{\mathrm{mem}})}^{\gamma_{\mathsf{mem}}} * \boxed{\bullet(F)}^{\gamma_{\mathsf{blocksize}}} * \boxed{\mathsf{ag}(\sigma^{\mathrm{global}})}^{\gamma_{\mathsf{global}}} * \boxed{\bullet(i)}^{\gamma_{\mathsf{stack}}} \\
&\quad * \boxed{\bullet(\sigma^{\mathrm{local}})}^{i^{\mathrm{local}}_{\mathrm{current}}} * \boxed{\bullet(\mathsf{dom}(\sigma^{\mathrm{local}}))}^{i^{\mathrm{domain}}_{\mathrm{current}}} * \boxed{\bullet(\mathsf{peek}\ \sigma^{\mathrm{alloca}})}^{i^{\mathrm{alloca}}_{\mathrm{current}}} \\
&\quad * \mathop{\Large *}_{(\mathrm{pop}\ i, \sigma^{\mathrm{stack}})[i] \mapsto (f, env)} \left(\boxed{\bullet(env)}^{f^{\mathrm{local}}} * \boxed{\bullet(\mathsf{dom}(env))}^{f^{\mathrm{domain}}} * \boxed{\bullet(\sigma^{\mathrm{frame}}[i+1])}^{f^{\mathrm{alloca}}} \right) \\
&\quad * \mathsf{allocSzRel}(\sigma, F) * \mathsf{stateWF}(\sigma, i)
\end{aligned}$$

heapCtx asserts ownership of ghost states at all ghost names. It keeps track of each of the authoritative

elements for the memory and allocation size control, and the resources at each frame. stateWf ensures that the allocated blocks are also present in the memory block, and that there is no duplication of names in the local environments. allocSzRel ensures that the physical allocation size agrees with what is recorded in the ghost state. stateWF and allocSzRel have the following definitions:

$$\begin{aligned}\mathsf{stateWF}(\sigma, i) :=& \mathsf{NoDup}(\mathit{dom}(\sigma^{\mathrm{local}})) * \mathsf{NoDup}(\sigma^{\mathrm{frame}}) * \mathit{dom}(\sigma^{\mathrm{mem}}) = \sigma^{\mathrm{frame}} \\ & * |i| = |\sigma^{\mathrm{frame}}| * (\forall n, n < |i| \Rightarrow i[n]^{\mathrm{index}} = |i| - n - 1) \\ \mathsf{allocSzRel}(\sigma, F) :=& (\forall b\ o, (b, o) \in F \Rightarrow b \in \mathit{dom}(\sigma^{\mathrm{mem}})) * (\forall b, b \in \mathit{dom}(\sigma^{\mathrm{mem}}) \Rightarrow \exists o, (b, o) \in F)\end{aligned}$$

10.3 Bijection ghost state

10.3.1 Memory bijection

The ghost states for the source and target programs are instantiated using ghost names $\gamma^{\mathrm{src}}_{\mathrm{heap}} = (\gamma^{\mathrm{src}}_{\mathrm{mem}}, \gamma^{\mathrm{src}}_{\mathrm{alloc}}, \gamma^{\mathrm{src}}_{\mathrm{global}})$ and $\gamma^{\mathrm{tgt}}_{\mathrm{heap}} = (\gamma^{\mathrm{tgt}}_{\mathrm{mem}}, \gamma^{\mathrm{tgt}}_{\mathrm{alloc}}, \gamma^{\mathrm{tgt}}_{\mathrm{global}})$. The logical assertions for Velliris programs are defined in Figure 10.1.

Note that there is lifting of dynamic values to logical blocks through the $\uparrow^{blk}$ operator, as LLVM IR instructions operate over typed values instead of their corresponding serialized bytes that would be stored to memory. $\uparrow^{blk}$ is an operation that serializes a dynamic value to a list of bytes. There is also a corresponding inverse operator $\downarrow^{blk}$ that deserializes a list of bytes to a dynamic value. The round trip property holds for any given dynamic value, where $\downarrow^{blk} (\uparrow^{blk} v) = v$.

We maintain a bijection between the source and target blocks. The algebra used on blocks is a finite bijection on blocks (the *GsetBij* algebra of Iris), with the ghost name $\gamma_{\leftrightarrow}$.

$$\mathsf{heapbij}(L, C) := \boxed{\bullet(L)}^{\gamma_{\leftrightarrow}} * \mathop{\Large *}_{(b_t, b_s) \in L} \mathsf{AllocRel}(b_t, b_s, C)$$

The bijection asserts that the blocks will have the same allocation state (i.e. they will be either allocated with the same size on both source and target, or not be allocated). Additionally, it describes whether a pair of locations that are in bijection are leaked to the public.

Memory and global resources

$$\ell_s \overset{q}{\mapsto}_{src} [b_s] := \boxed{\circ[\ell_s \leftarrow (q, b_s)]}^{\gamma^{\text{src}}_{\text{mem}}}$$

$$\ell_s \overset{q}{\mapsto}_{src} v_s := \boxed{\circ[\ell_s \leftarrow (q, v_s^{\uparrow blk})]}^{\gamma^{\text{src}}_{\text{mem}}}$$

$$\ell_s \mapsto_{src} v_s := \ell_s \overset{1}{\mapsto}_{src} v_s$$

$$\ell_s \mapsto_{src} [b_s] := \ell_s \overset{1}{\mapsto}_{src} [b_s]$$

$$\text{AllocSz}_{\text{src}}\ \ell_s\ n := \boxed{\circ[\ell_s \leftarrow (1, \text{some}(n))]}^{\gamma^{\text{src}}_{\text{blocksize}}}$$

$$\text{Global}_{\text{src}}\ A := \boxed{\text{ag}(A)}^{\gamma^{\text{src}}_{\text{global}}}$$

$$\ell_t \overset{q}{\mapsto}_{tgt} [b_t] := \boxed{\circ[\ell_t \leftarrow (q, b_t)]}^{\gamma^{\text{tgt}}_{\text{mem}}}$$

$$\ell_t \overset{q}{\mapsto}_{tgt} v_t := \boxed{\circ[\ell_t \leftarrow (q, v_t^{\uparrow blk})]}^{\gamma^{\text{tgt}}_{\text{mem}}}$$

$$\ell_t \mapsto_{tgt} v_t := \ell_t \overset{1}{\mapsto}_{tgt} v_t$$

$$\ell_t \mapsto_{tgt} [b_t] := \ell_t \overset{1}{\mapsto}_{tgt} [b_t]$$

$$\text{AllocSz}_{\text{tgt}}\ \ell_t\ n := \boxed{\circ[\ell_t \leftarrow (1, \text{some}(n))]}^{\gamma^{\text{tgt}}_{\text{blocksize}}}$$

$$\text{Global}_{\text{tgt}}\ A := \boxed{\text{ag}(A)}^{\gamma^{\text{tgt}}_{\text{global}}}$$

Frame resources

$$\text{Frame}_{\text{src}}\ i := \boxed{\circ(i)}^{\gamma^{\text{src}}_{\text{heap}}}$$

$$[\ell_s := v_s]_{\text{src}}\ i := \boxed{\circ[\ell_s \leftarrow (1, v_s)]}^{\gamma^{\text{src}}_{\text{heap}}, i^{\text{local}}_{\text{current}}}$$

$$\text{LocalDomain}^i_{\text{src}}\ L := \boxed{\text{ag}(A)}^{\gamma^{\text{src}}_{\text{heap}}, i^{\text{domain}}_{\text{current}}}$$

$$\text{Alloca}^i_{\text{src}}\ A := \boxed{\circ(A)}^{\gamma^{\text{src}}_{\text{heap}}, i^{\text{alloc}}_{\text{current}}}$$

$$\text{Frame}_{\text{tgt}}\ i := \boxed{\circ(i)}^{\gamma^{\text{tgt}}_{\text{heap}}}$$

$$[\ell_t := v_t]_{\text{tgt}}\ i := \boxed{\circ[\ell_s \leftarrow (1, v_t)]}^{\gamma^{\text{tgt}}_{\text{heap}}, i^{\text{local}}_{\text{current}}}$$

$$\text{LocalDomain}^i_{\text{tgt}}\ L := \boxed{\text{ag}(A)}^{\gamma^{\text{tgt}}_{\text{heap}}, i^{\text{domain}}_{\text{current}}}$$

$$\text{Alloca}^i_{\text{tgt}}\ A := \boxed{\circ(A)}^{\gamma^{\text{tgt}}_{\text{heap}}, i^{\text{alloc}}_{\text{current}}}$$

Figure 10.1: Bijection ghost state

$$\begin{aligned}
&\text{AllocRel}(\ell_t, \ell_s, C) := \\
&\quad (\text{AllocSz}_{\text{src}}\ \ell_s\ \bot * \text{AllocSz}_{\text{tgt}}\ \ell_t\ \bot) \lor \\
&\quad (\exists\ n, b_s, b_t.\text{AllocSz}_{\text{src}}\ \ell_s\ (\text{some}(n)) * \text{AllocSz}_{\text{tgt}}\ \ell_t\ (\text{some}(n)) * \\
&\qquad (((\ell_t, \ell_s), \text{some}(1)) \in C) \lor \\
&\qquad (((\ell_t, \ell_s), \bot) \in C * \ell_s \mapsto_{\text{src}} [b_s] * \ell_t \mapsto_{\text{tgt}} [b_t] * \mathcal{V}\ b_t\ b_s) \\
&\qquad (\exists q, q < 1 * (((\ell_t, \ell_s), \text{some}(q)) \in C * \ell_s \overset{1-q}{\mapsto}_{\text{src}} [b_s] * \ell_t \overset{1-q}{\mapsto}_{\text{tgt}} [b_t] * \mathcal{V}\ b_t\ b_s) \lor)
\end{aligned}$$

AllocRel includes extra case analysis on the fractional leaking of public resources. The bijection assertion for locations is

$$(b_t, i_t) \leftrightarrow_h (b_s, i_s) := \boxed{\circ\{(b_t, b_s)\}}^{\gamma_{\leftrightarrow}} * i_t = i_s$$

where the blocks are in bijection and the offsets are equal.

10.4 State interpretation

We finally have all the definitions in place to define the state interpretation of Velliris.

$$\mathcal{S}(\sigma_t, \sigma_s) := \exists C, S.\mathsf{heapbij}(C, S) * \mathsf{heapCtx}_{\gamma^{\mathsf{src}}_{\mathsf{heap}}}(\sigma_s) * \mathsf{heapCtx}_{\gamma^{\mathsf{tgt}}_{\mathsf{heap}}}(\sigma_t) * \boxed{\bullet(C)}^{\mathsf{checkedout}} * \mathsf{globalBij}$$

The state interpretation connects the ghost states to the physical resources σ_t and σ_s. It keeps track of the current local and public locations, carrying around the authoritative view of VIR ghost resources. This state interpretation is used to define the weakest pre-condition in Chapter 11, and this use of state interpretation is typical of Iris Hoare triples.

Chapter 11

Weakest-precondition Model and Adequacy

In this chapter, we explain the Iris model used to instantiate a simulation with Interaction Trees. The coinductive machinery is akin to Simuliris and the parametric coinduction library ("paco") used in the ITree library. However, we use these constructions of fixpoints to construct our Velliris simulation within the Iris logic.

First, we give an introductory text on fixpoints, with standard definitions of a Knaster-Tarski least and greatest fixpoint. This will be the necessary background to understand how to define greatest and least fixpoints in the Iris logic. More precisely, the simulation defined for Interaction Trees is a mixed least-greatest fixpoint, which uses parameterized coinduction instead of step-indexing to reason about recursive programs.

11.1 Knaster-Tarski fixpoints in bi_{Iris}

The Bunched Implication (BI) logic of Iris is an embedded logic within the type theory of Coq (a variant of the Calculus of (co-)Inductive Constructions).

The Iris logic is a *bunched* logic, in that it is a substructural logic for reasoning about resources using the separating conjuction($*$) and magic wand operator ($-\!\!*$). This portion of the logic forms the *resource algebra* for separation logic. A nicety of the logic is that modeling *recursion* is an orthogonal aspect: it is typically constructed using a *step-indexed logical relation* defined over a small-step semantics. Because the issue of modeling *recursion* is orthogonal to using the resource fragment of the logic, it is not necessary to inherit both aspects to enjoy the benefits of the framework.

In fact, we use a different model of fixpoints over Interaction Trees in this setting, while inheriting the resourceful fragment of the logic. In this section, we explain the least and greatest fixed point construction that we use for our model.

11.1.1 Least and greatest fixed point construction à la Knaster-Tarski

We can build *least* and *greatest* fixpoints, derived from the Knaster-Tarski theorem, in the Iris logic. Propositions in the logic of Iris (bi_{Iris}) form a complete lattice, in a analogous manner to how propositions in Coq form a complete lattice. This fixed point construction is the one used in this dissertation, and we compare it with the Banach construction in the next subsection.

11.1.1.1 Background: Least and greatest fixed points and the Knaster-Tarski Theorem

A *complete lattice* is a partially ordered set (L ,≤), where all subsets $A \subseteq L$ have a join (least upper bound) and meet (greatest lower bound) defined. A monotone function $f \in L \longrightarrow L$ over this set is an *order-preserving* function. Given a monotone function f on L, a *pre-fixed point* of f is a set $r \subseteq L$ such that $f(r) \leq r$. A *post-fixed point* of f is a set $r \subseteq L$ such that $r \leq f(r)$. We denote μf for the *least fixed point* of f, which is the smallest pre-fixpoint of f. Also, we use νf for the *greatest fixed point* of f, which is the greatest post-fixpoint of f. Knaster-Tarski's theorem states that given a monotone function on a complete lattice, there exists a unique least fixed point and greatest fixed point.

11.1.1.2 Knaster-Tarski fixpoint construction in bi_{Iris}

Propositions in the logic of Iris (bi_{Iris}) form a complete lattice within itself, with the following definition.

Definition 8. The set of affine bi_{Iris} propositions iProp form a complete lattice with the following construction. The ordering between two elements x and y of iProp is $\Box(x \mathrel{-\!\!*} y)$. The supremum of the set A is $\bigvee A := \exists a, (a \in A) * a$, (i.e. there exists some proposition in the set that holds), and the infimum is $\bigwedge A := \forall a, (a \in A) \mathrel{-\!\!*} a$, (i.e. all propositions in the set hold). The cup is the $\vee$ operator and $\wedge$ operator defined in the logic, and the bottom and top element can be given as the $\top$ and $\bot$ propositions that are embedded from the Coq logic.

Given this complete lattice, a least and greatest fixpoint can be defined within the bi logic.

Definition 9. The least fixpoint $\mu_{\text{bi}} f$ is $\forall \Phi,\ \Box(\forall\ x,\ f\ \Phi\ x \mathrel{-\!\!*} \Phi\ x) \mathrel{-\!\!*} \Phi\ x$.

Definition 10. The greatest fixpoint $\nu_{\text{bi}} f$ is $\exists \Phi, \Box(\forall\ x, \Phi\ x \mathrel{-\!\!*} f\ \Phi\ x) * \Phi\ x$.

We use these least and greatest fixpoint definitions in order to instantiate our mixed fixpoint simulation in Section 11.2. In order to use this fixpoint construction, we define a new *weakest precondition* model of Iris, which is a mixed least-greatest fixpoint simulation between two ITrees.

A noteworthy remark is that in this setting, we do not use the later modality in order to model recursion. Essentially, using this Knaster-Tarski fixpoint, we can regard our logic as an "Iris light", Iris^{light} ("Iris without the step-indexing"), where we keep the basic logic of bunched implications [OP99] and the persistence modality, but we ignore all the step-indexing aspects of Iris (e.g., the later modality , guarded recursion, and impredicative invariants). Thus, we fix the step-index to be zero in this setting. Simuliris [GSS+22] uses the same notion of fixpoints as this setting, and was the inspiration for using the Iris logic in our setting. Another benefit of using Knaster-Tarski fixpoints is that there are well-developed proof techniques for coinductive simulations (which are by definition over Knaster-Tarski fixpoints), such as parameterized coinduction.

11.1.2 Remark: Guarded fixed point construction à la Banach: step-indexed logics and the later modality

We make a brief remark regarding the Knaster-Tarski fixpoint and the fixpoint construction typically used within the Iris logic. The fixpoint construction in the Iris logic, which is used for the standard *weakest precondition model* is the fixpoint derived from the Banach fixed point theorem. This technique allows users of Iris to define a program semantics as a small-step relation, and then reason about recursion and higher-order state using a step modality.

The standard carrier type of the Iris logic is an ordered family of equivalences (o.f.e.'s). similarly presented as the construction by Di Gianantonio and Miculan [GM02]. It is possible to define fixpoints over *complete* ordered family of equivalences, where a fixpoint may be given by Banach's Theorem. The fixpoint generated à la Banach will introduce step-indexing modalities into the equation, which would introduce extra bureaucracy to keep track of in the midst of coinductive proofs.

11.2 Simulation Relation

The Velliris simulation allows us to prove various refinements over programs correct inside a separation logic. We first see how the relational Hoare quadruples are defined in this section, and then we give an idea of the key lemma that enables the adequacy proof of the logical relation in Section 11.3. The definitions are defined in a language-generic way, and specifically for an arbitrary Interaction Tree given a stateful interpretation, such that the proof effort does not need to be repeated.

The simulation relation $\{P\}\ e_t \preceq e_s\ \{Q\}$ is at the core of Velliris. Velliris simulations defined inside a separation logic, and specifically in Iris^{light}, as described above.

In the style of Iris weakest preconditions, the Velliris simulation is defined as:

$$\{P\}\ e_t \preceq e_s\ \{Q\} :=\ \Box(P \mathrel{-\!\!*} \mathsf{sim}\ e_t\ e_s\ Q^{\uparrow})$$

where $\mathsf{sim}\ e_t\ e_s\ Q$) is the weakest precondition that we need to impose such that e_s simulates e_t with postcondition Q. The persistence modality $\Box$ ensures that the simulation relation is duplicable (i.e. the fact that e_s simulates e_t can be used arbitrarily many times). We define two variants of hoare triples using sim $e_t \preceq e_s[Q] := \mathsf{sim}\ e_t\ e_s\ Q$, where Q is a relation over ITrees and $e_t \preceq e_s\{Q\} := \mathsf{sim}\ e_t\ e_s\ Q^{\uparrow}$, where Q is a relation over returned values. The $\uparrow$ is a lifting that lifts a relation over values to a relation over ITrees: $Q^{\uparrow}(t,s) := \exists\ \sigma_t, \sigma_s, v_t, v_s.t = \mathsf{ret}\ (\sigma_t, v_t) * s = \mathsf{ret}\ (\sigma_s, v_s) * \mathcal{S}(\sigma_t, \sigma_s) * Q(v_t, v_s)$.

To write Hoare triples, we use the lifted relation $e_t \preceq e_s\{Q\}$ with Q as a relation over returned *values*. The relation $e_t \preceq e_s[Q]$ is useful for writing *coinductive* proofs, as we'll describe some of its properties in Section 11.2.4.

11.2.1 Velliris model: sim definition

In this subsection, we give the definition of the sim simulation that at the core of Velliris Hoare triples.

sim is a stateful simulation over programs that maintains a stateful invariant at each lock-step of the simulation.

$$\mathsf{sim}\ e_t\ e_s\ Q := \forall\ \sigma_t, \sigma_s.\ \mathcal{S}(\sigma_t, \sigma_s) \Rrightarrow \mathsf{isim}\ Q\ (\llbracket e_t \rrbracket\ \sigma_t)(\llbracket e_s \rrbracket\ \sigma_s)$$

Intuitively, this definition says that given two states σ_t and σ_s that satisfy a state invariant $\mathcal{S}$, the interpretation of e_t and e_s with initial states σ_t and σ_s will maintain the inner simulation relation isim with the postcondition Q. The inner simmulation relation isim is a mixed least-greatest fixpoint over ITrees that we will discuss in the next subsection.

e_t and e_s are ITrees at layer L_0 (i.e. e_t and e_s are of type itree L_0 R_1 and itree L_0 R_2). The function $[\![-]\!]$: itree L_0 $R \to \mathsf{stateT}_S$ (itree L_2) R is a stateful *interpretation* function that interprets all stateful events in layer L_0. We remind the readers the definition of a state transformer applied to an itree E, which is a state-threading function, where stateT_S (itree E) $:= S \to$ itree E $(S \times R)$.

The relation $\mathcal{S}$: state $\to$ state $\to$ iProp (iProp being a proposition residing in the Iris^{light} logic) is the *state invariant* relation that is maintained at each step of the simulation. More precisely, the sim relation requires that, given two states that maintain the state invariant $\mathcal{S}(\sigma_t, \sigma_s)$, the interpretation of e_t and e_s will maintain the simulation relation isim. The definition of the simulation is parameterized both by the interpretation function $[\![-]\!]$ and state invariant relation $\mathcal{S}$, so the user is free to choose its definition when instantiating the simulation for a given language.

The concrete definition of the state interpretation invariant for Velliris ($\mathcal{S}_{\mathsf{VIR}}$), which is specific to the VIR language, is given in Chapter 10 when we discuss the ghost theory of Velliris.

11.2.2 isim definition

We define the *internal simulation relation* isim over a stateful program that returns a state and a value. In order to define the isim, we need to define a functor isimF that we will take the mixed least-greatest fixpoint over. The definition of isimF is given in Figure 11.1.

The internal simulation functor isimF is a function that takes as argument a function for defining its greatest fixpoint ($\mathcal{G}$), a function for defining its least fixpoint ($\mathcal{L}$), a postcondition for expressions (Φ), and a source and target expression (s_t, s_s). In general, there are eight cases to consider for the simulation: the *base step* case, *visible step* case, *locking silent step* case, the two *stutter* cases, the *source UB* case, the *source error* case, and the *mismatched* case.

The base case is where the postcondition can be satisfied by the current state of the program, and we can conclude by stating the postcondition Φ. In the visible step case, the greatest fixpoint $\mathcal{G}$ is called on all

$\mathsf{isimF}\ \mathcal{G}\ \mathcal{L}\ \Phi\ s_t\ s_s \triangleq$

$$
\begin{array}{lll}
\Rrightarrow \Phi\ s_t\ s_s & & \textit{base step} \\
\quad \vee\ ((s_t = \mathsf{vis\ call}(f_t, args_t, attr_t)\ k_t) * (s_s = \mathsf{vis\ call}(f_s, args_s, attr_s)\ k_s) * & & \textit{call spec step} \\
\quad\ (\exists C.\ \mathcal{S}(\sigma_t, \sigma_s) * C^C_{\mathsf{ev}}(f_t, args_t, attr_t, f_s, args_s, attr_s) * & & \\
\quad\quad (\forall v_t.v_s, \mathcal{S}(\pi_1(v_t), \pi_1(v_s)) * C^C_{\mathsf{ans}}(args_t, attr_t, args_s, attr_s, \pi_2(v_t), \pi_2(v_t)) & & \\
\quad\quad\quad -\!\!* \Rrightarrow \mathcal{G}(k_t\ v_t, k_s\ v_s))) \vee & & \\
\quad\ (\exists C.\ \mathcal{S}(\sigma_t, \sigma_s) * \overrightarrow{\mathcal{V}}^C(args_t, args_s) * & & \textit{call lock step} \\
\quad\quad (\forall v_t.v_s, \mathcal{S}(\pi_1(v_t), \pi_1(v_s)) * \overrightarrow{\mathcal{V}}^C(\pi_2(v_t), \pi_2(v_t)) & & \\
\quad\quad\quad -\!\!* \Rrightarrow \mathcal{G}(k_t\ v_t, k_s\ v_s)))) & & \\
\quad \vee\ ((s_t = \mathsf{vis}\ ev_t\ k_t) * (s_s = \mathsf{vis}\ ev_s\ k_s) * & & \textit{visible step} \\
\quad\ e_t = e_s * (\forall v_t, v_s.v_t = v_s -\!\!* \Rrightarrow \mathcal{G}(k_t\ v_t, k_s\ v_s)) & & \\
\quad \vee\ ((s_t = \mathsf{tau}\ t_t)\ * \mathcal{L}\ \Phi\ t_t\ s_s & & \textit{stutter step (target)} \\
\quad \vee\ ((s_s = \mathsf{tau}\ t_s)\ * \mathcal{L}\ \Phi\ t_t\ s_s & & \textit{stutter step (source)} \\
\quad \vee\ ((s_t = \mathsf{tau}\ t_t) * (s_s = \mathsf{tau}\ t_s)\ * \mathcal{G}\ \Phi\ t_t\ s_s & & \textit{locking silent step} \\
\quad \vee\ (s_s = \mathsf{Exc})\ \vee\ (s_s = \mathsf{UB}) & & \textit{source error/UB}
\end{array}
$$

Figure 11.1: isimF definition

configurations satisfying the appropriate invariants for events (C is the external call invariant, while $\mathcal{E}$ is the non-state-affecting events invariant). The locking silent step case is where there is a silent (`Tau`) step on both computations, and where we peel off the silent steps in lock-step and call the greatest fixpoint. The stutter case is where a silent step can be ignored for either the source or target computation. The source UB case and error cases are when there is an undefined behavior (UB) or error in the source program, so the simulation holds trivially. For all other cases, the source and target programs are out of sync (mismatched), and thus not in simulation.

isim is defined by forming a mixed greatest-least fixpoint using the Knaster-Tarski construction we presented in Section 11.1.1.2.

$$\mathsf{isim}\ \Phi\ e_t\ e_s := \nu_{\mathsf{bi}}(\mu_{\mathsf{bi}}(\mathsf{isimF}))$$

11.2.3 Memory attribute logical interpretation

In this section, we discuss the logical interpretation of memory attribute, as used in the call cases described in Section 11.2.2. The *event predicate* C_{ev} and *answer predicate* C_{ans} are defined as follows.

Definition 11. The *event predicate* $C^C_{\mathsf{ev}}(f_t, args_t, attr_t, f_s, args_s, attr_s)$ is defined as $\overrightarrow{\mathcal{V}}(args_t, args_s) * \mathcal{V}(f_t, f_s) * attr_t = attr_s * \mathsf{checkout}(C) * \mathsf{attrib\_interp}(args_t, args_s, attr_t, C)$.

The *answer predicate* $C^C_{\mathsf{ans}}(args_t, attr_t, args_s, attr_s, v_t, v_s)$ is defined as $\mathcal{V}(v_t, v_s) * attr_t = attr_s * \mathsf{checkout}(C) * \mathsf{attrib\_interp}(args_t, args_s, attr_t, C)$.

With these definitions in place, we provide the logical interpretation for memory attributes below.

Attribute	Logical interpretation for arguments (a_t, a_s) and checkout set C
Nothing	$C = \emptyset$
`readonly`	$\forall(\ell_t,\ \ell_s) \in C, C(\ell_t,\ \ell_s) = q \land q < 1$
`argmemonly`	$\forall(\ell_t,\ \ell_s) \in \mathsf{zip}\ a_t\ a_s,\ (\ell_t,\ \ell_s) \notin C$
`argmemonly, readonly`	$\forall(\ell_t,\ \ell_s) \in \mathsf{zip}\ a_t\ a_s, \exists q, C(\ell_t,\ \ell_s) = q \land q < 1$

The logical specifications are externally equivalent to the attribute specifications in LLVM IR. From the perspective of the caller, the *readonly* function reads resources that are visible only from the caller. Because of this, if the callee had read any resources that is not visible from the caller, the reasoning principles should not change from the caller's perspective. From a program transformation perspective, the *readonly* specification that it does not read from any region of memory is unnecessarily strong, because local reads that is not externally observable should not affect the external environment. The nop function (i.e. function with no effect) vacuously satisfies all logical specifications. Figure 11.2 show minimal examples of functions which satisfy their attributes. We have proved that the examples in figure 11.2 satisfy our logical specification, which demonstrates a proof of existence for functions which satisfy these specifications.

11.2.4 sim rules

The main proof rules of the simulation $e_t \preceq e_s\ \{\Phi\}$ are described in Figure 11.3. Most of these rules are reminiscent of standard relational hoare logics for monadic programs. The simulation is monotonic (SimMono), and also monotonic under the fancy update operator (SimBupdMono). The cut principle

```
define void @ex1() readonly {
  %1 = alloca i32
  %1 = load i32, i32* %0
}

define void @ex2(i32 %1) argmemonly readonly {
  %1 = load i32, i32* %1
}

define void @ex2(i32 %1) argmemonly {
  store i32 42, i32* %1
}
```

Figure 11.2: Example functions that satisfy their logical attributes

holds for the bind operator (SimBind), and the inversion rules for fmap and ret. These rules as proven as lemmas and are properties that hold for the simulation; for the complete set of rules and proofs, refer to the mechanization.

Coinductive reasoning. While the triple $e_t \preceq e_s \{\Phi\}$ is useful for stating postconditions about returned values, we have a variant $e_t \preceq e_s [\ \Phi\ ]$ which states a postcondition about whole computations. It is defined as $e_t \preceq e_s [\ \Phi\ ] := \Box(P \mathrel{-\!\!*} \mathsf{sim}\ e_t\ e_s\ Q)$. This triple does not have the lifting ($\uparrow$) over the postcondition, which means that the postcondition is over programs, not values. This is especially useful for accumulating information in coinductive proofs.

Figure 11.4 contains the proof rules pertaining to coinduction. Most notably, the SimIter rule allows reasoning about the iteration combinator (iter). The user of this rule can simply prove that a relational loop invariant F will be preserved by the loop body without explicit coinduction.

Language-specific mutual recursion principle. The mutual recursion principle law is shown in Figure 11.5, where it states that a mutually recursive function over the body f and g with initial values x and y can be shown equivalent if an invariant F is preserved over the function bodies. The proof of the mutual recursion law involves coinductive proof of language-specific inversion laws over interpreted results of a given expression.

$$\frac{\Phi(\texttt{ret}\ v_t,\ \texttt{ret}\ v_s)}{v_t \preceq v_s\ \{\Phi\}}\ \textsc{SimBase} \qquad \frac{\forall\ e_t, e_s.\Phi(e_t, e_s) \mathrel{-\!\!*} \Phi'(e_t, e_s) \qquad e_t \preceq e_s\ \{\Phi\}}{e_t \preceq e_s\ \{\Phi'\}}\ \textsc{SimMono}$$

$$\frac{e_t \preceq e_s\ \{\Psi\} \qquad \forall v_t\ v_s.\ (k_t\ x) \preceq (k_s\ x)\ \{\Phi\}}{(x \leftarrow e_t\,;;\ k_t\ x) \preceq (x \leftarrow e_s\,;;\ k_s\ x)\ \{\Phi\}}\ \textsc{SimBind}$$

$$\frac{\forall\ e_t, e_s.\Phi(e_t, e_s) \mathrel{-\!\!*} \Rrightarrow \Phi'(e_t, e_s) \qquad e_t \preceq e_s\ \{\Phi\}}{\Rrightarrow e_t \preceq e_s\ \{\Phi'\}}\ \textsc{SimBupdMono}$$

$$\frac{e_t \preceq e_s\ \{\Phi\}}{(\mathsf{tau}\ e_t) \preceq (\mathsf{tau}\ e_s)\ \{\Phi'\}}\ \textsc{SimTau} \qquad \frac{e_t \preceq e_s\ \{\Phi\}}{(\mathsf{tau}\ e_t) \preceq e_s\ \{\Phi'\}}\ \textsc{SimTauL} \qquad \frac{e_t \preceq e_s\ \{\Phi\}}{e_t \preceq (\mathsf{tau}\ e_s)\ \{\Phi'\}}\ \textsc{SimTauR}$$

$$\frac{\Phi(\mathsf{vis}\ ev_t\ k_t, \mathsf{vis}\ ev_s\ k_s) \vee (\mathsf{handle\_event}\ (\mathsf{sim}\ \Phi) k_t\ k_s\ ev_t\ ev_s)}{(\mathsf{vis}\ ev_t\ k_t) \preceq (\mathsf{vis}\ ev_s\ k_s)\ \{\Phi\}}\ \textsc{SimVis} \qquad \frac{}{e_t \preceq \mathsf{Exc}\ \{\Phi'\}}\ \textsc{SimExc}$$

$$\frac{(\mathsf{fmap}\ f\ e_t) \preceq (\mathsf{fmap}\ g\ e_s)\ \{\Phi\}}{e_t \preceq e_s\ \{(\mathsf{bimap}\ f\ g\ \Phi')\}}\ \textsc{SimFmapInv} \qquad \frac{(\mathsf{ret}\ v_t) \preceq (\mathsf{ret}\ v_s)\ \{\Phi\}}{\mathcal{S}(\pi_1\ v_t, \pi_1\ v_s) * \Phi(\pi_2\ v_t, \pi_2\ v_s)}\ \textsc{SimRetInv}$$

Figure 11.3: Basic proof rules for simulation

11.3 Adequacy

With the model in place, we can state our main results on adequacy. We define a whole-program relation $p_t \preceq p_s$ in the Iris logic implies whole-program refinement in the Coq logic. The whole-program relation $p_t \preceq p_s := p_t \preceq p_s\ \{\mathcal{V}\}$ is a relation over two programs with the value relation $\mathcal{V}$ as the postcondition.

Theorem 2 (Adequacy)**.** Given programs p_t (target) and p_s (source) that are closed, if the source program program is well-behaved (i.e. does not throw an exception), then if $\mathcal{S}(\sigma_t, \sigma_s) * p_t \preceq p_s$, then $[\![p_t]\!]\sigma_t \approx_{\mathcal{V}} [\![p_s]\!]\sigma_s$.

The adequacy theorem states that for well-behaved source programs, and the whole-program relation holds for sim in the Iris logic along with a state interpretation holding for initial states σ_t and σ_s, the relation $[\![p_t]\!]\sigma_t \approx_{\mathcal{V}} [\![p_s]\!]\sigma_s$ holds in the meta logic.

$$\frac{e_t \preceq e_s\ \{\Phi\}}{e_t \preceq e_s\ [\ \Phi\ ]}\ \textsc{SimLift} \qquad \frac{e_t \preceq e_s\ [\ \Phi^{\uparrow}\ ]}{e_t \preceq e_s\ \{\Phi\}}\ \textsc{SimRet}$$

$$\frac{\begin{array}{l}\square(\forall \Phi, e_t, e_s. F\ \Phi\ e_t\ e_s \mathrel{-\!\!*} (e_t = \mathsf{tau}\ e_t' * e_s = \mathsf{tau}\ e_s * e_t' \preceq e_s'\ \{((F\ \Phi) \vee_2 \Phi))\} \vee \\ \qquad (e_t = \mathsf{vis}\ ev_t\ k_t * e_s = \mathsf{vis}\ ev_s\ k_s * \mathsf{handleEvent}\ (\mathsf{isim}\ ((F\ \Phi) \vee_2 \Phi))\ k_t\ k_s\ ev_t\ ev_s))\end{array}}{F\ \Phi\ e_t\ e_s \mathrel{-\!\!*} e_t \preceq e_s\ [\ \Phi\ ]}\ \textsc{SimPaco}$$

$$\frac{\begin{array}{l}F\ i_t\ i_s \\ \square(\forall i_t, i_s.\ F\ i_t\ i_s \mathrel{-\!\!*} \\ \qquad f_t\ i_t \preceq f_s\ i_s\ [\ \lambda e_t, e_s. \\ \qquad\qquad (\exists r_t, r_s. e_t = \mathsf{ret}\ (\mathtt{inl}\ r_t) * e_s = \mathsf{ret}\ (\mathtt{inl}\ r_s) * F\ i_t\ i_s) \\ \qquad\qquad \vee (\exists r_t, r_s. e_t = \mathsf{ret}\ (\mathtt{inr}\ r_t) * e_s = \mathsf{ret}\ (\mathtt{inr}\ r_s) * \Phi\ e_t\ e_s)\ ])\end{array}}{\mathsf{iter}(f_t, i_t) \preceq \mathsf{iter}(f_s, i_s)\ [\ \Phi\ ]}\ \textsc{SimIter}$$

Figure 11.4: Parameterized coinduction with isim

$$\frac{\begin{array}{c}F\ i_t\ i_s \qquad \square(\forall i_t, i_s.\ F\ i_t\ i_s \mathrel{-\!\!*} x \preceq y\ [\ \lambda e_t, e_s.\ \Phi\ x\ y * F\ i_t\ i_s\ ]) \\ \square(\forall i_t, i_s. \overrightarrow{\mathcal{V}}(args_t, args_s) * \mathcal{V}(f_t, f_s) \mathrel{-\!\!*} f\ (\mathtt{Call}^{\mathcal{V}_u}(f_t, args_t)) \preceq g\ (\mathtt{Call}^{\mathcal{V}_u}(f_s, args_s))\ [\ \mathcal{V}\ ])\end{array}}{\mathsf{interp\_mrec}\ f\ x \preceq \mathsf{interp\_mrec}\ g\ y\ [\ \Phi\ ]}$$

Figure 11.5: Mutual recursion law

11.4 Contextual refinement

Contextual refinement ensures a compiler can replace the expression e_s—which may be open—with the specified expression e_t. In this section, we define the contextual refinement for Velliris, and demonstrate how it can be proved using the simulation relation. We first introduce the notion of contextual refinement in Section 11.4.1, introduce the coinductive reasoning principles that are necessary to prove the contextual refinement result 9.4.

11.4.1 Contextual Refinement

The program context in VIR is a mutually recursive control flow graph with a hole at an arbitrary place. The context $C[-]$ consists of the (possibly open) mutually recursive control flow graph C, with a hole that takes in a (possibly open) function definition, which may be referred to by C and also can call functions in C.

Definition 12 (Contextual refinement)**.** $e_t \sqsubseteq_{\mathsf{ctx}} e_s := \forall C, \mathsf{wf}\ C \wedge \mathsf{wf}_{prog}(\llbracket C[e_t] \rrbracket \sigma_t)(\llbracket C[e_s] \rrbracket \sigma_s) \Rightarrow$

$(\llbracket C[e_t] \rrbracket \sigma_t) \approx_{\mathcal{V}^\downarrow} (\llbracket C[e_s] \rrbracket \sigma_s)$.

The well-formedness condition over programs (wf) states that the source program does not go wrong, and that the program composed with the context $(\llbracket C[e] \rrbracket \sigma)$ is closed. Our main result for contextual refinement, *contextual adequacy*, is as follows.

Theorem 3 (Contextual adequacy)**.** Given $e_t \preceq_{\log}^{fun} e_s$, then $e_t \sqsubseteq_{\text{ctx}} e_s$.

The result of contextual adequacy allows us to lift results about open function calls in the Velliris logic to a closing context in the Coq logic. In order to prove the contextual adequacy, we need two more components: a *logical relation*, and *coinductive reasoning* principles to prove theorems about whole programs and (mutually-recursive) control flow graphs.

11.4.2 Logical Relation

We use the standard technique for proving contextual refinements by using *logical relations*. Because the syntax of VIR involves various constructs, we must define a logical relation for each construct. We prove a fundamental theorem result, where we define a logical relation for each syntactic VIR construct and show that each relation so defined is reflexive.

The invariant for the logical relation is defined in Figure 11.6, which states the invariant for ghost resources across a control flow graph. It states that (1) there is some local domain in both the source and target program so that they are pointwise related, (2) the frame index is maintained, and (3) the locally allocated set is not leaked as a public location (i.e. is not included in the checked-out set).

The definitions of the logical relations are given in Figure 11.7 and 11.8. Figure 11.7 show the logical relations for expressions, terminators, instructions, phi nodes, code, block, and (open) control flow graphs. Expressions, terminators, and phi nodes should not affect the domain of stack-allocated resources, whereas code, block, and (open) control flow graphs increase the allocated set of locations monotonically.

$$\mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)} := \exists args_t, args_s.\ \mathsf{LocalDomain}^{i_t}_{\mathsf{tgt}}\ (\overrightarrow{\pi_1}(args_t)) * \mathsf{LocalDomain}^{i_s}_{\mathsf{src}}\ (\overrightarrow{\pi_1}(args_s))$$

$$\mathsf{Frame}_{\mathsf{tgt}}\ i_t * \mathsf{Frame}_{\mathsf{src}}\ i_s * \mathsf{Alloca}^{i_t}_{\mathsf{tgt}}\ A_t * \mathsf{Alloca}^{i_s}_{\mathsf{src}}\ A_s * \mathsf{checkout}(C) *$$

$$(\mathop{\Large *}_{(l_t,v_t);(l_s,v_s)\in args_t;args_s} [\ l_t := v_t\ ]i_t * [\ l_s := v_s\ ]i_s * l_t = l_s * \mathcal{V}_{\mathsf{U}}(v_t, v_s)) *$$

$$(\mathop{\Large *}_{v_t;v_s\in A_t,A_s} \mathcal{V}_{\mathsf{U}}(v_t, v_s) * (v_t, v_s) \notin C)$$

Figure 11.6: Invariant for Velliris logical relation

$$e_t \preceq^{exp}_{\mathsf{log}\ (A_t,A_s,C)} e_s := \forall i_t, i_s.\ \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)} -\!\!* [\![e_t]\!]^{\uparrow_{\mathsf{exp}}}_{\mathsf{expr}} \preceq [\![e_s]\!]^{\uparrow_{\mathsf{exp}}}_{\mathsf{expr}} \{\lambda v_t, v_s. \mathcal{V}_{\mathsf{U}}(v_t, v_s) * \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)}\}$$

$$t_t \preceq^{term}_{\mathsf{log}\ (A_t,A_s,C)} t_s := \forall i_t, i_s.\ \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)} -\!\!* [\![t_t]\!]^{\uparrow_{\mathsf{exp}}}_{\mathsf{term}} \preceq [\![t_s]\!]^{\uparrow_{\mathsf{exp}}}_{\mathsf{term}}$$
$$\{\lambda v_t, v_s.\ ((\exists b_t, b_s. v_t = \mathtt{inl}\ b_t * v_s = \mathtt{inl}\ b_s * b_t = b_s) \vee$$
$$(\exists b_t, b_s. v_t = \mathtt{inr}\ b_t * v_s = \mathtt{inr}\ b_s * \mathcal{V}_{\mathsf{U}}(b_t, b_s)) * \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)}\}$$

$$\iota_t \preceq^{instr}_{\mathsf{log}\ (A_t,A_s,C)} \iota_s := \forall i_t, i_s.\ \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)} -\!\!* [\![\iota_t]\!]^{\uparrow_{\mathsf{instr}}}_{\mathsf{instr}} \preceq [\![\iota_s]\!]^{\uparrow_{\mathsf{instr}}}_{\mathsf{instr}} \{\exists A'_t, A'_s. \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t@A'_t,A_s@A'_s)}\}$$

$$\phi_t \preceq^{phi}_{\mathsf{log}\ (A_t,A_s,C)} \phi_s := \forall i_t, i_s.\ \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)} -\!\!* [\![\phi_t]\!]^{\uparrow_{\mathsf{instr}}}_{\phi} \preceq [\![\phi_s]\!]^{\uparrow_{\mathsf{instr}}}_{\phi}$$
$$\{\lambda' (l_t, v_t),' (l_s, v_s).\ l_t = l_s * \mathcal{V}_{\mathsf{U}}(v_t, v_s) * \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)}\}$$

$$\Phi_t \preceq^{phis}_{\mathsf{log}\ (A_t,A_s,C)} \Phi_s := \forall i_t, i_s.\ \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)} -\!\!* [\![\Phi_t]\!]^{\uparrow_{\mathsf{instr}}}_{\Phi} \preceq [\![\Phi_s]\!]^{\uparrow_{\mathsf{instr}}}_{\Phi}$$
$$\{\lambda' (l_t, v_t),' (l_s, v_s). l_t = l_s * \mathcal{V}_{\mathsf{U}}(v_t, v_s) * \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)}\}$$

$$c_t \preceq^{code}_{\mathsf{log}\ (A_t,A_s,C)} c_s := \forall i_t, i_s.\ \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)} -\!\!* [\![c_t]\!]^{\uparrow_{\mathsf{instr}}}_{code} \preceq [\![c_s]\!]^{\uparrow_{\mathsf{instr}}}_{code} \{\exists A'_t, A'_s. \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t@A'_t,A_s@A'_s)}\}$$

$$b_t \preceq^{blk}_{\mathsf{log}\ (A_t,A_s,C)} b_s := \forall i_t, i_s.\ \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)} -\!\!* [\![b_t]\!]^{\uparrow_{\mathsf{instr}}}_{blk} \preceq [\![b_s]\!]^{\uparrow_{\mathsf{instr}}}_{blk}$$
$$\{\lambda v_t, v_s.\ ((\exists b_t, b_s. v_t = \mathtt{inl}\ b_t * v_s = \mathtt{inl}\ b_s * b_t = b_s) \vee$$
$$(\exists b_t, b_s. v_t = \mathtt{inr}\ b_t * v_s = \mathtt{inr}\ b_s * \mathcal{V}_{\mathsf{U}}(b_t, b_s)) *$$
$$\exists A'_t, A'_s. \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t@A'_t,A_s@A'_s)}\}$$

$$o_t \preceq^{ocfg}_{\mathsf{log}\ (A_t,A_s,C)} o_s := \forall i_t, i_s.\ \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)} -\!\!* [\![o_t]\!]^{\uparrow_{\mathsf{instr}}}_{ocfg} \preceq [\![o_s]\!]^{\uparrow_{\mathsf{instr}}}_{ocfg}$$
$$\{\lambda v_t, v_s.\ ((\exists id_t, id_s. v_t = \mathtt{inl}\ id_t * v_s = \mathtt{inl}\ id_s * id_t = id_s) \vee$$
$$(\exists b_t, b_s. v_t = \mathtt{inr}\ b_t * v_s = \mathtt{inr}\ b_s * \mathcal{V}_{\mathsf{U}}(b_t, b_s)) *$$
$$\exists A'_t, A'_s. \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t@A'_t,A_s@A'_s)}\}$$

Figure 11.7: Logical relations for Velliris

$$c_t \ \preceq^{cfg}_{\log\ (A_t,A_s,C)}\ c_s := \forall i_t, i_s.\ \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t,A_s)} \mathrel{-\!\!*} [\![c_t]\!]^{\uparrow_{\mathsf{instr}}}_{cfg} \preceq [\![c_s]\!]^{\uparrow_{\mathsf{instr}}}_{cfg}$$
$$\{\lambda v_t, v_s.\ \mathcal{V}_{\mathsf{U}}(v_t, v_s) * \exists A'_t, A'_s. \mathsf{Inv}^{(i_t,i_s)}_{(C,A_t@A'_t,A_s@A'_s)}\}$$

$$f_t \ \preceq^{fun}_{\log\ (A_t,A_s,C)}\ f_s := \forall i_t, i_s.\ (|i_s| > 0 \to |i_t| > 0) *$$

$$\mathsf{Frame}_{\mathsf{tgt}}\ i_t * \mathsf{Frame}_{\mathsf{src}}\ i_s * \mathsf{checkout}(C) * (\mathop{\Large *}_{v_t,v_s \in args_t,args_s} \mathcal{V}_{\mathsf{U}}(v_t, v_s)) \mathrel{-\!\!*}$$

$$[\![f_t]\!]^{\uparrow_{\mathsf{instr}}}_{fun} \preceq [\![f_s]\!]^{\uparrow_{\mathsf{instr}}}_{fun}$$

$$\{\lambda v_t, v_s.\ \mathcal{V}_{\mathsf{U}}(v_t, v_s) * \mathsf{Frame}_{\mathsf{tgt}}\ i_t * \mathsf{Frame}_{\mathsf{src}}\ i_s * \mathsf{checkout}(C)\}$$

$$F_t \ \preceq^{funs}_{\log\ (A_t,A_s,C)}\ F_s := \forall i, i_t, i_s, a_t, f_t, a_s, f_s.(|i_s| > 0 \to |i_t| > 0) *$$

$$F_t[i] = (a_t, f_t) * F_s[i] = (a_s, f_s) * \mathcal{V}_{\mathsf{Dyn}}(a_t, a_s) * \mathsf{Frame}_{\mathsf{tgt}}\ i_t * \mathsf{Frame}_{\mathsf{src}}\ i_s *$$

$$\mathsf{checkout}(C) * (\mathop{\Large *}_{v_t,v_s \in args_t,args_s} \mathcal{V}_{\mathsf{U}}(v_t, v_s)) \mathrel{-\!\!*}$$

$$[\![f_t]\!]^{\uparrow_{\mathsf{instr}}}_{fun} \preceq [\![f_s]\!]^{\uparrow_{\mathsf{instr}}}_{fun}$$
$$\{\lambda v_t, v_s.\ \mathcal{V}_{\mathsf{U}}(v_t, v_s) * \mathsf{Frame}_{\mathsf{tgt}}\ i_t * \mathsf{Frame}_{\mathsf{src}}\ i_s * \mathsf{checkout}(C)\}$$

Figure 11.8: Logical relations for Velliris, continued.

Figure 11.8 show the logical relations for functions and list of function definitions. Any stack-allocated and local resource is available only at the scope of the control flow graph, and thus the logical relation invariant is not relevant for functions and function definitions. Refer to Section 3.3 for the denotation of each syntactic construct in VIR.

Theorem 4 (Fundamental Theorem)**.** For any well-formed function definitions F, $F \ \preceq^{funs}_{\log\ (A_t,A_s,\emptyset)}\ F$.

Proof. Each of the logical relations for syntactic constructs must be proved. 1. **Expression**: follows by case analysis; the cases for Struct and Array follow by induction. 2. **Terminator**: follows by case analysis. 3. **Instruction**: follows by reflexivity of *concretized expressions* in a deterministic setting, along with the invariants being preserved by local writes and reads. For the call case, we prove the reflexivity by a strong

lock step on calls. 4. **Code**: follows by straightforward induction on the result of instruction reflexivity. The **Block** case follows immediately. 5. **Phi**: follows by case analysis; the **Phis** case is a induction over the results of the reflexivity of **Phi**. 6. **Ocfg**: follows by a parameterized coinduction using the SimIter rule. 7. **Cfg**: follows immediately from the **Ocfg** case. 8. **Function**: follows from well-bracketedness of calls, using the derived laws on frames pop and push. 9. **Function definitions**: follows from an induction on function definitions and the well-formedness property over definitions that assumes uniqueness of function addresses.

□

The proof of the fundamental theorem involves the use of SimIter and SimMrec in order to reason about control-flow graphs and mutually-recursive control flow graphs. The case for *code* (list of instructions), *phi* nodes, and aggregate data types follow by induction. The invariant Inv is crucial for managing ownership over stack-allocated memory—given a code block, the allocated set of locations increase monotonically and its ownership is kept track of by the public bijection, such that at function return the stack-allocated resources can be released in-sync.

We use the fundamental theorem for proving a contextual closure result, as described below.

Theorem 5 (Contextual closure)**.** If the context C is well-formed and $e_t \preceq_{\text{log}}^{fun} e_s$, then $C[e_t] \preceq_{\text{log}}^{funs} C[e_s]$.

The well-formedness predicate states that addresses for functions are unique, the programs are well-typed, and that operations do not values of size 0 to memory. With the proof of contextual closure and the mutual recursion law, we prove the contextual adequacy result, as stated in Section 11.4.1.

In order to prove contextual refinement, we use the language-specific mutual recursion principle defined in Figure 11.5. Given the proof of contextual closure using this mutual recursion law, we can now state the contextual refinement result.

Definition 13 (Contextual refinement)**.** $e_t \sqsubseteq_{\text{ctx}} e_s := \forall C, \mathsf{wf}\ C \land \mathsf{wf}_{prog}(\llbracket C[e_t] \rrbracket \sigma_t)(\llbracket C[e_s] \rrbracket \sigma_s) \Rightarrow (\llbracket C[e_t] \rrbracket \sigma_t) \approx_{\mathcal{V}\downarrow} (\llbracket C[e_s] \rrbracket \sigma_s)$.

The well-formedness condition over programs (wf) states that the source program does not go wrong, and that the program closed over with the context ($\llbracket C[e] \rrbracket \sigma$) is closed. The adequacy for contextual refinement is as follows.

Theorem 6 (Contextual adequacy)**.** Given $e_t \preceq^{fun}_{\log} e_s$, then $e_t \sqsubseteq_{\text{ctx}} e_s$.

The proof of contextual adequacy follows from the fundamental theorem, adequacy, and the language-specific mutual recursion principle from Figure 11.5. This result of contextual adequacy allows us to lift results about open function calls in the Velliris logic to a closing context in the Coq logic. For instance, this is useful for lifting the result of verified example optimizations between code blocks shown in previous sections (see Section 9.2.4, Section 9.3.4, and Section 9.4.2) to whole programs.

11.5 Related Work

LLVM IR Semantics. There is ample existing work that aims to build formal semantics for (oftentimes just parts of) the LLVM IR. Notable examples include the prior versions of Vellvm [ZNMZ12, ZNMZ13], Alive [LMNR15, MN17], Crellvm [KKS+18], and K-LLVM [LG20a] projects. There are also attempts to characterize LLVM's undefined behaviors [LKS+17], its concurrency semantics [CV17], and memory models [KHM+15a, LHJ+18b]. Of these, K-LLVM stands out as the most complete formal model of the LLVM IR to date. Constructed using the K Framework [RS14, Ros17], K-LLVM is an executable reference specification of nearly all of the LLVM IR constructs, including rich support of *intrinsics*. K-LLVM is high-fidelity enough to pass a suite of standard LLVM IR unit tests. Li's dissertation [Li20] introduces a notion of "per-location simulation" with respect to the K-LLVM memory model semantics and uses it to establish the correctness of a variety of program transformations that are expressible by Mansky, et al.'s Morpheus (a DSL for specifying compiler optimizations) [MGGA16]. By comparison, the VIR semantics we present here is less complete than K-LLVM: VIR has no support for concurrency, and supports only a few of the LLVM IR intrinsics, for instance. However, Velliris, as a program logic embedded in Iris/Coq, is potentially more flexible than the K-LLVM (one can draw on the full power of interactive theorem proving if necessary), and, to our knowledge, the contextual adequacy result of this thesis has no analog in the K-LLVM setting.

Relational Separation Logics. Benton's Relational Hoare Logic [Ben04] and Nanevski, *et al.*'s Relational Hoare Type Theory [NBG13] have been shown to be useful for reasoning about program transformations and properties such as information flow. *Relational separation logic*, introduced by Yang [Yan07], was developed to specify and verify how two pointer programs are related. Since then, there have been variants of relational

separation logics [EDM23, Gre23] which prove properties about contextual refinement, simulation, and security. Most notably, both Trillium [TGS+21] and Simuliris [GSS+22] are relational separation logics in the Iris framework, both focused on a concurrent operational semantics. On the other hand, Velliris works with a sequential, monadic semantics based on the Interaction Trees framework, scaled up to the VIR semantics. Other logics for simulations have mostly been developed in the context of using refinement for *program verification* [Dan18, TJH17, SGG+21, LFS14, LF16, LF18]: the goal is to show that an *implementation* of an abstract data type implements a *specification*, which is distinct from our goal of verifying *program optimizations*.

Chapter 12

Conclusion

This dissertation builds a modular foundation for reasoning about program transformations in LLVM. In Part I, we introduced VIR, a modular and executable semantics for a significant sequential subset of LLVM IR. This approach is grounded in an Interaction Tree semantics, a structure that provides a more compositional approach to defining language semantics while retaining the ability to extract an executable interpreter. In Part II, we established a formal metatheory for the analysis of layered interpreters, offering an extensible framework for lifting interpreters and structural rules. This framework characterizes interpretable monads and introduces a relational reasoning system for assessing equivalences across interpretations. Lastly, in Part III, we constructed a relational separation logic framework aimed at verifying program transformations on VIR, offering a novel perspective on verifying transformations involving external calls.

12.1 Future work

12.1.1 Part I. VIR Semantics

Connecting the semantics to a compilation pass. One of the benefits of LLVM IR comes from its use by various front-ends and back-ends: many modern languages such as C, Rust, and Haskell emit LLVM IR, and LLVM IR can generate common instruction architectures sets such as PowerPC, Intel X86, MIPS, and RISC-V. A line of future direction is to connect VIR to a front-end and/or a backend, fully connecting it to a compiler pipeline and verifying a compilation pass, or a full stack of compilation. There is on-going work on verifying a front-end pass through the HELIX project [Zal21], where a front-end for high-performance and high-assurance numerical computing is compiled to the VIR framework.

Support for concurrency. Another direction for future work is adding concurrency to the VIR semantics. This direction requires two significant components: (1) defining a concurrent semantics and reasoning techniques for an Interaction Tree based semantics that can scale, and (2) developing a suitable weak memory

model for LLVM IR. As for the semantics, there has been developments of Interaction Tree-like semantics for representing nondeterministic programs [CHH+23], and it is future work to determine whether this new semantics is fit for representing concurrent LLVM IR programs.

12.1.2 Part II. A Layered Equational Framework.

Nondeterminism and probability Combining nondeterminism with other effects in a generic way is a long standing problem in denotational semantics. The standard forms of distributive law is known to not hold when combining nondeterminism and probability. Kozen and Silva [KS23] has recently shown a lightweight construction of distributive laws when combining these two effects together using a multiset representation. A direction for future work in eqmR is to explore the multiset representation for nondeterministic effects using Kozen and Silva's result.

Using different notions of equality. The `interp` function that is seen for intepretable monads can be seen as a structure-preserving function, preserving certain structural laws across interpretation. The eqmR approach can be seen as a setoid approach where a monad is paired with a certain notion of equivalence. However, it may be more natural to work with a type theory with a more expressive notion of equivalnce that can manipulate structure-preserving functions, such as in cubical type theory [CCHM16] and directed type theory [WL20].

12.1.3 Part III. A Separation Logic Framework.

Verification of realistic optimization algorithm. A direction of future work for the separation logic framework in Part III is the verification of a realistic optimization algorithm. An extensive case study through a verification of a realistic optimization algorithm can showcase the effectiveness of the framework and can help explore any limitations in the framework through verifying a realistic optimization pass.

Connecting the framework to analysis passes. An important part of LLVM IR optimizations is its various analysis passes; most optimizations involve several analysis passes through the program. An interesting avenue of future work is to develop a general verified framework for LLVM IR optimization which connects to a software component that performs analysis passes.

BIBLIOGRAPHY

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, page 147–160, New York, NY, USA, 1999. Association for Computing Machinery.

[Ahm04] Amal Jamil Ahmed. *Semantics of Types for Mutable State.* PhD thesis, USA, 2004. AAI3136691.

[AHM+17] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, 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 '17, page 515–529, New York, NY, USA, 2017. Association for Computing Machinery.

[AM01] Andrew W. Appel and David McAllester. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, Sep 2001.

[Apf10] Heinrich Apfelmus. The Operational Monad Tutorial. *The Monad.Reader*, Issue 15, 2010.

[APH+08] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

[App11] Andrew W. Appel. Verified Software Toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BBG+20] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL):7:1–7:30, 2020.

[BCF+14] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 87–100, New York, NY, USA, 2014. Association for Computing Machinery.

[BÉ93] Stephen L. Bloom and Zoltán Ésik. *Iteration Theories - The Equational Logic of Iterative Processes.* EATCS Monographs on Theoretical Computer Science. Springer, 1993.

[Ben04] Nick Benton. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. *SIGPLAN Not.*, 39(1):14–25, Jan 2004.

[BKBH09] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. Relational Semantics for Effect-Based Program Transformations: Higher-Order Store. In *Proceedings of*

the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP '09, page 301–312, New York, NY, USA, 2009. Association for Computing Machinery.

[BP14] Andrej Bauer and Matija Pretnar. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science*, Volume 10, Issue 4, Dec 2014.

[BP15] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015. Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.

[BPYA15] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified Correctness and Security of OpenSSL HMAC. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 207–221, USA, 2015. USENIX Association.

[Cap05] Venanzio Capretta. General Recursion via Coinductive Types. *Logical Methods in Computer Science*, Volume 1, Issue 2, Jul 2005.

[CBG+18] 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.

[CCHM16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct 1991.

[Cha13] Arthur Charguéraud. Pretty-Big-Step Semantics. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, page 41–60, Berlin, Heidelberg, 2013. Springer-Verlag.

[CHH+23] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. Choice trees: Representing nondeterministic, recursive, and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 7(POPL), 2023.

[Chl10] Adam Chlipala. A Verified Compiler for an Impure Functional Language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 93–106, New York, NY, USA, 2010. ACM.

[CJS+23] Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design*

and Implementation (OSDI 23), pages 871–886, Boston, MA, Jul 2023. USENIX Association.

[CV17] Soham Chakraborty and Viktor Vafeiadis. Formalizing the Concurrency Semantics of an LLVM Fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 100–110. IEEE Press, 2017.

[DAB09] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical Step-Indexed Logical Relations. In *Proceedings of the 2009 24th Annual IEEE Symposium on Logic In Computer Science*, LICS '09, page 71–80, USA, 2009. IEEE Computer Society.

[Dan18] Dan Frumin and Robbert Krebbers and Lars Birkedal. ReLoC: A mechanised relational logic for fine-grained concurrency. In *LICS*, pages 442–451, 2018.

[DSG+22] Derek Dreyer, Simon Spies, Lennard Gäher, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, David Swasey, and Jan Menz. Semantics of Type Systems Lecture Notes, July 2022.

[EDM23] Marco Eilers, Thibault Dardinier, and Peter Müller. CommCSL: Proving Information Flow Security for Concurrent Programs Using Abstract Commutativity. *Proc. ACM Program. Lang.*, 7(PLDI), Jun 2023.

[Ell12] Charles Ellison. *A formal semantics of C with applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2012.

[ELN+13] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A Fully Verified Executable LTL Model Checker. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 463–478, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[FHW21] Simon Foster, Chung-Kil Hur, and Jim Woodcock. Formally Verified Simulations of State-Rich Processes using Interaction Trees in Isabelle/HOL, 2021.

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

[GLN08] Jean Goubault-Larrecq, Slawomir Lasota, and David Nowak. Logical relations for monadic types. *Math. Struct. Comput. Sci.*, 18(6):1169–1217, 2008.

[GM02] Pietro Di Gianantonio and Marino Miculan. A Unifying Approach to Recursive and Co-recursive Definitions. In *Types for Proofs and Programs*, 2002.

[Gre23] Simon Oddershede Gregersen. *Higher-Order Separation Logic for Distributed Systems and Security*. PhD thesis, Aarhus University, 2023.

[GSC+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 653–669, USA, 2016. USENIX Association.

[GSK+18] Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 646–661, New York, NY, USA, 2018. Association for Computing Machinery.

[GSS+22] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations. *Proc. ACM Program. Lang.*, 6(POPL), Jan 2022.

[HDNV12] Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. The marriage of bisimulations and Kripke logical relations. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 59–72. ACM, 2012.

[HNDV13] 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*, POPL '13, pages 193–206, New York, NY, USA, 2013. ACM.

[HPP06] Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1):70 – 99, 2006. Clifford Lectures and the Mathematical Foundations of Programming Semantics.

[HRRS20] Claudio Hermida, Uday S. Reddy, Edmund P. Robinson, and Alessio Santamaria. Bisimulation as a Logical Relation. *CoRR*, abs/2003.13542, 2020.

[HS00] Peter Hancock and Anton Setzer. Interactive Programs in Dependent Type Theory. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, page 317–331, Berlin, Heidelberg, 2000. Springer-Verlag.

[JG09] Patricia Johann and Neil Ghani. A principled approach to programming with nested types in Haskell. *High. Order Symb. Comput.*, 22(2):155–189, 2009.

[JJKD17] 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), Dec 2017.

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek

Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[JSS+15] 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 Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[KHM+15a] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A Formal C Memory Model Supporting Integer-Pointer Casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 326–335, New York, NY, USA, 2015. Association for Computing Machinery.

[KHM+15b] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A Formal C Memory Model Supporting Integer-Pointer Casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 326–335, New York, NY, USA, 2015. Association for Computing Machinery.

[KI15] Oleg Kiselyov and Hiromi Ishii. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 94–105, New York, NY, USA, 2015. Association for Computing Machinery.

[KJB+17] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The Essence of Higher-Order Concurrent Separation Logic. pages 696–723, Mar 2017.

[KJTO+20] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In Peter Müller, editor, *Programming Languages and Systems*, pages 336–365, Cham, 2020. Springer International Publishing.

[KKH+16] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight Verification of Separate Compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 178–190, New York, NY, USA, 2016. Association for Computing Machinery.

[KKS+18] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park,

Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. Crellvm: Verified Credible Compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, page 631–645, New York, NY, USA, 2018. Association for Computing Machinery.

[KLL+19] 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*, CPP 2019, pages 234–248, New York, NY, USA, 2019. ACM.

[KMNO14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 179–191, New York, NY, USA, 2014. Association for Computing Machinery.

[KS23] Dexter Kozen and Alexandra Silva. Multisets and Distributions, 2023.

[KSS13] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In Chung-chieh Shan, editor, *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Boston, MA, USA, September 23-24, 2013*, pages 59–70. ACM, 2013.

[KW06] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 141–152. ACM, 2006.

[KW15] Robbert Krebbers and Freek Wiedijk. A typed C11 semantics for interactive theorem proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 15–27. ACM, 2015.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[LABS12] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, Jun 2012.

[Lam99] Leslie Lamport. Specifying Concurrent Systems with TLA+. *Calculational System Design*, pages 183–247, April 1999.

[Ler09] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, Jul 2009.

[LF16] Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair schedul-

ing. In *POPL*, pages 385–399, 2016.

[LF18] Hongjin Liang and Xinyu Feng. Progress of concurrent objects with partial methods. *PACMPL*, 2(POPL):20:1–20:31, 2018.

[LFS14] Hongjin Liang, Xinyu Feng, and Zhong Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *CSL-LICS*, pages 65:1–65:10, 2014.

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284 – 304, 2009. Special issue on Structural Operational Semantics (SOS).

[LG18] Liyi Li and Elsa L. Gunter. IsaK: A Complete Semantics of $\mathbb{K}$. Technical report, University of Illinois at Urbana-Champaign, 2018.

[LG20a] Liyi Li and Elsa Gunter. K-LLVM: A Relatively Complete Semantics of LLVM IR. In *34rd European Conference on Object-Oriented Programming, ECOOP 2020, Berlin, Germany*, 2020.

[LG20b] Liyi Li and Elsa L. Gunter. K-LLVM: A Relatively Complete Semantics of LLVM IR. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:29, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[LGL17] Ugo Dal Lago, Francesco Gavazzo, and Paul Blain Levy. Effectful applicative bisimilarity: Monads, relators, and Howe's method. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.

[LH00] Sheng Liang and Paul Hudak. Modular Denotational Semantics for Compiler Construction. *Lecture Notes in Computer Science*, 1058, May 2000.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 333–343, New York, NY, USA, 1995. Association for Computing Machinery.

[LHJ+18a] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno Lopes. Reconciling high-level optimizations and low-level code in LLVM. *Proceedings of the ACM on Programming Languages*, 2:1–28, Oct 2018.

[LHJ+18b] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. Reconciling High-Level Optimizations and Low-Level Code in LLVM. *Proc. ACM Program. Lang.*, 2(OOPSLA), Oct 2018.

[Li20] Liyi Li. *A verification framework suitable for proving large language translations*. PhD thesis, 2020.

[LKS+17] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. Taming Undefined Behavior in LLVM. *SIGPLAN Not.*, 52(6):633–647, Jun 2017.

[LL97] G. Le Lann. An analysis of the Ariane 5 flight 501 failure-a system engineering perspective. In *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*, pages 339–346, 1997.

[LLH+21] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded Translation Validation for LLVM. PLDI '21, 2021.

[LMNR15] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with ALIVE. PLDI 15, pages 22–32. ACM, 2015.

[LP18] Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, 2018.

[LRGCH18] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular Verification of Programs with Effects and Effect Handlers in Coq. In *FM 2018 - 22nd International Symposium on Formal Methods*, volume 10951 of *LNCS*, pages 338–354, Oxford, United Kingdom, Jul 2018. Springer.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[LW22] Yao Li and Stephanie Weirich. Program Adverbs and Tlön Embeddings. *Proc. ACM Program. Lang.*, 3(ICFP), 2022.

[LXK+22] Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. C4: verified transactional objects. *Proc. ACM Program. Lang.*, 6(OOPSLA):1–31, 2022.

[MAA+19] 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), Jul 2019.

[MAN17] William Mansky, Andrew W. Appel, and Aleksey Nogin. A Verified Messaging System. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct 2017.

[McB15] Conor McBride. Turing-Completeness Totally Free. In Ralf Hinze and Janis Voigtländer,

editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2015.

[MD24] William Mansky and Ke Du. An Iris Instance for Verifying CompCert C Programs. volume 3, New York, NY, USA, 2024. Association for Computing Machinery.

[MDJD22] 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 International Conference on Programming Language Design and Implementation*, PLDI 2022, page 841–856, New York, NY, USA, 2022. Association for Computing Machinery.

[MGD+19] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.*, 3(POPL), Jan 2019.

[MGGA16] William Mansky, Elsa L. Gunter, Dennis Griffith, and Michael D. Adams. Specifying and executing optimizations for generalized control flow graphs. *Science of Computer Programming*, 130:2–23, Nov 2016. Funding Information: This material is based upon work supported in part by NSF Grant CCF 13-18191 . Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. Publisher Copyright: © 2016 Elsevier B.V.

[MGZ15] William Mansky, Dmitri Garbuzov, and Steve Zdancewic. An Axiomatic Specification for Sequential Memory Models. In *Computer Aided Verification - 27th International Conference, CAV 2015*, 2015.

[MHRVM20] Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. The next 700 Relational Program Logics. *Proc. ACM Program. Lang.*, 4(POPL), Dec 2020.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. Foundations of Computing Series. The MIT Press, 1996.

[MML+16] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the Depths of C: Elaborating the de Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 1–15, New York, NY, USA, 2016. Association for Computing Machinery.

[MN17] David Menendez and Santosh Nagarakatte. Alive-Infer: Data-driven Precondition Inference for Peephole Optimizations in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 49–63. ACM, 2017.

[Mog89a] Eugenio Moggi. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth*

Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989, pages 14–23. IEEE Computer Society, 1989.

[Mog89b] Eugenio Moggi. Computational lambda-calculus and monads. pages 14–23, Jun 1989. Full version, titled *Notions of Computation and Monads*, in Information and Computation, 93(1), pp. 55–92, 1991.

[Mog90] Eugenio Moggi. An Abstract View of Programming Languages. Technical Report ECS-LFCS-90-113, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1990.

[Mog91] Eugenio Moggi. Notions of Computation and Monads. *Inf. Comput.*, 93(1):55–92, 1991.

[NBG13] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Dependent Type Theory for Verification of Information Flow and Access Control Policies. *ACM Trans. Program. Lang. Syst.*, 35(2):6:1–6:41, 2013.

[Nec97] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery.

[NHK+15] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, page 166–178, New York, NY, USA, 2015. Association for Computing Machinery.

[NLWSD14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating State Transition Systems for Fine-Grained Concurrent Resources. In Zhong Shao, editor, *Programming Languages and Systems*, pages 290–310, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[OMKT16] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional Big-Step Semantics. In Peter Thiemann, editor, *Programming Languages and Systems*, pages 589–615, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[OP99] Peter W. O'Hearn and David J. Pym. The Logic of Bunched Implications. *The Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In *Annual Conference for Computer Science Logic*, 2001.

[PA19] Daniel Patterson and Amal Ahmed. The next 700 Compiler Correctness Theorems (Functional Pearl). *Proc. ACM Program. Lang.*, 3(ICFP), Jul 2019.

[PG14] Maciej Piróg and Jeremy Gibbons. The Coinductive Resumption Monad. In Bart Jacobs, Alexandra Silva, and Sam Staton, editors, *Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA, June 12-15, 2014*, volume 308 of *Electronic Notes in Theoretical Computer Science*, pages 273–288. Elsevier, 2014.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 71–84, New York, NY, USA, 1993. Association for Computing Machinery.

[PP03a] Gordon D. Plotkin and John Power. Algebraic Operations and Generic Effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

[PP03b] Gordon D. Plotkin and John Power. Algebraic Operations and Generic Effects. *Appl. Categorical Struct.*, 11(1):69–94, 2003.

[PP09] Gordon Plotkin and Matija Pretnar. Handlers of Algebraic Effects. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, page 80–94, Berlin, Heidelberg, 2009. Springer-Verlag.

[PP13] Gordon D Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, 9(4), Dec 2013.

[Rc10] Grigore Roşu and Traian Florin Şerbănută. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397 – 434, 2010. Membrane computing and programming.

[Rey02] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, page 55–74, USA, 2002. IEEE Computer Society.

[RGL+23] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.*, 7(PLDI), Jun 2023.

[Ros17] Grigore Rosu. K - A Semantic Framework for Programming Languages and Formal Analysis Tools. In Doron Peled and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, NATO Science for Peace and Security. IOS Press, 2017.

[RPS+19a] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at Large: A Survey of Engineering of Formally Verified Software. *Found. Trends Program. Lang.*, 5(2–3):102–281, Sep 2019.

[RPS+19b] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. QED at

large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2-3):102–281, 2019.

[RS14] Grigore Rosu and Traian Florin Serbanuta. K Overview and SIMPLE Case Study. In *Proceedings of International K Workshop (K'11)*, volume 304 of *ENTCS*, pages 3–56. Elsevier, Jun 2014.

[San12] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, USA, 2nd edition, 2012.

[SB19] Wouter Swierstra and Tim Baanen. A Predicate Transformer Semantics for Effects (Functional Pearl). *Proc. ACM Program. Lang.*, 3(ICFP), Jul 2019.

[SBCA15] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 275–287, New York, NY, USA, 2015. Association for Computing Machinery.

[SCK+19] Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.*, 4(POPL), Dec 2019.

[SCL+23] 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.

[ŠevčíkVN+13] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM*, 60(3):22, 2013.

[SGG+21] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. Transfinite Iris: Resolving an existential dilemma of step-indexed separation logic. In *PLDI*, 2021.

[SHC+23] Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. Semantics for Noninterference with Interaction Trees. In *Proceedings of the 37th Annual European Conference on Object-Oriented Programming (ECOOP 2023)*, 2023.

[SHK+16] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.

[SHL+22] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-

Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: Verification of Machine Code against Authoritative ISA Semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 825–840, New York, NY, USA, 2022. Association for Computing Machinery.

[SJT+23] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. Grove: A Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 113–129, New York, NY, USA, 2023. Association for Computing Machinery.

[SLK+21] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 158–174, New York, NY, USA, 2021. Association for Computing Machinery.

[SSS+23] 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.

[Ste94] Guy L. Steele. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 472–492, New York, NY, USA, 1994. Association for Computing Machinery.

[SvdW11] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Math. Struct. Comput. Sci.*, 21(4):795–825, 2011.

[Swi08] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

[SWYS23] 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, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[SZ21] Lucas Silver and Steve Zdancewic. Dijkstra Monads Forever: Termination-Sensitive Specifications for Interaction Trees. *Proc. ACM Program. Lang.*, 5(POPL), Jan 2021.

[TB21] Amin Timany and Lars Birkedal. Reasoning about Monotonicity in Separation Logic. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 91–104, New York, NY, USA, 2021. Association for Computing Machinery.

[Tea20] The Coq Development Team. The Coq Proof Assistant, version 8.11.0, Jan 2020.

[TGS+21] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal. Trillium: Unifying Refinement and Higher-Order Distributed Separation Logic. *CoRR*, abs/2109.07863, 2021.

[TJH17] Joseph Tassarotti, Ralf Jung, and Robert Harper. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP*, volume 10201 of *LNCS*, pages 909–936, 2017.

[Wad90] Philip Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 61–78, New York, NY, USA, 1990. Association for Computing Machinery.

[WL20] Matthew Z. Weaver and Daniel R. Licata. A Constructive Model of Directed Univalence in Bicubical Sets. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, page 915–928, New York, NY, USA, 2020. Association for Computing Machinery.

[WWS19] Yuting Wang, Pierre Wilke, and Zhong Shao. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.*, 3(POPL), Jan 2019.

[XZH+20] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction Trees. *Proceedings of the ACM on Programming Languages*, 4(POPL), 2020.

[Yan07] Hongseok Yang. Relational separation logic. *Theor. Comput. Sci.*, 375:308–334, 2007.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6):283–294, Jun 2011.

[YZZ22] Irene Yoon, Yannick Zakowski, and Steve Zdancewic. Formal Reasoning about Layered Monadic Interpreters. *Proc. ACM Program. Lang.*, 6(ICFP), Aug 2022.

[Zal21] Vadim Zaliva. HELIX: From Math to Verified Code. Jan 2021.

[ZBY+21] 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 2021.

[ZHHZ20] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An Equational Theory for Weak Bisimulation via Generalized Parameterized Coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, 2020.

[ZHK+21] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In Liron Cohen and Cezary Kaliszyk, editors, *12th*

International Conference on Interactive Theorem Proving (ITP 2021), volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[ZNMZ12] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 427–440, New York, NY, USA, 2012. Association for Computing Machinery.

[ZNMZ13] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proc. 2013 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, 2013.