Modular, Compositional, and Executable Formal Semantics for LLVM IR

YANNICK ZAKOWSKI, Inria, France
CALVIN BECK, University of Pennsylvania, USA
VADIM ZALIVA, Carnegie Mellon University, USA
IRENE YOON, University of Pennsylvania, USA
ILIA ZAICHUK, Digamma.ai
STEVE ZDANCEWIC, University of Pennsylvania, USA

What do we want from our representations and proof methodologies to best support the large-scale development of mechanized formal semantics of programming languages? We want compositionality, to allow semantics to be given to subcomponents of programs, modularity, to alleviate the burden of carrying unwieldy invariants about whole programs, and executability, to enable the validation of a model’s correctness during development. In this paper, we propose an alternative to the time-tested technique of using relationally-specified transition systems for verified compilation. Instead, we use interaction trees, an executable free monad equipped with a rich equational theory, to define more modular, compositional, and executable formal semantics implemented in Coq. We explore this approach in VIR, the formalization of a large, sequential subset of LLVM IR. We validate the approach both by proving relevant formal properties and via an ongoing extensive case study that seeks to verify the compilation of domain-specific language for numerical algorithms to LLVM in the HELIX system.

1 INTRODUCTION

The CompCert [24] C compiler was pivotal to the history of verified compilation, paving the way to large-scale software verification of real-world programming languages [36]. Its introduction provided the backbone for a variety of innovative technologies [2, 3, 11, 37, 38] and energized similar verification efforts for other programming languages [14, 20, 49].

Most of these projects define the semantics of the programming language using the time-tested techniques of relationally-specified transition systems given by small-step operational semantics. From a proof-technique standpoint, these approaches often rely on (backward) simulations, carefully crafting elementary simulation diagrams and stitching them together co-inductively to obtain termination-sensitive results.1

Their widespread success speaks to the viability of these fundamental techniques. However, they have some drawbacks. First, they often lack compositionality: the desired small-step operational semantics is not usually definable purely by induction on syntax (as would be achieved by a more denotational approach). 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

1There are some notable exceptions—see Section 7 for a discussion of work by Chlipala [5] and Owens, et al. [33].

Authors’ addresses: Yannick Zakowski, Inria, France, zakowski@seas.upenn.edu; Calvin Beck, University of Pennsylvania, USA, hobbies@seas.upenn.edu; Vadim Zaliva, Carnegie Mellon University, USA, vzaliva@cmu.edu; Irene Yoon, University of Pennsylvania, USA, euisuny@cis.upenn.edu; Ilia Zaichuk, Digamma.ai, izaichuk@digamma.ai; Steve Zdancewic, University of Pennsylvania, USA, stevez@cis.upenn.edu.

2020. XXXX-XXXX/2020/9-ART $15.00
https://doi.org/
way to validate the model’s correctness. Lack of executability also precludes the use of tools like QuickChick [21]. An alternative is to write painstakingly hand-crafted interpreters—CompCert and Vellvm went to some lengths in this regard—but that incurs the additional burden of proving the correspondence between the small-step 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 likely to evolve over time. In this paper, 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 LLVM IR. To do so, we make heavy use of interaction trees [43], a recent 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 [29], letting us prove program equivalences largely by induction and elementary rewriting.

We focus on the Low-Level Virtual Machine (LLVM) framework [22] 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 [49, 50], Alive [28, 31], Crelvm [17], and the K-LLVM [27] projects, as well as attempts to characterize LLVM’s undefined behaviors [23], its concurrency semantics [4], and memory models [15].

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 work mentioned above and built using modern proof engineering-techniques—in particular, ITree-based semantics form its core specification technology. 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 complicated to model together. Furthermore, LLVM IR’s semantics leaves some behaviors intentionally underspecified to give optimizations and backend implementations more leeway to be aggressive.

We believe LLVM IR’s complexities make it all the more important to formalize, and that doing so establishes the compositionality and modularity of ITrees-based semantics. While the work by Xia, et al. demonstrated ITrees in a “toy” setting, here we aim to use them at scale.

In summary, this paper 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 distinguishes 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 [35] and composed together with no additional syntax. The semantic model is defined in terms of a fully “propositional” (nondeterministic) specification to capture the nondeterministic quirks of the language, but we also implement an executable interpreter that shares almost all of the code with the propositional semantics. Sections 2–4 describe this design, introducing the requisite background about ITrees along the way.

**VIR Metatheory.** We demonstrate how the compositional semantics gives rise to a primitive, but very expressive heterogeneous relational program logic, 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 also lets us prove the correctness of the VIR executable interpreter with respect to the model almost for free. Section 5 covers these results.

**VIR Validation.** We also validate VIR’s design in two ways. The executable interpreter lets us experiment with the VIR semantics and, invaluably, cross-validate against LLVM IR implementations on a (growing) set of test cases. More significantly, we provide evidence of the scalability of our approach by reporting on the ongoing verification of a VIR backend for HELIX, a domain-specific language for implementing high-performance numerical algorithms. This case study, described in Section 6, highlights ITrees’ expressiveness—we used them to define an alternative HELIX semantics and proved it equivalent to its pre-existing formalization—and the use of their relational-reasoning facilities to set-up compiler correctness proofs.

As alluded to above, there is a large body of prior work from which we draw inspiration. Section 7 compares our approach to the closest. Section 8 concludes.

## 2 VIR: A FORMALIZATION OF LLVM IR

The primary focus of this paper is the use 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 covers most features of the sequential fragment of LLVM IR as per its informal specification, 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. For expository purposes, however, we restrict our presentation to a representative subset of VIR.

### 2.1 Syntax

VIR’s syntax is shown on Figure 1. At the top-level, a VIR program is a mutually recursive \( \text{cfg}(\text{mcfg}) \) defined as a set of mutually recursive functions. Each function is a single control-flow-graph \( \text{cfg} \), 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, \( \Phi \)-nodes, a list of instructions, and a terminator. The \( \Phi \)-nodes are used to maintain SSA form [6], 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 \( \text{instr} \) set in a three-address-style format and intended to be executed sequentially after the \( \Phi \)-nodes are set. The instructions we consider 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 bits 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 [τ], and pointers τ*.

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.

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

The void value, none, is a placeholder for operations with no meaningful return values. VIR supports 1, 8, 32 and 64 bit integers 4, but in this paper 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 4.3.5 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 [23]. 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 \(a + \text{poison} \equiv 2 \star \text{poison} \equiv \text{poison} \) hold true. Although accounting for poison entails numerous semantic peculiarities, poison is simply modeled as its own defined value.

In contrast, the undef 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 undef value twice is not guaranteed to return the same value: \(\text{undef}_{i64} + \text{undef}_{i64} \equiv \text{undef}_{i64} \) holds true, but \(\text{undef}_{i64} + \text{undef}_{i64} \neq 2 \star \text{undef}_{i64} \) is an inequality, as the right hand side cannot be odd.

To account for these peculiarities, we introduce underdefined values (uv):

\[
\text{uv} \in \mathcal{V}_{\text{u}} := \uparrow \mathcal{V} \mid \text{undef}_{\tau} \mid \text{op} \mathcal{V}_{\text{u}} \mathcal{V}_{\text{u}}
\]

Underdefined values are a superset of defined values—we write ↑ for the corresponding injection—but they also contain the special value undef\(_\tau\) (we omit the subscript \(\tau\) when the type is unimportant). Extending the semantics of arithmetic operations to a set interpretation of undef\(_\tau\) would prevent

\footnote{We use CompCert’s finite integers in our development.}
us from interpreting two successive “reads” to an undefined value differently. Instead, we can manipulate “symbolic” values built from any supported VIR arithmetic operator over $\mathcal{V}$.

### 3 Interaction Trees: Background

Interaction Trees [43](ITrees) are a data structure that represents effectful and potentially divergent computations. ITrees provide the tools necessary to define semantic domains for denotational semantics of languages achieving three core properties: compositionality – definition by structural recursion –, modularity – definition of effects in isolation –, and executability.

Formally, ITrees are a coinductive variant of the free monad, parameterized by a set of events $E$ and a return type $R$. Events characterize the impure interactions of a computation with its environment: they are represented as a family of types, i.e. are of type $Type \rightarrow Type$, such that each event carries in its type the nature of the value that the environment is expected to provide in response.

The definition of ITrees is shown above. Its underlying functor can do three things: (1) return a pure value of type $R$, (2) perform a silent step, represented as a $\ Tau$ constructor followed by the rest of the computation, or (3) emit a visible event $e$ followed by the remaining computation. In this last case, the continuation is parameterized by the value (of answer type $\lambda x. k(x)$) returned by the environment in response to the event.

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 ignores any finite amount of internal computation, and is central to the verification of correctness of program transformations, as described in Section 5.

#### 3.1 Compositional Semantics through rich combinators

ITrees, as a variant of the free monad, naturally enjoy a monadic structure. Better, they support rich fixed point combinators, allowing for compositional definitions of a wide range of semantics.

The $\ ret$ monadic operation embeds the pure computation returning the value $r$. The $\ bind$ operator composes sequentially two ITrees: $\ bind: \ itree \ E A \rightarrow (A \rightarrow \ itree \ E B) \rightarrow \ itree \ E B$. We write $x \leftarrow t ;; k(x)$ for $\ bind \ t \ (\lambda x. k(x))$.

The ITree library provides a trigger $e$ operator, which defines a computation that does nothing but invoke the event $e: \ trigger \ e \triangleq \ Vis \ e \ (\lambda x. Ret \ x)$. When an event $e$ returns an empty type, a polymorphic trigger can be defined, allowing for the event to be raised at any return type: $\ polytrigger \ e \triangleq \ Vis \ e \ (\lambda x. match \ x \ with \ end)$.

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. We first consider the $\ iter$ combinator, allowing to conveniently model iteration and tail recursive calls. Consider its type: $\ iter \ E A B \ (body: \ A \rightarrow \ itree \ E \ (A \oplus B)): \ A \rightarrow \ itree \ E B$. $A$ can be thought of as the type of an accumulator parameterizing the body of the iterator. The execution of the body may result in either a new value for this accumulator over which the body should be executed again, or in a value of a return type $B$, signaling that the iteration has terminated. The $\ iter$ combinator internalizes the recursive calls and ties the knot, resulting in a function that, given an initial accumulator, returns...
an ITree that can only return a final result of type $B$. Note that nothing tells us that the computation will ever return, but these concerns are completely internalized inside of the ITree data-structure.

For non-tail-recursive calls, ITrees support a general combinator for mutually recursive computations: $\text{mrec}$. The combinator is better understood through its type: $\text{mrec} \ (\text{bodies} \ : \ D \leadsto \ \text{itree} \ (D \oplus E)) : D \leadsto \ \text{itree} \ E$. Thinking of $D$ as the event encoding a call to one of the mutually defined functions, and assuming that to each of these possible calls we know an ITree-implementation, the combinator ties the recursive knot and returns computations only interacting through $E$.

### 3.2 Modular Semantics through Event Handlers

ITrees are able to give a modular semantics because they can define impure computation while remaining agnostic about the implementation of these events. The first phase of our semantics will denote a VIR program as such an uninterpreted tree: this is the focus of Section 4.2.

Effects in programming should be understood in a broad sense: any step of computation whose behavior relies on information that is not easily locally available should be modeled as interacting with an environment that will act as an oracle to provide this information. Such a loose definition naturally encompasses interactions with usual notions of state and memory, but also events such as function calls, jumps or failures. Crucially, note that although each event corresponds to a semantic notion, they do not necessarily map directly to syntactic constructs of a given language.

Each event interface $E$ can be associated with a handler that implements the effect of events in $E$ via operations in a monad $M$. Handlers may be lifted to interpreters of interaction trees, which fold over an entire tree in order to embed the computation into $M$. This means that we have $\text{interp} \ (h : E \leadsto M) : \text{itree} \ E \leadsto M$. Importantly, since ITrees themselves form a monad, we do not have to interpret the whole interface at once: for instance, the state monad transformer $\text{StateT} \ S$ allows us to interpret the state events $\text{StE}$ of an ITree of type $\text{itree} (E \oplus \text{StE} \oplus F)$ in isolation.

The semantics of the represented language must be organized in stages of interpretation to retrieve semantic modularity: Section 4.3 covers this concept in detail.

### 3.3 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 interpreter for LLVM described in Section 4.5.

### 4 A COMPOSITIONAL AND MODULAR LLVM SEMANTICS

Interaction trees provide a shallow representation of effectful and diverging computations in Coq. They support combinators for control-flow graphs as well as monadic interpreters [39]. These powerful tools suggest a general methodology for building denotational domains for a wide variety of programming languages. Given a syntax $L$ of any language, we proceed in three steps:

1. Identify and define the set $E$ of events a program $p \in L$ may trigger;
2. By induction on $L$, use the ITree combinators to compute a representation of programs as elements of $\text{itree} \ E$;
3. Define a handler for each family of events in $E$ and use those to interpret the result of step 2.

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

\[ \mathcal{G} \triangleq \text{GRead}^{\mathcal{V}}(i) \mid \text{GWrite}^{\mathcal{V}}(i, v) \]
\[ \mathcal{L} \triangleq \text{LRead}^{\mathcal{V}}(i) \mid \text{LWrite}^{\mathcal{V}}(i, v) \]
\[ \mathcal{S}_\mathcal{L} \triangleq \text{LPush}^{\mathcal{V}}(\text{args}) \mid \text{LPop}^{\mathcal{V}}(\text{args}) \]

Memory model interactions

\[ \mathcal{M} \triangleq \text{MPush}^{\mathcal{V}}(\text{args}) \mid \text{MPop}^{\mathcal{V}}(\text{args}) \]
\[ \text{Load}^{\mathcal{V}}(\tau, i) \mid \text{Store}^{\mathcal{V}}(\tau, a, v) \mid \text{Alloc}^{\mathcal{V}}(\tau) \mid \text{GEP}^{\mathcal{V}}(r, a, v_1, v_2) \mid \text{PtoI}^{\mathcal{V}}(a) \mid \text{ItoP}^{\mathcal{V}}(i) \]

Internal, external, and intrinsic calls

\[ C \triangleq \text{Call}^{\mathcal{V}}(a, u\text{args}) \]
\[ C_\mathcal{E} \triangleq \text{Call}^{\mathcal{V}}(a, d\text{args}) \]
\[ I \triangleq \text{Intrinsic}^{\mathcal{V}}(f, d\text{args}) \]

Nondeterminism and UB

\[ \mathcal{P} \triangleq \text{Pick}^{\mathcal{V}}(u) \]
\[ \mathcal{U} \triangleq \text{UB}^{\mathcal{V}}(u) \]
\[ \mathcal{F} \triangleq \text{Throw}^{\mathcal{V}}(u) \]
\[ \mathcal{D} \triangleq \text{Debug}^{\mathcal{V}}(\text{msg}) \]

Failure and debugging

This section describes how this conceptually concise recipe can be applied to build our formal model of VIR. We inventory VIR’s effects in Section 4.1 and derive from it the sets of events we manipulate. Section 4.2 describes how to represent each syntactic piece of VIR as an interaction tree, building up to the representation of mcfgs. Section 4.3 defines the concrete semantics of each category of effects through the definition of the handler for their corresponding events. Finally, Section 4.4 ties every component together and tackles the initialization of the memory to obtain the complete semantic model of VIR.

4.1 An inventory of LLVM’s events

Figure 2 depicts the eleven categories of events that can be triggered by a VIR program. Although these events are granted meaning during the third step of building a monadic interpreter (see Section 4.3), this preliminary effort is not completely devoid of semantics: at this point we specify the types of the events, which constrain the types of the handlers that will concretely implement their semantics. As a foretaste to its eventual semantic interpretation, each event will be introduced alongside its intended meaning in this section.

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 of global variables, 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 \( \text{LPush}^{\mathcal{V}}(\text{args}) \) event is used to push a fresh local environment that is initialized with an association list of variables to \( \mathcal{V} \)’s, representing the arguments passed to the function. The \( \text{LPop}^{\mathcal{V}}(\text{args}) \) event is used to pop 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 4.3.

Memory events, \( \mathcal{M} \), are much richer. A program can push or pop a (memory) frame, performing an \( \text{MPush}^{\mathcal{V}}(\text{args}) \) or \( \text{MPop}^{\mathcal{V}}(\text{args}) \), within which new storage can be dynamically allocated via the \( \text{Alloc}^{\mathcal{V}}(\tau) \) event. Memory cells can store data with a \( \text{Store}^{\mathcal{V}}(\tau, a, v) \), and data may be loaded from a cell with a \( \text{Load}^{\mathcal{V}}(\tau, i) \). 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). Accessing data inside of an aggregate structure is denoted by \( \text{GEP}^{\mathcal{V}}(\tau, d, d\text{os}) \). Finally, conversions between addresses and integers, \( \text{PtoI}^{\mathcal{V}}(a) \), and reciprocally, \( \text{ItoP}^{\mathcal{V}}(i) \), are supported.\(^5\)

There are three kinds of function calls in VIR: internal calls, external calls, and calls to “intrinsics.” Internal calls, \( C \), should be the result of the denotation of the corresponding function: it can

\(^5\)The support for these constructs crucially relies on the quasi-concrete memory model described in Section 4.3.5.
therefore return any $\mathcal{V}_u$. On the other hand, external calls, $C_E$, will not be resolved internally and can be implemented by any external means: they can only process and return defined values in $\mathcal{V}$. Intrinsic is an LLVM mechanism allowing lightweight language extensions: their names and semantics are standardized, but their addresses cannot be taken—they essentially act as additional instructions. The VIR semantics is parameterized by an extensible set of supported intrinsics, whose associated event, $I$, is isomorphic to $C_E$, but whose interpretation will be handled differently.

LLVM IR is a non-deterministic language. As described in Section 2.2, one source of non-determinism is the *undefined value* $\text{undef}_e$: our semantics tackles this sticking point by manipulating the symbolic underdefined values, $\mathcal{V}_u$, as long as possible. When the computation nonetheless reaches a point requiring a uniquely determined $\mathcal{V}$, an oracle is invoked to pick a defined value out of the symbolic through a so-called $\text{Pick}^{\mathcal{V}}(uv) \in \mathcal{P}$ event.

A second source of non-determinism comes from the undefined behaviors themselves. Undefined behaviors have earned a poor reputation from unsafe languages like C, but they are an essential feature of intermediate representations for compilers. Undefined behavior is used to pass down certain assumptions between different stages of compilation—for instance, that a branch is dead code—such that, if an action could result in undefined behavior, the compiler may assume that it does not occur. Therefore, if a compiler detects that a path of execution leads to undefined behavior, it may substitute any behavior in place of this execution (generally choosing whatever is more efficient, or convenient). 6 Semantically, this means that we need an event to which we can give any meaning. As explained in Section 3, this polymorphism is achieved through an event whose returned type is void, $\mathbb{U}^0 \in \mathcal{U}$. We write $\text{raise} \mathbb{U}$ for the polymorphic triggering of $\mathbb{U}^0$.

Finally, the two last categories of events, $\text{Throw}^0 \in \mathcal{F}$ and $\text{Debug}^0(m) \in \mathcal{D}$ respectively express dynamic errors (representing incomplete feature coverage in our semantics) and dynamic debug messages. We write $\text{fail}$ for the polymorphic triggering of $\text{Throw}^0$.

4.2 Representing VIR programs as Interaction Trees

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

$$\text{virE} \triangleq \mathcal{C} \oplus I \oplus G \oplus (S_L \oplus L) \oplus M \oplus P \oplus \mathcal{U} \oplus D \oplus \mathcal{F}$$

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

$$\llbracket p \rrbracket_{mcfg} (\tau : \text{dtyp} \ (f : \mathcal{V}) \ (args : \text{list} \ (\mathcal{V})) : \text{itree} \ \text{virE} \ \mathcal{V}_u$$

which, given a $\text{mcfg}$ $p$, a return type $\tau$, the address of the starting function $f$, and a list of arguments $\text{args}$, internalizes the semantics into a single interaction tree over the $\text{virE}$ interface.

The definition of $\llbracket \_ \rrbracket_{mcfg}$ 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. Figure 3 provides the signatures of the main functions used to represent syntactic sub-components as non-interpreted interaction trees.

Notice these signatures only expose the type of value returned by the computation, and the interface over which it can trigger effects. They are oblivious to the potential divergence of the evaluation of these components: this concern is entirely internalized into the ITree data-structure.

We have applied this process to all of LLVM IR’s main features in our formal development. For ease of presentation, the following sections will demonstrate how to represent the syntax of VIR,
\[ \begin{align*}
\text{Exp} & \doteq G \oplus L \oplus M \oplus P \oplus U \oplus D \oplus F \\
\text{Ins} & \doteq C \oplus I \oplus \text{Exp} \\
\text{vir} & \doteq C_E \oplus I \oplus G \oplus (S_L \oplus L) \oplus M \oplus P \oplus U \oplus D \oplus F \\
[\text{e}]_e & : \text{itree ExpV}_u \\
[\text{id}, \text{ins}]_f & : \text{itree InsE} () \\
[\text{term}]_b & : \text{itree ExpE (bid + V_u)} \\
[b]_b & : \text{itree InsE (bid + V_u)} \\
[\phi]^{\text{bidsource}}_\phi & : \text{itree ExpE (id + V_u)}
\end{align*} \]

Fig. 3. Representation as ITrees functions: summary of signatures

\[(\uparrow \text{poison}) \oplus _\_ = \uparrow \text{poison} \quad (\uparrow \text{poison}) \odot _\_ = \uparrow \text{poison} \quad \text{concretize_or_pick } wo = \]
\[- \odot (\uparrow \text{poison}) = \uparrow \text{poison} \quad \_ \odot \text{poison} = \text{raiseUB} \quad \text{if is_concrete } wo \]
\[(\uparrow \text{do1}) \oplus (\uparrow \text{do2}) = \text{do1} + \text{do2} \quad (\uparrow \text{do1}) \odot \text{do2} = \text{if } \text{do2} = \text{do1} = 0 \]
\[\text{then raiseUB } \text{else } \text{do1/} \text{do2} \quad \text{else trigger (Pick V (wo)).} \]

Fig. 4. Binary operations on under-defined values

\[\hat{\%}i_e = \text{trigger (LRd}^V (i)) \]
\[\hat{\@i} = \text{do} \leftarrow \text{trigger (GRd}^V (i)) ; \text{ret (do)} \]
\[e_1 + e_2 = u_1 + u_2 \quad e_1 / e_2 = u_1 / u_2 \]

Consider something as simple as a local variable \(\%i\). Its meaning is seen as a computation performing the effect of accessing the local environment to retrieve the value associated to \(i\). Thus, at this stage, it is represented using the elementary \(\text{trigger} ()\) combinator introduced in Section 3, triggering the \(\text{LRd}^V (i)\) event. Notice that the return type of this event is precisely \(V_u\); 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 \(\text{GRd}^V (i)\) event, whose return type is statically guaranteed to contain defined values. We bind the triggered result to \(do \in V\) and inject this bound value into the domain of under-defined values.

Binary operations, like the addition of integers, are represented by (1) taking the ITTree representation of each subexpression \(e_1\) and \(e_2\), and (2) binding the results of these computations to \(u_1, u_2 \in V_u\), respectively, and then (3) performing the basic operation on \(u_1\) and \(u_2\) and returning the result. Division, however, is more complex because division by 0 is undefined behavior. If the denominator is an undefined value, we will need to \(\text{pick}\) a valid concretization, \(do \in V\). In the inlined figure above, we use \(\text{concretize_or_pick}\) for this purpose, which either injects the denominator into \(V\) if it is already concrete, or triggers a \(\text{Pick}^V (u_2)\) event that acts as an oracle for concretizing \(V_u\) values — the semantics of pick operations is given by a separate handler, as discussed in Section 4.3. In Section 5, we will see how the propositional monad transformer offers a solution to modeling undefined behavior and nondeterminism in more detail. Finally, note that the third step of this representation, performing the basic operation, involves a small wrapper for poison cases and to trigger \(U\) events via \(\text{raiseUB}\) when division by 0 occurs, as seen in Figure 4.
4.2.2 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. Instructions do not return any values, so their return type is unit. Their representation function builds upon \(\llbracket \_ \rrbracket_{e}\), as defined in Figure 5. Representing an operation \((id, e)\) reduces to calling \(\llbracket e \rrbracket_{e}\) and binding its result with the trigger of the local write \(\text{LWr}^{(1)}(id, uv)\), thanks to the computability of representation functions and the monadic structure of ITrees.

Memory operations require extra care. Consider \(\text{load}(r, e)\), a memory load operation from an address expression \(e\) of type \(\tau\). The address \(ua\) resulting from \(\llbracket e \rrbracket_{e}\) should be used to trigger the appropriate memory event. However, for simplicity, our memory model can only be indexed by defined memory addresses, and stores defined values. To retrieve a defined value, we therefore resolve any underdefinedness in \(ua\) by picking 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. Thus, the full behavioral specification of \(\text{load}(r, e)\) reads as: evaluate \(e\), pick a concrete address out of the resulting set of valid addresses, if it’s poison, raise undefined behavior, and otherwise load the value stored in memory at this address and bind the result at the provided local variable. Stores and allocations follow a similar pattern.

Finally, we turn to call instructions. There are three kinds of calls: internal, external, and intrinsic calls. The distinction between internal and external calls is a property of the ambient \(\text{mcfg}\), and is not in scope for individual \(\text{cfgs}\). Thus, resolving this differentiation is always done at the level of \(\text{mcfgs}\). In contrast, the list of supported intrinsics is a parameter of our semantics, and they can always be resolved statically. A call \((f, \text{args})\) instruction is represented by first sequentially interpreting the list of arguments \((\text{args})\) using a monadic map, \(\text{mapm}_{\_}\). We then decide if this is an intrinsic call by comparing the name of the function \((f)\) to the known list of intrinsics. If the function is an intrinsic, arguments are concretized to defined values and passed to the dedicated \(I\) event. Otherwise, the address of the function is retrieved from its name, and passed to a function call event. In both cases, the resulting value is bound to the associated local variable \(id\), as usual.
4.2.3 Terminators. Terminators can either return the identity of the next block to be evaluated, or signify the end of the current function call by returning a value. This dichotomy is reflected with a disjoint sum of block identifiers and under-defined values for the return type of the ITrees (See inlined figure 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 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. Finally, all other cases are considered erroneous.

\[
\begin{align*}
\text{return}(e) &\quad \downarrow = \downarrow e \leftarrow \text{ret(\text{inr} u w)} \\
\text{branch}(e, b_l, b_r) &\quad \downarrow = \downarrow e \leftarrow \text{inr}(\text{trigger}(\text{Pick}(u a))) ;
\end{align*}
\]

\[
\text{match } d o \text{ with } \\
|\quad g &\quad \Rightarrow \text{if } g = \text{l} 1 \text{ then ret(\text{inl} } b_l) \text{ else ret(\text{inl} } b_r) \\
|\quad \text{poison} &\quad \Rightarrow \text{raiseUB} \\
|\quad _ &\quad \Rightarrow \text{fail}
\]

4.2.4 Control-flow graphs. We now turn our attention to the representation of VIR functions, i.e. of cfgs. More generally, we want to be able to denote open functions—a subgraph of mutually referential, labeled control-flow-graph blocks that refer to block labels not in the subgraph—in order to reason compositionally about them. Therefore, we will define the representation of a list of blocks: \( \text{list}(\text{block}) \rightarrow \text{bid} \rightarrow \text{itree cfgE(\text{bid} + \mathcal{V}_u)} \) as a function that takes as an argument the label of the block at which to start the computation. This function will have to loop, resolving 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.

Assuming that we have managed to define such a function, defining the representation of a closed cfg is simply a matter of representing its blocks, and interpreting a final label as an error (which would be an invalid jump).

Denoting a list of blocks is, however, a new challenge. The fragments of syntax we have considered so far are quite elementary: they all represent finite computations. This is naturally not the case here since the program may loop indefinitely by jumping through a cycle of blocks. A simple monadic bind is therefore insufficient to sequence the representation of the terminator for a given block with the representation of the next block, we need a fixed-point combinator to tie the knot.

Each jump to a new block can be thought of as a tail recursive call, and thus this looping structure be represented using the iterate operator introduced in Section 3. Indeed, consider for the accumulator type \( A \triangleq \text{bid} \). Re-entering the combinator with a new value of the accumulator therefore consists in taking as argument the label of the new block to evaluate. As for return type, we may either return a value, or end up with the label of a block that does not belong to our open function: we therefore have \( B \triangleq \text{bid} + \mathcal{V}_u \).

We need to consider one last element for the representation of functions: \( \Phi \)-nodes. The meaning of a set of \( \Phi \)-nodes is to assign to a local variable the meaning of an expression based on the label of the previously visited block. Additionally, all \( \Phi \)-nodes need to be executed “in parallel”, 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 a source label \( \text{bid}_u \), we can represent the computation returning the value to be bound at a given \( \Phi \)-node.

\[
\text{[(id, } \Phi \text{args)}]_k^\text{bid}_u = \text{op} \leftarrow \text{args[bid]} \text{ in } u w \leftarrow \text{[op]}_e \leftarrow \text{ret(id, } u w)\]
\]
The representation of a list of Φ–nodes then simply retrieves the association list of identifiers to under-defined values, before performing the local writes.

\[
\llbracket \Phi \rrbracket_{\Phi}^{bids} = dos \leftarrow \text{mapm}(\llbracket \Phi \rrbracket_{\Phi}^{bids}) \cup \text{mapm}(\lambda(id, do). \text{trigger}(\text{LWr}^{-1}(id, do))) \quad \text{dos}
\]

The semantics of Φ–nodes presents a difficulty: it should take place at the beginning of the evaluation of a block, but depends on the identity of the block from which the control flow comes. We therefore view the act of setting Φ–nodes as a part of the meaning of a jump toward a new block: it happens last during the execution of the previous block. (Note that denoting Φ–nodes when jumping rather than when entering a block is still correct with respect to our definition of \([\_]_{cfg}\) because LLVM IR explicitly prohibits entry blocks of functions to contain Φ–nodes.)

We can now put everything together to denote lists of blocks as shown below.

\[
\llbracket \text{bks} \rrbracket_{\text{bks}} = \text{iter body}
\]

where \text{body bids} = \text{try bk} \leftarrow \text{bks [bids]} \text{with ret (inr (inl bids)) in}

\[
bd \leftarrow \llbracket \text{bks} \rrbracket_{\text{bids}} ;
\]

match \text{bd} with

\[
| \text{inr dv} \Rightarrow \text{ret (inr dv)}
\]

\[
| \text{inl bk} \Rightarrow \text{try bk} \leftarrow \text{bks [bids]} \text{with ret (inr (inl bids)) in}
\]

\[
\llbracket \text{bk}(\text{phis})_{\Phi}^{\text{bids}} ; \text{ret (inl bids)}
\]

When \text{mv} is an option value, we write \text{try x} \leftarrow \text{mv} with t in k to bind the content of \text{mv} to x in k if it is a Some constructor, and return t otherwise. We write \text{x} \leftarrow \text{mv} in k \triangleq \text{try x} \leftarrow \text{mv} with fail in k.

4.2.5 Mutually Recursive Control-Flow Graphs. Lastly, we represent \text{mcfgs}, i.e. sets of mutually recursive \text{cfgs}. The main task is tying the knot of function calls, similar to the \text{cfg} blocks in last section. However, the \text{iter} combinator falls short this time: calls are not necessarily tail recursive. We therefore rely on \text{mrec}, the general combinator for mutual recursion introduced in Section 3. Taking the domain of calls \text{D} \triangleq \text{C}, and assuming that we know how to represent any function call as an itree, the \text{mrec} combinator will 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 \text{mcfg}. We can therefore statically know their global identifiers, and build an association list\textsuperscript{7} of type \text{fundefs} \text{:} list (\text{V} \to \text{list} (\text{V})) \to \text{itree virE \text{V}_u))\text{ mapping each function address to its ITree representation. As shown below, the body passed to \text{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. As alluded to in Section 4.1, we need to concretize the arguments in this case.

\[
\llbracket \text{mcfg} \rrbracket_{\text{mcfg fundefs f args}} = \text{mrec body (Call}^{\text{V}_u}(f, \text{args}))
\]

where \text{body (Call}^{\text{V}_u}(uf, \text{args})) =

\[
df \leftarrow \text{trigger (Pick}^{\text{V}}(uf)) ;;
\]

match \text{fundefs} [df] with

\[
| \text{Some f_den} \Rightarrow f_den \text{ (args)}
\]

\[
| \text{None} \Rightarrow \text{dargs} \leftarrow \text{mapm} (\lambda v. \text{trigger (Pick}^{\text{V}}(v))) \text{ args} ;;
\]

\text{trigger (Call}^{\text{V}}(uf, \text{dargs}))

\text{, Vol. 1, No. 1, Article . Publication date: September 2020.}
4.3 Handling Events

Section 4.2 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 6, 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. The order of interpretation is also relevant. As will be developed in Section 5, we want to be able to establish that a program \( p_1 \) refines a program \( p_2 \) in the simplest context allowing the refinement to be established. Our meta-theory establishes that refinement at a given level, entails the refinement at all subsequent levels — we prove this fact in Section 5. By leveraging this result, we can therefore reason about programs more abstractly when their behavior do not rely on certain features of VIR.

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 6) 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 5).

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 propT\( E \) monad of “propositional sets of computations.”

\footnote{Constructing this list happens when initializing the global, top-level state. See Section 4.4.}
4.3.1 \( I \): 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 map associating each name to a semantic function of type 8

\[
\lambda \text{env}. (\text{match } e \text{ with } \\
\text{| LPush}^I (\text{args}) \Rightarrow \\
\text{| GWr}^I (l, v) \Rightarrow \\
\text{| GRd}^V (l) \Rightarrow \\
\text{| LPop}^I \Rightarrow )
\]

\[
\lambda \text{env} : \text{stateTEnv} \ (\text{ittree } E_1) \ = \lambda \text{env} : \text{stateTFrame} + \text{Stack} (\text{ittree } E_3) \ =
\]

\[
\text{match is_intrinsics } (f_{name}) \text{ with } \\
\text{| Some } f \Rightarrow v \leftarrow f \text{ args in ret } v \\
\text{| None } \Rightarrow \text{trigger } (\text{Intrinsic}^V (f_{name}, \text{args}))
\]

8If 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.

Fig. 7. Handlers for Interpretation Levels

4.3.2 \( G \): 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 GRd\(^V\)\((k)\) and GWr\(^I\)\((k, v)\) events into lookups and insertions into this map, respectively. The map \(\text{env}\) is constructed at initialization time and is constant thereafter.

4.3.3 \( L \): Locals. Local variables are handled analogously to globals, a \( L \) event corresponds to a look up of a variable in a map of type EnvL. The scope of local variables will handled by \( S_L \) events.

4.3.4 \( S_L \): Stack. \( S_L \) stack events are triggered when calling a function and returning from a function. These \( S_L \) events, LPush\(^I\)\((as)\) and LPop\(^I\), 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. It is convenient to keep the current frame (which must always exist) as a separate component for the state. We omit a bit of simple “glue code” that lifts handle\(_L\) to work on a pair, compatibly with handle\(_S_L\)—separating them in this way means that we can reason about a function body without even knowing whether there is a stack present.

4.3.5 \( M \): Memory. The handler for VIR’s memory events is far more complex than the handlers described above. Unfortunately, lack of space prohibits us from describing it in full detail—indeed there are numerous papers whose sole purpose is to explain variations on low-level memory model semantics (see Section 7). The VIR implementation is closest to the quasi-concrete model proposed by Kang, et al. [15]. 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 (SByte), which contain actual bytes or representation information. 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. When \( M \) events are handled, they are interpreted into a state
monad containing this map of logical blocks, as well as a list of stack frames. To properly handle
pointer-to-integer 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.

To store a \( V \) into memory, we must first serialize it into a list of SBytes. All of VIR’s \( V \)’s, modulo
addresses, which are a parameter of the semantics, have a known size and thus can easily be
serialized. Pointers, being represented as Coq integers, have a “fake” byte representation where the
entire address is stored in the first SByte and various PtrFrag elements are used to pad the result
to be 8 bytes in order to obtain a 64-bit address. Uninitialized bytes are represented with \( \text{SUndef} \).

One further complication to the model is the use of stack frames. Each time a function is called,
an \( \text{MPush}() \) event is triggered, which when handled should allocate a new stack frame. These stack
frames are implemented as lists that hold the identifiers of all memory blocks allocated during this
function call (unlike for locals, memory allocated by a caller should be accessible in a callee, so we
can’t just allocate a new map of memory each time). When an LLVM function returns, an \( \text{MPop}() \)
event is triggered, and the stack frame is freed.

\[ \text{Alloca}^{V}(\tau), \text{Load}^{Vu}(\tau, a), \text{and Store}^{J}(a, v) \] events are fairly standard. \( \text{Alloca}^{V}(\tau) \) allocates
a new empty block with a size matching \( \tau \) to the current stack frame. \( \text{Store}^{J}(a, v) \) serializes
\( v \) into bytes, storing them at address \( a \) in memory, and triggering failure if \( a \) is not allocated.
\( \text{Load}^{Vu}(\tau, a) \) deserializes the bytes stored at \( a \) in memory, also failing on unallocated addresses. The
\( \text{GEP}^{V}(\tau, do, vs) \) event is slightly more unusual, but implements LLVM’s \text{getelementptr} instruction,
which is used for indexing into aggregate data structures, where \( \tau \) is the type of the structure, \( do \) is
the base address of the structure, and \( vs \) is a list of indices. The final two \( M \) events are \text{PtoIT}^{V}(a)
and \text{ItoP}^{V}(a), which represent pointer-to-integer and integer-to-pointer casts respectively. These
operations are supported by the quasi-concrete memory model.

While this memory model is quite low-level and complex, we expose high level operations for
reading and writing values and arrays that abstract away from the low-level byte representations.
This has proved to be invaluable in our proofs for the HELIX to VIR compiler, since HELIX does
not depend upon bytewise interactions with memory.

4.3.6 \( \mathcal{P} \): Pick. When implementing the handlers for a \( \text{Pick}^{V}(u) \) event, which resolve nondeter-
minism, 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 \text{propT} e A \cong \text{mtree} E A \rightarrow E. 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 \text{executable} versions of the VIR
semantics, we can use any handler that implements \text{one} of the allowable behaviors, but provides
a way to run VIR programs. We will see in Section 5 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:

\[
\text{model\_handle}_{E} \text{Pick}^{V}(u) : \text{propT}_{E_{5}} V = \{ t \mid \begin{cases}
    t \approx \text{fail} & u \in E_{5} \\
    t \approx \text{ret} v & do \in u \\
    t \approx \text{raiseUB} & do \in u \land do \neq \text{poison}
\end{cases} \}
\]

Here, the “\( \approx \)” symbol stands for ITTree equivalence (discussed more below). The set \( [u]_{C} \) all possible
defined values corresponding to \( u \). For example, we have \( [2/\text{undef}_{164}]_{C} = \{ 2, 1, 0, \text{poison} \} \)
because \( 2/2 = 1, 2/1 = 2, 2/0 = \text{poison}, \) and \( 2/n = 0 \) for all other (unsigned) \( n \). Thus, handling
\( \text{Pick}^{V}(2/\text{undef}_{164}) \) might trigger undefined behavior or it might yield one of 0, 1, or 2,
nondeterministically. If there are no concretizations of \( u \), then 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 \( \text{undef} \) in the
underdefined expression $u$ and then evaluating the expression to obtain a defined value.

$$\text{exec.handle}_P \text{ Pick}^V(u) : \text{propT}_E \ V = \text{ret default}(u)$$

### 4.3.7 $U$: Undefined Behavior

Undefined behavior is an important aspect of an intermediate representation like LLVM. VIR represents undefined behavior through $U$ events. A $U$ event is triggered whenever undefined behavior is encountered, either by being directly triggered from the interpretation of the program, as in the case of a store to poison, but also less directly through under-defined values and $P$ events, such as a division by `undef` as described above. As with $P$, there are both propositional and executable handlers.

The $\text{propT}$ handler is trivial: it permits the set of all itrees of the appropriate type:

$$\text{model.handle}_U \text{ UB}^0 : \text{propT}_E \ V = \{ t \mid t : \text{itree} \ E_5 \ V \}$$

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

$$\text{exec.handle}_U \text{ UB}^0 : \text{propT}_E \ V = \text{fail}$$

### 4.4 Stitching the semantics together

We know how to represent our syntax as ITrees, and have defined a handler for each event these trees may trigger. We are almost ready to build our full semantics.

**Lifting the Handlers.** Handlers are defined specifically over the domain $E$ of events they interpret. A bimap operation allows to extend them over an arbitrary disjoint sum $E \oplus F$ where events in $F$ are left untouched. Combined with the interpreter described in Section 3, we can therefore define the interpreters over complete interaction trees as depicted in Figure 6.

**Composing the interpreters.** Having defined the interpreters for each layer of our semantics, it remains to compose them into an interpreter interp\_vir. The order in which we compose these interpreter, depicted on Figure 6, is chosen to delay as far down the chain as possible the introduction of the prop monad.

**Top-level Representation and Interpretation.** At the top-level, an LLVM program is parsed into a VIR representation containing the declarations of globals $^9$, the $\text{mcfg}$, 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 $\lbrack \_ \rbrack_{\text{mcfg}}$:

$$\lbrack \text{prog} \rbrack_{\text{VIR}} \\text{main args mcfg} =$$

$$\text{genv } \leftarrow \text{build_global_env} (\text{prog}) ;;$$

$$\text{defs } \leftarrow \text{mapm} (\lambda \text{cfg}. \ f v \leftarrow \text{trigger} (\text{GRd}^V(\text{cfg}.\text{entry})) ;; \text{ret} (f v, \lbrack \text{cfg} \rbrack_{\text{cfg}}) \text{prog}$$

$$\text{addr } \leftarrow \text{trigger} (\text{GRd}^V(\text{main})) ;;$$

$$\lbrack \text{prog} \rbrack_{\text{mcfg}} \text{defs} (\uparrow \text{addr}) \text{ args}$$

We can at last obtain the full model for VIR as interp\_vir($\lbrack \_ \rbrack_{\text{VIR}}$). Interestingly, if instead of composing all the layers of interpretation, but instead define interp\_vir\_4 stopping the interpretation at the fourth level, we obtain a semantics for VIR interp\_vir($\lbrack \_ \rbrack_{\text{VIR}}$) that do not introduce the prop monad — we come back to this idea in Section 5. Finally, one can also interpret all stages, but using different handlers: the left path on Figure 6 hence define the propositional model, where the right path leads to an executable interpreter for VIR.

---

$^9$We elide the details of the initialization of the global environment, keeping build\_global\_env opaque.
4.5 Running the Executable Semantics

Ultimately, to run our VIR code, we use a minimal driver, written in OCaml, that recursively walks through the ITrees generated by the executable interpreter, performing the appropriate action for each remaining event. We defined executable implementations for the handlers for nondeterministic events in Section 4.3.6 and Section 4.3.7, but this is not quite the full picture, as we have left a few events uninterpreted in Coq: $\mathcal{F}$ (failure), and $\mathcal{C}_E$ (external calls) have semantic content, and it is with respect to these observations that the model is defined. Debug events $\mathcal{D}$ have no semantic content, but we exploit the ability to write event handlers on the OCaml side to allow "printf-style" debugging. The ability to do this for a large Coq formalization is, in our experience, invaluable.

Testing Framework. Our development contains an (unverified) parser from LLVM IR’s surface syntax to VIR’s internal AST, and a pretty printer in the reverse direction. We can therefore insert VIR-verified passes into an otherwise unverified compilation chain and interoperate with LLVM IR-native passes. Executing LLVM IR allows for basic differential testing between our semantics and the LLVM implementation. This is hugely beneficial for determining whether the semantics reflect the real world implementation—a critical step in the case of VIR, which intends to reflect the behavior of existing implementations, despite LLVM only having an informal specification.

For these purposes, we currently have a moderately sized (and growing) suite of LLVM IR programs over which we perform some differential testing between our interpreter, and the Clang compiler. In the future we envision to expand our testing infrastructure to randomly generated programs, in the style of CSmith [44], by leveraging the QuickChick infrastructure [21].

The modular approach taken to defining the semantics of VIR allowed us to get an executable interpreter from the semantics with very little effort, which enabled more substantial testing of the semantics early on. In contrast, both the Vellvm and CompCert projects have spent substantial efforts during their development to define interpreters and relate them to the relational semantics.

5 EQUIVALENCE AND REFINEMENT OF VIR PROGRAMS

One of VIR’s primary goals is to serve as a formal semantics suitable for the verification of LLVM optimization passes, so we require a notion of what it means for an optimization to be correct: we need a refinement relation between LLVM programs. Due to nondeterminism present in the LLVM specification (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 $p$. Defining this behavioral inclusion in a compositional way turns out to be subtle: at some of the stages of interpretation, the nondeterminism is present at the level of the ITree representation. For instance, an expression that yields undef would be denoted by the tree $\text{ret undef}$, which should be refined by the tree $\text{ret 0}$ (or any other i64 constant). For those stages we want a refinement relation $\succeq_R$ on ITrees such that $\text{ret undef} \succeq_R \text{ret 0}$. Subsequent interpretation of the “pick” events generated by observing undef, yields sets of ITrees (represented in Coq by the type $\text{prop} T_E$), so it is only at that point that we obtain the behavioral inclusion $\lbrack \text{trigger} \ (\text{Pick}^V \ (\text{undef})) \rbrack = \{\text{ret } n \mid n \in \text{i64} \} \supseteq \{\text{ret 0} \} = \lbrack \text{ret 0} \rbrack$.

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. As a consequence, we obtain 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 same machinery for proving refinements at the level of ITrees also plays an important role in reasoning about ITree programs more generally, acting as a relational program logic in the style of...
that proposed by Maillard, et al. [29]. We use relational reasoning to prove program transformations correct and to verify the correctness of compilers targeting VIR, as explained below.

5.1 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 Taus) where all values returned along corresponding branches are related by $R$. We omit the definition of $\approx_R$ here, instead focusing on its relevant properties (see these papers [43, 45] for details).

Technically, $\approx_R$ is an equivalence relation only when $R$ is; the usual notion of weak bisimulation is recovered as the instance $\approx_{eq}$, where the relation is chosen to be Coq’s Leibnitz equality, eq, and we leave off the subscript in this case. 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 $t_1 \succeq_R t_2$ 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 8 shows the key equivalences that allow us to exploit the monadic structure and semantics of interpretations. The laws shown above the line in Figure 8 hold for any monad that supports a suitable implementation of the Loop combinator, which includes ITrees and many monads built from them—especially important for the VIR semantics are liftings of ITrees via the state monad transformer, yielding the monad stateT S (itree E). The Coq development uses typeclasses to characterize such monads and to overload $\approx$ with suitable notions of equivalence for each. The laws shown below the line are specific to ITrees, and they explain how Tau and Vis interact with bind. The first of these laws, (Tau t) $\approx t$, lets us ignore any (finite number) of Tau’s, which is where $\approx$ gets the name “equivalence up to taus.”

Figure 9 shows the relational reasoning principles that hold for $\approx_R$, for an arbitrary relation $R$. In the case of refinements, the EuttRet rule establishes the basic relation between values returned by the computation and reflexivity of $R$ ensure that the computation refines itself. In the EuttTrans 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, EuttTrans implies that rewriting with the monad and interpretation laws is sound for refinement: since eq $\circ R = R = R \circ eq$ for any relation $R$. This means that we can string refinements and equivalences together to reach a desired conclusion. For instance, from $t_1 \approx t_2 \succeq_R t_3 \succeq_R t_4 \approx t_5$ we can conclude $t_1 \succeq_R t_5$.

Rule EuttMon says that monotonicity allows us to prove a stronger refinement relation to establish a weaker one, and EuttInterp says that interpretation with respect to the same handler preserves any refinement relation (intuitively, since handlers affect only the visible events of the tree, the leaves remain in the refinement relation). Finally, EuttClosbind (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. EuttClosbind plays a crucial role in reasoning about ITrees—we will see in more detail below how it is used.

Every equivalence and rule stated in Figures 8 and 9 corresponds to a lemma proved in the Coq development. Some, like EuttMon follow by straightforward coinductive proofs. Several of the others are considerably more challenging, but we elide those technical details.

\[\text{For ITrees, the Monad and Interpretation laws hold even for strong bisimulation, which doesn’t ignore Tau’s, allowing us to rewrite using those equations in more contexts, but we gloss over the distinction here for the sake of simplicity.}\]
General Monad Laws

\[ \begin{align*}
(x \leftarrow \text{ret } v \;;; k \ x) & \approx (k \ v) \\
(x \leftarrow t \;;; \text{ret } x) & \approx t \\
(x \leftarrow (y \leftarrow s \;;; t) \;;; u) & \approx (y \leftarrow s \;;; x \leftarrow t \;;; u)
\end{align*} \]

General Interpreter Laws

\[ \begin{align*}
\text{interp } h (\text{trigger } e) & \approx h \_ e \\
\text{interp } h (\text{Ret } t) & \approx \text{ret } r \\
\text{interp } h (x \leftarrow t \;;; k \ x) & \approx x \leftarrow (\text{interp } h \ t) \;;; \text{interp } h (k \ x)
\end{align*} \]

ITree-specific Structural Laws

\[ \begin{align*}
\text{EuttRet} & : \frac{R(r_1, r_2)}{\text{ret } r_1 \approx_R \text{ret } r_2} \\
\text{EuttTrans} & : \frac{t_1 \approx_R t_2 \quad t_2 \approx_R t_3}{t_1 \approx_R t_3} \\
\text{EuttCloBind} & : \frac{t_1 \approx_R t_3 \quad R_1 \subseteq R_2}{(x \leftarrow t_1 \;;; (k_1 \ x)) \approx_R (x \leftarrow t_2 \;;; (k_2 \ x))} \\
\text{EuttMon} & : \frac{t_1 \approx_R t_2 \quad R_1 \subseteq R_2}{(\text{interp } h \ t_1) \approx_R (\text{interp } h \ t_2)}
\end{align*} \]

5.2 Refinement and Interpretation into Prop

Recall that the type \( \text{propT}_E \), defined as \( \forall A, \text{itree } E A \rightarrow P \), (propositionally) represent sets of ITrees (modulo \( \approx \)) and is used by the VIR semantics to model nondeterminism in the language definition. The type \( \text{propT}_E \) is nearly\(^{11}\) a monad, where the return and bind operations are defined as shown below.

Definition 5.1 (\( \text{propT}_E \) operations).

\[ \begin{align*}
\text{ret } (x : A) : \text{propT}_E A & = \lambda (t : \text{itree } E A). t \approx \text{ret } x \\
\text{bind } (\text{spec}_A : \text{propT}_E A) (k_{\text{spec}} : A \rightarrow \text{propT}_E B) : \text{propT}_E B & = \\
\lambda (t : \text{itree } E B). \exists (t_a : \text{itree } E A) \exists (k : A \rightarrow \text{itree } E B). t \approx (x \leftarrow t_a \;;; (k \ x)) \quad \land \ t_a \in \text{spec}_A \quad \land \ \forall (a : A). (\text{returns } t_a \ a) \Rightarrow (k \ a) \in (k_{\text{spec}} a)
\end{align*} \]

Here, \( \text{ret} \) lifts a value into the singleton set containing the pure itree that simply returns the value. The \( \text{bind} \) operation is more interesting; the resulting set contains all trees that can be factored into a subtree \( t_a \) satisfying the predicate \( \text{spec}_A \), bound to a continuation \( k \) that maps every answer \( a \) that might be returned by \( t_a \) to a tree satisfying \( k_{\text{spec}} a \). The returns \( t_a \) predicate is inductively defined and, crucially, might 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 equivalences.

The semantics of nondeterministic operations like pick is given by interpretation into \( \text{propT}_E \). Unlike the computation-level interpreters, once we move to propositionally-defined sets, it is useful to allow those sets to be constrained by refinement relations, so we generalize its type to include a relation on the underlying ITree type:

\[ \text{propT}_E \rightarrow \text{propT}_E \]

\(^{11}\)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 [29] and doesn’t pose a problem because we rarely need to perform rewrites at the specification level.
Definition 5.2 (interp\_prop). Let $h_{\text{spec}} : E \leadsto \text{prop}\_T F$ be a (propositional) handler and $R : R \rightarrow R \rightarrow \mathbb{P}$ be a relation, then $\text{interp}\_\text{prop}_R h_{\text{spec}}$ has type $\text{itree} E R \rightarrow \text{prop}\_T F R$ and is defined as a coinductive predicate satisfying these properties:

- If $R(r_1, r_2)$ and $t_2 \approx \text{ret} r_2$ then $t_2 \in \text{interp}\_\text{prop}_R h_{\text{spec}} (\text{Ret} r_1)$
- If $t_2 \in \text{interp}\_\text{prop}_R h_{\text{spec}} t_1$ then $t_2 \in \text{interp}\_\text{prop}_R h_{\text{spec}} (\text{Tau} t_1)$
- If $t_2 \approx (\text{bind} t_a k_2)$ for some $t_a$ such that $t_a \in (h_{\text{spec}} e)$ and $k_2$ such that $\forall (a : A), (\text{returns} t_a a) \Rightarrow (k_2 a) \in \text{interp}\_\text{prop}_R h_{\text{spec}} (k_1 a)$, then $t_2 \in \text{interp}\_\text{prop}_R h_{\text{spec}} (\forall \text{vis} e k_1)$

This definition of $\text{interp}\_\text{prop}$ satisfies the general interpreter laws in Figure 8. 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 5.3 (Handler Correctness). A handler $h : E \leadsto \text{itree} F$ is correct with respect to a specification $h_{\text{spec}} : E \leadsto \text{prop}\_T F$, written as $h \in h_{\text{spec}}$, if and only if $\forall t e, (h T e) \in h_{\text{spec}} T e$.

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

Lemma 5.4 ($\text{interp}\_\text{prop}$ correct). For any handler $h \in h_{\text{spec}}$, any reflexive relation $R$, and any tree $t : \text{itree} E A$ it is the case that $(\text{interp}_R h t) \in \text{interp}\_\text{prop}_R h_{\text{spec}} t$.

A significantly less trivial property establishes that the analog of the EUTT interp rule from Figure 9 also holds when we interpret into the Prop\_T monad.

Lemma 5.5 ($\text{interp}\_\text{prop}$ respects refinement). For any $h_{\text{spec}}$ and any partial order $R$, if $t_1 \sim_R t_2$, then $(\text{interp}_R h_{\text{spec}} t_1) \supseteq (\text{interp}_R h_{\text{spec}} t_2)$.

5.3 VIR Refinements

The relational machinery defined above lets us give a clean semantics to LLVM’s underspecified values and undefined behaviors. Moreover, we can straightforwardly define appropriate refinement relations that work at any level of interpretation shown in Figure 6 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. For instance if an optimization does not affect memory operations, then there is no need to do any reasoning about how memory events are interpreted. For example, it should be possible to prove that performing constant propagation and dead code elimination yield refined programs at level $L_0$ before interpreting any events, because these transformations do not change whether or not any other event happens, and do not rely upon how any events are handled.

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 VIR. For concrete values this is trivial, refinement is reflexive and anything can refine poison; however, as we have established, LLVM makes great use of (typed) underdefined 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 $\mathcal{V}_\nu$. At the base case we have $[\text{undef}_r]_C = \{ o | o \text{ is a concrete value of type } \tau \}$ where the notation $[x]_C$ represents the set of concrete values of $x$. For instance $[\text{undef}_{i64}]_C$ is the set of all 64-bit integers. Since $\mathcal{V}_\nu$ contains “delayed” computations like $2 \times \text{undef}_{i64}$, the sets are nontrivial. In this case, we have that $[2 \times \text{undef}_{i64}]_C$ is the set of even 64-bit integers.

\begin{footnote}{Except that, as for bind associativity, the bind law holds in only one direction, again due to nondeterminism.}

, Vol. 1, No. 1, Article . Publication date: September 2020.
Definition 5.6 (Uvalue refinement). We say that $a \in V_u$ refines $u \in V_u$ precisely when $[u]_C \supseteq [a]_C$ and we write $u \geq a$ for that relation.

Uvalue refinement, namely $t_1 \approx_2 t_2$, 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: $\top = \lambda x \cdot \top$, which relates all elements of its domain, for those components of the state. Thus, at $L_1$ we use the refinement $t_1 \approx_{T \times \geq} t_2$, and each subsequent state interpretation adds another $T \times -$ to the relation.

A simple consequence of (the state-monad instance of) `EuttInterp` is that refinement at an earlier level implies refinement at later levels after interpreting more events, for instance, we have:

Lemma 5.7 ($L_0$ to $L_1$ refinement). For any global state $g$, if $t_1 \approx_2 t_2$ then $(\text{interp\_global} \ t_1 \ g) \approx_{T \times \geq} (\text{interp\_global} \ t_2 \ g)$.

All of the events in VIR aside from $\cal{P}$ and $\cal{U}$ are interpreted into a state monad, so the connections between these layers are treated analogously to the lemma above. For $\cal{P}$ and $\cal{U}$, which lift their events into the $\text{prop}_E$ monad, we use Lemma 5.5 to lift tree refinement to set inclusion (it is trivial to show that the requisite relation built from $\geq$ reflexive and transitive), which gives us the desired definition of top level refinement for VIR programs.

Soundness of the executable interpreter. One pleasing 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 5.4 by showing that the executable handlers for $\cal{P}$ and $\cal{U}$ are correct with respect to their propositional specifications (which is entirely straightforward, since for $\cal{P}$ the only requirement is that the handler choose a concrete value of the appropriate type and for $\cal{U}$ allows any behavior at all).

Theorem 5.8 (VIR interpreter soundness). For any program $p$, $(\text{interpreter} \ p) \in \text{model} \ p$.

Examples of refinements: undef optimizations. This infrastructure also allows for powerful local reasoning about VIR programs. We are able to prove properties about different structural elements in VIR and lift those to properties about VIR programs as a whole. For instance, we might want to prove that one basic block is equivalent to another in VIR, and then show that swapping these blocks leads to equivalent programs as well. The kind of reasoning that we might want to do is illustrated by `undef` in LLVM. For instance, we might want to refine $r = \text{undef} \cdot \text{undef}$ to $r = \text{undef}$, replacing an expression with a multiplication by a trivial underdefined value. The reasoning behind this is ultimately quite simple, if you have $\text{undef} \cdot \text{undef}$ then we can refine this to $1 \cdot \text{undef}$, as $1$ is a valid concretization of `undef` and thus $1 \cdot \text{undef} = \text{undef}$ is a refinement of `undef`.

5.4 Reasoning via Postconditions

From the point of view of reasoning, we can think of $R$ as a postcondition satisfied by these related computations. 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
HELIX transforms a program in HCOL, a language resembling mathematical notation, through a series of intermediate languages down to machine code, as shown in Figure 10. Each translation gradually lowers the abstraction level and performs optimizations for the target architecture.

Using a variety of verification techniques each translation step from HCOL to FHCOL [46–48] has been proven correct. FHCOL is compiled to a subset of LLVM IR that is compatible with VIR, allowing it to take advantage of the formalization efforts for VIR. This section describes an ongoing effort to prove the correctness of the translation pass from FHCOL to VIR.

Proving the correctness of this translation is an excellent stress test for VIR’s infrastructure, as both languages are significantly different. FHCOL is a highly-specialized imperative DSL designed to operate on fixed-length vectors of floating-point numbers with a relatively simple memory model, consisting of blocks that directly store 64-bit floating point values, inspired by CompCert’s first memory model [25]. In contrast, VIR is a general-purpose, low-level language, equipped with a more complex memory model, as described in Section 4.3.5. Furthermore, the compiler from FHCOL, as well as the big-step semantics for FHCOL, were developed independently, and with no prior knowledge of VIR’s semantics.
The details of this ongoing verification effort are far beyond the scope of this paper\(^\text{13}\). In the remainder of this section, we describe at a high level how the expressiveness of our semantic framework can be leveraged to tackle this problem following the schema shown in Figure 11. The first step is to follow the recipe from Section 4 to define an ITree-based semantics for FHCOL, and prove that this semantics refines the preexisting one. Next, we will state the correctness of the translation, using the generality of the heterogeneous relation \(\approx_R\) to relate the completely different memory models involved. Finally, we take advantage of the compositionality of our approach to prove this correctness theorem by straightforward induction on FHCOL’s syntax, following the natural structure of the compiler. The proofs are conducted in the postcondition-driven style described in Section 5.4.

### 6.1 A sound ITree-based semantics for FHCOL

FHCOL is equipped with a big-step operational semantics. As the language is strongly normalizing, the structure of the semantics is relatively simple: given an FHCOL operator \(\text{op}\), an evaluation context \(\sigma\) and an initial memory \(\text{mem}\), \((\text{eval}_{\text{FHCOL}} \text{op} \sigma \text{mem})\) either fails, or returns the final memory.\(^{14}\) A distinguished pointer \(\text{out}\) is used to contain the vector result of the computation.

We have defined an alternate ITree semantics for FHCOL following VIR’s approach, as described through Section 4, albeit at a smaller scale. An FHCOL operator \(\text{op}\) is represented as an ITree \([\text{op}]_{\text{FHCOL}}\) acting over its own interface of memory and failure events. The memory events are then handled into a state monad over FHCOL’s memory model, similar to the handler from Section 4.3.5, resulting in an interpretation function \(\text{interp}\). Finally, we have formally proven that this denotation is a sound refinement of FHCOL’s big-step semantics:

\[
\text{(eval}_{\text{FHCOL}} \text{op} \sigma \text{mem} = \text{Some mem'}) \rightarrow (\text{interp}_{\text{mem}} ([\text{op}]_{\text{FHCOL}}) \text{mem} \approx \text{Ret} (\text{mem'}, \text{tt}))
\]

The proof proceeds by induction over FHCOL’s syntax, leveraging the equational reasoning described in Figure 8. The new semantics is designed to match up with the original, providing the necessary glue to reason in terms of ITrees. The proof is fairly straightforward\(^{15}\), short of the encoding of FHCOL for loops into ITrees’ iterators.

### 6.2 Verification of the FHCOL to VIR translation

Given ITree representations for FHCOL and VIR, we can now express the correctness of the compiler from FHCOL to VIR using the exact same approach as the one developed in Section 5.3. To this end, we define an equivalence between the top-level notations of FHCOL and VIR programs as follows, glossing over the initialization of the memories:

\[
\text{op} \equiv \text{mcfg} \triangleq (\text{interp}_{\text{mem}} ([\text{op}]_{\text{FHCOL}} \cdot \text{trigger} (\text{Read out}) \text{memH}_{\text{init}}) \approx \text{Top}
\]

\[
(\text{interp}_{\text{vir}} \text{mcfg} \text{mcfgV}_{\text{init}} l_{\text{init}} g_{\text{init}})
\]

where \(\text{Top} (\_\_ \text{vec}) (\_\_ \text{da}) \cdot \exists \text{arr}, \text{da} = [\text{arr}] \land \forall i, \text{arr}[i] = \text{float vec}[i]
\]

This states that the final result in FHCOL, read from the special out pointer, yields an array identical to the one returned by the VIR program.

The compilation of FHCOL’s operators does not exhibit underdefined values and does not cause undefined behaviour, so the equivalence can be established with VIR’s fourth level of interpretation, and we can conveniently reason without introducing the \(\text{prop}\) monad. The final equivalence to

\(^{13}\)For reference, HELIX as a whole is roughly 45 kloc. FHCOL itself contains a cumulative number of 11 operators, and 22 built-in functions. The compiler from FHCOL to VIR is over 1 kloc.

\(^{14}\)The formal definition uses fuel to simplify the proofs of termination guarantees, but this distinction is superficial.

\(^{15}\)The complete definition of the new semantics and the proof of it equivalence to big-step semantics is only 843 loc.
the top level VIR semantics is done via an inclusion of refinements similar to those described in
Lemma 5.7. Stating the correctness of the compiler is then straightforward:

\[ \forall \text{op mcfg}, \text{compile(op)} = \text{Some mcfg} \rightarrow \text{safe op} \rightarrow \text{op} \equiv \text{mcfg} \]

Assuming that the compilation of an operator \( \text{op} \) succeeds, and that the compiled operator is safe,
i.e. does not fail, the resulting \( \text{mcfg} \) is equivalent to \( \text{op} \).

The relation \( \text{Top} \) expresses the notion of functional equivalence relevant to our use case, while
the \( \text{eutt} \) relation itself ensures the result is termination sensitive, and that no external calls may
happen in the VIR code. It however says nothing about how the memories relate at the end of
the executions: such a relation is naturally not strong enough to be established. The approach
therefore relies on the rule \( \text{EuttMon} \) (Figure 9): we strengthen this relation to establish
by induction over the syntactic subcomponents of FHCOL’s operators.

The backbone of the proof relies on a complex memory invariant \( \text{Inv} \) that shows how FHCOL’s
notions of memory and evaluation context relates to VIR’s notions of global state, local state, and
memory. This invariant is a subset of the simulation invariant we would need to spell out for a
backward-simulation-based approach. However, due to the compositionality and modularity of
these ITrees-based approaches, we need only focus on the relation between the memories of the
both languages. We do not need the typical reasoning about control flow, usually encoded through
relations between the respective syntactic terms, or respective program counters.

Furthermore, since we can give meaning to syntactic subcomponents of both FHCOL and VIR,
we are able to easily express the correctness of each code-generation function used by the compiler.
To do so, we establish similarly that the corresponding denotations are related by \( \equiv_R \) for \( R \) a relation
specific to the case considered and locally strengthening \( \text{Inv} \).

We have formalized this infrastructure, and fruitfully applied it to prove correct the compilation
of numerical and arithmetic operations. Scaling to the whole language is a work in progress.

7 RELATED WORK AND DISCUSSION

There is a large literature on formal verification of software artifacts [36]. Here we focus on the
works most closely connected to the VIR development.

**Verified compilers** The CompCert [24] C compiler has proven to be a pivotal development in
the domain of verified compilation by tackling a real world programming language and nontrivial
optimizations formally in the Coq proof assistant [41]. CompCert’s extraordinary success has
fueled numerous projects aiming to expand upon its results. Examples include the addition of
concurrency [37], the support for linking open programs [34, 38], or the preservation of security
properties [3]. Others have developed their own infrastructure in order to tackle different languages:
the CakeML [20] 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 \( \text{whole} \) programs, thereby disallowing linking. A rich line of works [16, 32, 38, 40, 42]
has sought to relax this restriction via compositional simulation techniques. These works have
struck different balances between expressiveness and proof obligations. Patterson and Ahmed [34]
have recently proposed a framework to express these results in a uniform and comparable way.

Another point of comparison comes from the structure of CertiKOS’ [10, 12] certified (concurrent)
abstraction layers. These layers share many properties with the relational reasoning techniques we
describe in Section 5, albeit the connections among such techniques requires further investigations.

The compositionality of ITrees has been leveraged by Xia et al. [43] to define a theory of linking
over an elementary assembly-like language. While we have not yet developed a similar theory at
the scale of VIR, the same approach should apply, allowing us to obtain compositional reasoning
principles of a different flavor than the previously mentioned works.
Non-small step approaches Interactions trees were developed as a general-purpose representation for effectful, interactive, and possibly-divergent code [43] and, besides programming language semantics, have been used for specifying network servers [18]. One of their distinguishing features is the pervasive use of coinduction, which is crucial to support recursion and iteration, but requires sophisticated proof techniques [13, 45]. Leroy and Grall [26] have experimented with coinduction to model divergence in the operational semantics of a lambda calculus, proving type soundness and verifying a compiler.

Two other exceptions to using relational small-step semantics approach are notable. Chlipala [5] verifies a compiler for a language shallowly-embedded in Coq. The language in question is total, and hence does not support 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 [33]—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 simply 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.

Stepping aside from verified compilers, the recent works on modeling infinite behaviors through the use of corules [1, 7] offer a promising new technique to establish termination-sensitive correctness results. It remains however unclear if these techniques are applicable to the mechanized verification of large scale software.

LLVM and C Semantics The Vellvm project [49] has focused its attention on the intermediate representation of the LLVM framework and has verified complex optimizations over it [50]. 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 small step relation parameterized by the whole mcfg considered. Proving any transformation of programs, changing the mcfg in play, therefore requires to relate two distinct semantics, which in turn requires heavy invariants. Our approach leads to a significantly cleaner semantics as emblematically illustrated by the removal of the heavy notion of program counter that Vellvm needs to manipulate.

A number of other projects have formalized various subsets of C [8, 19], or LLVM IR— such as Alive [28, 31], Crellvm [17], and the K-LLVM [27] projects. Others have focused their attention more specifically on characterizing the LLVM’s undefined behaviors [23] and its concurrency semantics [4]. Even more specifically, modeling realists memory models is an active area of research in itself [15, 25, 30]. 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. Nonetheless, none of these works rely on a mechanized denotational semantics as we do, which constitutes the core of our contribution.

LLVM IR’s semantics is complex, but also evolving: subtle interactions between poison and undef have led to recent proposals [23] to simplify the semantics via a freeze instruction, which in VIR would affect where PickV() events occur and simplify the treatment of under-defined values. The most current LLVM language definition adopts a complex “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 partially relieved by the modularity of its semantics.
In this work, we have demonstrated that modern advances in the semantics of programming languages are ready to be applied to the mechanization of industrial-sized languages. We achieve this result by defining a semantics based on interaction trees for large sequential subset of LLVM IR satisfying compositionality, modularity, and executability. We have furthermore argued that exploring such semantics present major benefits to the formal verification of compilers.

The VIR infrastructure raises immediate prospects for future work, including expanded coverage of LLVM IR features. We plan to finish the verification of the HELIX front end, and take this opportunity to explore automating the equational reasoning used in this style of proofs. Executability of VIR's Coq semantics also opens new possibilities: we wish to explore the use of QuickCheck to significantly scale our testing efforts. At a more fundamental level, the viability of the approach to model concurrency remains an open question, though preliminary explorations are promising. We plan to investigate as we work to extend VIR to support LLVM IR’s concurrency primitives.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. mnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES


