Modular, Compositional, and Executable Formal Semantics for LLVM IR

YANNICK ZAKOWSKI, Inria, France
CALVIN BECK, University of Pennsylvania, USA
IRENE YOON, University of Pennsylvania, USA
ILIA ZAICHUK, Taras Shevchenko National University of Kyiv, Ukraine
VADIM ZALIVA, Carnegie Mellon University, USA
STEVE ZDANCEWIC, University of Pennsylvania, USA

This paper presents a novel formal semantics, mechanized in Coq, for a large, sequential subset of the LLVM IR. In contrast to previous approaches, which use relationally-specified operational semantics, this new semantics is based on monadic interpretation of *interaction trees*, a structure that provides a more compositional approach to defining language semantics while retaining the ability to extract an executable interpreter. Our semantics handles many of the LLVM IR's non-trivial language features and is constructed modularly in terms of *event handlers*, including those that deal with nondeterminism in the specification. We show how this semantics admits compositional reasoning principles derived from the interaction trees equational theory of weak bisimulation, which we extend here to better deal with nondeterminism, and we use them to prove that the extracted reference interpreter faithfully refines the semantic model. We validate the correctness of the semantics by evaluating it on unit tests and LLVM IR programs generated by HELIX.

CCS Concepts: • **Software and its engineering** → **Semantics**; **Compilers**; • **Theory of computation** → **Program verification**; **Denotational semantics**.

Additional Key Words and Phrases: Semantics, Monads, Coq, LLVM, Verified Compilation

ACM Reference Format:
Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, Compositional, and Executable Formal Semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (August 2021), 30 pages. https://doi.org/10.1145/3473572

1 INTRODUCTION

The CompCert [Leroy 2009] C compiler was pivotal to the history of verified compilation, paving the way to large-scale software verification of real-world programming languages [Ringer et al. 2019]. Its introduction provided the backbone for a variety of innovative technologies [Appel 2011; Barthe et al. 2020; Gu et al. 2016; Ševčík et al. 2013; Song et al. 2019] and energized similar verification efforts for other programming languages [Bodin et al. 2014; Jung et al. 2017; Kumar et al. 2014; Zhao et al. 2012].

Most of these projects define the semantics of the programming language using *relationally-specified transition systems* given by *small-step operational semantics*. Roughly speaking, such

Authors' addresses: Yannick Zakowski, Inria, France, yannick.zakowski@inria.fr; Calvin Beck, University of Pennsylvania, USA, hobbes@seas.upenn.edu; Irene Yoon, University of Pennsylvania, USA, euisuny@cis.upenn.edu; Ilia Zaichuk, Taras Shevchenko National University of Kyiv, Ukraine, zoickx@knu.ua; Vadim Zaliva, Carnegie Mellon University, USA, vzaliva@cmu.edu; Steve Zdancewic, University of Pennsylvania, USA, stevez@cis.upenn.edu.

2475-1421/2021/8-ART67
https://doi.org/10.1145/3473572

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

These techniques have had widespread success; however, they also have some drawbacks. First, they often lack *compositionality*: the desired small-step operational semantics is not usually definable purely by induction on syntax . Second, and relatedly, they often lack *modularity*: side effects of the language become reified in the step relation, often leading to additional components such as program counters, heaps, or pieces of program text that are needed to define the relation but complicate the invariants needed to reason about it. Finally, because a relational model is not *executable*, it is difficult to test the language semantics during its development, which is a useful way to validate the model's correctness. Lack of executability also precludes the use of tools like QuickChick [Lampropoulos and Pierce 2018]. An alternative is to write painstakingly hand-crafted interpreters—CompCert [Leroy 2009], Vellvm [Zhao et al. 2012], and JSCert [Bodin et al. 2014] went to significant lengths in this regard—but that incurs the additional burden of proving (and maintaining) the correspondence between the operational semantics and the interpreter.

Compositionality, modularity, and executability are critical to ease the design, development, and upkeep of a formal language semantics, especially for large "real world" languages whose features are complex and evolving over time. In this 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 the LLVM. To do so, we draw on classic ideas about how to structure monadic interpreters [Steele 1994] and make heavy use of *interaction trees* [Xia et al. 2020], a recent Coq formalism that provides (1) expressive monadic combinators for defining compositional semantics, (2) effect handlers for the modular interpretation of effectful programs, and (3) a coinductive implementation that can be extracted into an executable definitional interpreter. These features allow for a strong separation of concerns: each syntactic sub-component can be given a self-contained meaning, and each effect of the language can be defined in isolation via an effect handler.

Moving away from traditional small-step operational semantics to an ITrees-based semantics not only simplifies the language definition, but also allows us to explore alternative means of proving compiler and optimization correctness properties. In particular, ITrees support a rich theory of refinement that facilitates relational reasoning proofs, much in the style of Maillard *et al.*'s Dijkstra monads [Maillard et al. 2020], Swierstra and Baanen's predicate transformers [Swierstra and Baanen 2019] or Benton's relational Hoare logic [Benton 2004], letting us prove program equivalences largely by induction and elementary rewriting. Though some of the relevant theory was presented in the paper by Xia *et al.* [Xia et al. 2020], nondeterminism in the LLVM IR prompted us to develop new machinery for working with "propositional interpreters," a key ingredient needed to establish the proof of adequacy of the extracted interpreter.

We focus on the LLVM framework [Lattner and Adve 2004] 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 [Zhao et al. 2012, 2013], Alive [Lopes et al. 2015; Menendez and Nagarakatte 2017], Crellvm [Kang et al. 2018], and K-LLVM [Li and Gunter 2020] projects, as well as attempts to characterize LLVM's undefined behaviors [Lee et al. 2017], its concurrency semantics [Chakraborty and Vafeiadis 2017], and memory models [Kang et al. 2015; Lee et al. 2018]. As witnessed by research activity surrounding it, LLVM IR's semantics isn't straightforward to specify, or even necessarily well-defined. Features like poison, undef, and integer–pointer casts, are complicated to model independently, and even more so together. We believe LLVM IR's complexities make it all the more important to formalize. While the semantics we present here is not the final word on the subject—most notably, the current memory model is not adequate for justifying some useful LLVM IR optimizations—we believe that we have developed the semantic ingredients needed to (eventually) define a "complete" model. Moreover, the emphasis we have put into the modularity of our semantics shall allow us to improve its quality over time to better approach (and react to changes in) "the" LLVM IR semantics.

The new VIR (Verified IR) development described here aims to fill the same niche as Vellvm, sharing that project's goal of being a platform for verified LLVM optimizations and compilers, but incorporating the insights of the works mentioned above and built using modern proof engineering-techniques—in particular, ITree-based monadic semantics form its core specification technology. While the work by Xia, *et al.* demonstrated ITrees in a "toy" setting, here we aim to use them *at scale*—our treatment of LLVM's phi-nodes, mutually recursive functions, undef values, pointers, and other rich data types is all new in comparison. As such, our results also provide a novel and useful recipe for how to formalize large, complicated language semantics in theorem provers based on dependent type theory. In summary, this 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 distinguish different effects: local environment, stack, global identifiers, memory model, nondeterminism, external function calls, *etc.*. These effects are implemented by independent event handlers in the style of algebraic effects [Plotkin and Power 2003] and composed together with no additional syntax. We give a novel semantic model that is defined in terms of a fully "propositional" specification to capture the nondeterministic quirks of the language, but we also implement an executable reference interpreter that shares almost all of the code with the propositional semantics. Sections 2–4 describe this design, introducing the requisite background about ITrees along the way.

Metatheory. We demonstrate how the compositional semantics gives rise to a primitive, but very expressive relational proof method, enabling termination-sensitive refinements of programs to be established without the use of explicit simulation diagrams or coinduction. The model justifies a definition of "correct program transformation" that can be proved at different levels of abstraction, leveraging the modularity of the semantics. In particular, programs that do not involve non-deterministic features can be reasoned about from the perspective of a deterministic semantics. This general-purpose proof infrastructure—many of our metatheoretic results apply to interaction tree semantics broadly and are not specific to VIR—also lets us prove the correctness of the VIR executable interpreter with respect to the model almost for free. Section 5 covers these results.

VIR Validation. We validate VIR in two ways: First, Section 6 describes HELIX [Zaliva et al. 2020], a verified compiler that targets VIR. This case study that demonstrates the utility of VIR, and our metatheory, for proving compiler correctness results. Second, the use of ITrees allows us

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

Fig. 1. A minimal subset of VIR's syntax

to extract an executable interpreter that can be used to experiment with the VIR semantics and, invaluably, cross-validate against LLVM IR implementations. We compare the semantics against the expected LLVM IR behaviors over a suite of more than 140 test cases. This interpreter is also used for properties-based testing with QuickChick, where we randomly generate simple programs to validate our interpreter against other LLVM IR implementations. For larger end-to-end tests, we also run code generated by HELIX. Section 7 describes these experiments.

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

2 VIR: A FORMALIZATION OF LLVM IR

The primary focus of this paper is the use of monadic interpretation of interaction trees to define and reason about a compositional, modular, and executable semantics for a "real-world" programming language as exemplified by LLVM IR. Our formal development[1] covers most features of the core sequential fragment of LLVM IR 11.0.0 as per its informal specification[2], including: the basic operations on 1-, 8-, 32-, and 64-bit integers, `Doubles`, `Floats`, structs, arrays, pointers, and casts; `undef` and `poison`; SSA-structured control-flow-graphs, global data, mutually-recursive functions, and support for intrinsics. The main features that are currently unsupported are: some block terminators (`switch`, `resume`, indirect branching, `invoke`), the `landing_pad` and `va_arg` instructions, architecture-specific floats and opaque types. The list of supported intrinsics is small, but user-extensible. From a semantics perspective, the main limitation of VIR has to do with the interaction between undefined values and the memory model: our implementation is sound, but prohibits the verification of some LLVM IR optimizations. See the discussions in Section 4.3 and 8 for more about these considerations.

For expository purposes, 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 *cfg* (*mcfg*) defined as a set of mutually recursive functions. Each function is a single control-flow-graph (*cfg*), which is a record that holds a name, formal variables binding its arguments, a block identifier as its entry point, and a list of blocks as its operational content.

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

Blocks are records holding an entry label, Φ−nodes, a list of instructions, and a terminator. The Φ−nodes are used to maintain SSA form [Cytron et al. 1991], dynamically assigning different values to a variable depending on the identity of the predecessor block in the control flow. The *code* field contains a list of instructions (*instr*) paired with registers (*id*) destined to receive the value computed by the associated instruction. The code is set in a three-address-style format and intended to be executed sequentially after the Φ−nodes are set. The instructions we consider here are the evaluation of expressions, function calls, and memory operations such as allocation, loads, and stores. Finally, a *terminator* determines how the control flow should continue after a block. We include conditional branches and return statements as terminators.

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

2.2 Dynamic Values

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

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

The void value, `none`, is a placeholder for operations with no meaningful return values. VIR supports 1, 8, 32 and 64 bit integers[3], 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 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 [Lee et al. 2017]. Deferred UB is instrumental for aggressive optimizations, but a semantic subtlety. The `poison` value is a tainting mark: it propagates to all values that depend on it, so equations such as $\texttt{poison} + \texttt{poison} \equiv 2 * \texttt{poison} \equiv \texttt{poison}$ hold true. Although accounting for `poison` entails numerous semantic peculiarities, `poison` is modeled as its own defined value.

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

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

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

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

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

3 INTERACTION TREES: BACKGROUND

Interaction Trees [Xia et al. 2020] (ITrees) are a data structure that represents effectful and potentially divergent computations. ITrees let us define (in Coq) domains for building compositional (denotational) semantics of languages; they modularize the *effects* of such a semantics while still retaining executability.

Formally, the type `itree E R` is a coinductive variant of the *free monad*, parameterized by a set of *events* `E` of kind `Type → Type` and a return type `R`. Events characterize the impure interactions of a computation with its environment, and each event of type `E A` carries with it the response type `A` expected from the environment. ITrees computations can be constructed by: (1) `Ret r`, returning a pure value of type `R`, (2) a silent step, represented as `Tau t`, where t is the rest of the computation, or (3) a *visible event* e followed by the continutation k, given by `Vis e k`. In this last case, k: `A → itree E R` is a function of the value (of answer type `A`) 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 is central to the verification of correctness of program transformations, as described in Section 5.

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

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

Fixed-point combinators allow for modeling loops and recursive programs. The `iter` combinator allows for conveniently modeling iteration and tail recursive calls. Consider its type: `iter` $(body : A \to$ `itree` $E\ (A \oplus B)) : A \to$ `itree` $E\ B$. Here, A can be thought of as the type of an accumulator parameterizing the body of the iterator. Executing the body may result in either a new accumulator value which signals that the body should be executed again, or in a value of type `B`, signaling that the iteration has terminated. For non-tail-recursive calls, ITrees support a general combinator for mutually recursive computations, `mrec` $(defs : D \rightsquigarrow$ `itree (D` $\oplus$ `E))` $: D \rightsquigarrow$ `itree E`, [4] where a D event represents a call to one of the mutually defined functions whose behaviors are given by *defs*. Besides recursive calls in D, the functions might trigger other events, E. The `mrec` combinator ties the recursive knot and returns computations only interacting through E.

Modular Semantics through Event Handlers. ITrees enable a modular semantics because they can define impure computations while remaining agnostic about the implementation of events. The first phase of our semantics denotes a VIR program as such an uninterpreted tree (see Section 4.2).

Each event interface E can also be associated with a *handler* of type $E \rightsquigarrow M$ that implements the effects of E via operations in a monad M. Handlers may be lifted to *monadic interpreters* [Steele 1994] of interaction trees, which fold over a tree, embedding the whole computation into M, so: `interp` $(h : E \rightsquigarrow M) :$ `itree` $E \rightsquigarrow M$. Importantly, since ITrees themselves form a monad, we do not have to interpret the whole interface at once: for instance, the state monad transformer `StateT` S allows us to interpret the state events $\mathtt{StE}_S$ of an ITree of type `itree(E` $\oplus$ $\mathtt{StE}_S \oplus$ `F) A` into `StateT` S `(itree` $(E \oplus F)$ `) A`—the state events are interpreted in isolation.

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

Global and local state

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

Internal, external, and intrinsic calls

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

Memory model interactions

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

Nondeterminism and UB

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

Failure and debugging

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

Fig. 2. VIR events. (Superscripts indicate return types.)

The VIR semantics is organized as stages of interpretation, exploiting modularity (see Section 4.3).

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 reference interpreter for LLVM described in Section 7.

4 A MODULAR LLVM SEMANTICS

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

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

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

This section applies this recipe to build our formal model of VIR. We inventory VIR's effects in Section 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 *mcfg*s. 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. At this point we specify the *types* of the events, which constrain the types of the handlers that will concretely implement their semantics.

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

Local stack events, $\mathcal{S}_{\mathcal{L}}$, provide a fresh local environment for each function call. The $\texttt{LPush}^{()}$ event pushes a fresh local environment initialized with an association list of variables to $\mathcal{V}_u$s, the

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

Fig. 3. Binary operations on under-defined values

arguments passed to the function. The $\texttt{LPop}^{()}$ event pops the stack frame when a function returns. Separating $\mathcal{L}$ and $\mathcal{S}_{\mathcal{L}}$ into two distinct domains of events allows for the denotation of functions to be oblivious to the existence of this stack of states, as will become apparent in Section 4.3.

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

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

LLVM IR is a non-deterministic language. The VIR semantics implements the *undefined value* $\texttt{undef}_\tau$ (recall Section 2.2), by manipulating the symbolic under-defined values, $\mathcal{V}_u$, as long as possible. When the computation nonetheless reaches a point requiring a uniquely determined $\mathcal{V}$, an oracle, modeled by $\texttt{Pick}^{\mathcal{V}}(uv) \in \mathcal{P}$ events, is invoked to choose a defined value.

A second source of non-determinism comes from *undefined behaviors*, which represent exceptional circumstances. If execution leads to undefined behavior, the LLVM semantics says that *any* behavior may substitute for this execution. [5] Semantically, this means that we need an event to which we can give any meaning; this polymorphism is achieved through an event, $\texttt{UB}^{\emptyset} \in \mathcal{U}$, whose returned type is void. We write `raiseUB` for the polymorphic triggering of $\texttt{UB}^{\emptyset}$.

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

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 interface built from the previously described events. More specifically, let us define the top-level interface for LLVM programs:

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

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

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

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

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

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

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

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

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

Binary operations, like the addition of integers, are represented by taking the ITree representation of each subexpression e_1 and e_2, binding the results of these computations to $uv_1, uv_2 \in \mathcal{V}_u$ respectively, and then performing the basic operation on uv_1 and uv_2 and returning the result. Division, however, is more complex because division by 0 is undefined behavior. If the denominator is an under-defined value, we will need to *pick* a valid concretization, $dv \in \mathcal{V}$. We use `pick` for this purpose, which either injects the denominator into $\mathcal{V}$ if it is already concrete, or triggers a $\texttt{Pick}^{\mathcal{V}}(uv_2)$ event that acts as an oracle for concretizing $\mathcal{V}_u$ values. Note that the basic operations must account for poison and trigger undefined behavior via `raiseUB` when division by 0 occurs, as seen in Figure 3.

Instructions. LLVM instructions are represented by a pair (id, ins) of a side-effectful instruction ins and an identifier id destined to receive the result of the operation. Their representation function builds upon $[\![\_]\!]_e$, as defined in Figure 4.

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

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

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

$$[\![(id, e)]\!]_i = uv \leftarrow [\![e]\!]_e \;;;\; \texttt{trigger}\ (\texttt{LWr}^{()}(id, uv))$$

$$\begin{array}{l}
[\![(id, \texttt{load}\ (\tau,\ e))]\!]_i = \\
\quad ua \leftarrow [\![e]\!]_e \;;;\; da \leftarrow \texttt{pick}(ua) \;;; \\
\quad \texttt{match}\ da\ \texttt{with} \\
\quad \mid \texttt{poison} \Rightarrow \texttt{raiseUB} \\
\quad \mid _ \Rightarrow uv \leftarrow \texttt{trigger}\ (\texttt{load}\ (\tau,\ da)) \;;; \\
\qquad \texttt{trigger}\ (\texttt{LWr}^{()}(id, dv))
\end{array}$$

$$\begin{array}{l}
[\![(_, \texttt{Store}^{()}(ev, ea))]\!]_i = \\
\quad uv \leftarrow [\![ev]\!]_e \;;;\; dv \leftarrow \texttt{pick}(uv) \;;; \\
\quad ua \leftarrow [\![ea]\!]_e \;;;\; da \leftarrow \texttt{pick}(ua) \;;; \\
\quad \texttt{match}\ da\ \texttt{with} \\
\quad \mid \texttt{poison} \Rightarrow \texttt{raiseUB} \\
\quad \mid _ \Rightarrow \texttt{trigger}\ (\texttt{Store}^{()}(da, dv))
\end{array}$$

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

$$\begin{array}{l}
[\![(id, \texttt{call}\ (f,\ args))]\!]_i = \\
\quad uvs \ \leftarrow \texttt{mapm}\ [\![_]\!]_e\ args \;;; \\
\quad retv \leftarrow [\\
\qquad \texttt{match}\ is_intrinsics\ (f)\ \texttt{with} \\
\qquad \mid \texttt{Some}\ s \Rightarrow \\
\qquad\quad vs \ \leftarrow \texttt{mapm}\ \ (\lambda\, uv\,.\,\texttt{pick}(uv))\ uvs \;;; \\
\qquad\quad dv \leftarrow \texttt{trigger}\ (\texttt{Intrinsic}^{\mathcal{V}}(s, vs)) \;;; \\
\qquad\quad \texttt{ret}\ (\uparrow dv) \\
\qquad \mid \texttt{None} \Rightarrow \\
\qquad\quad f \ \ \leftarrow [\![f]\!]_e \;;;\; \texttt{trigger}\ (\texttt{Call}^{\mathcal{V}_u}(uf, uvs))] \;;; \\
\quad \texttt{trigger}\ (\texttt{LWr}^{()}(id, retv))
\end{array}$$

Fig. 4. Denoting instructions as ITrees

Terminators. Terminators either return the identity of the next block to be evaluated, or signal the end of the current function call by returning a value. This dichotomy is reflected in the ITree's return type, a disjoint sum of block identifiers and under-defined values (see below). The representation is otherwise as expected: `return` (e) evaluates e and returns its right injection. A `branch` (e, b_l, b_r) evaluates e and performs a case analysis on its result. In the first case, the result is a 1-bit integer, and the value is treated as a boolean to decide which branch to take and thus a block identifier is returned. Branching on a poisoned value is considered an undefined behavior, so a `raiseUB` is returned. All other cases are considered erroneous.

$$\begin{array}{lcl}
[\![\texttt{return}\ (e)]\!]_t & = & uv \leftarrow [\![e]\!]_e \;;;\; \texttt{ret}\ (\texttt{inr}\ uv) \\
[\![\texttt{branch}\ (e,\ b_l,\ b_r)]\!]_t & = & \\
\quad uv \leftarrow [\![e]\!]_e \;;;\; dv \leftarrow \texttt{pick}(ua) \;;; & & \\
\quad \texttt{match}\ dv\ \texttt{with} & & \\
\quad \mid g & \Rightarrow & \texttt{if}\ g =_1 1\ \texttt{then ret}\ (\texttt{inl}\ b_l)\ \texttt{else ret}\ (\texttt{inl}\ b_r) \\
\quad \mid \texttt{poison} & \Rightarrow & \texttt{raiseUB} \\
\quad \mid _ & \Rightarrow & \texttt{fail}
\end{array}$$

Control-flow graphs. We next consider the representation of VIR functions, *i.e.*, of *cfg*s. More generally, we want to be able to denote *open* functions—a subgraph of mutually referential, labeled control-flow-graph blocks that might refer to block labels not in the subgraph—in order to reason compositionally about them. Therefore, we define the representation of a list of blocks: $[\![bks]\!]_{\text{bks}}\ (b_f, b_s)$ as a function that takes as an argument the label b_s of the source block at which to start the computation, as well as the label b_f of the block visited last. This function loops, using the `iter` operator (see Section 3) combinator to resolve the control flow of the mutual references among the blocks, until it either finds a return statement, or computes the label of a block that does not belong to the sub-control flow graph.

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

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

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

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

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

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

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

Defining the representation of a *closed cfg* (right) is simply a matter of representing its blocks and interpreting a final label as an error (an invalid jump).[6]

Mutually Recursive Control-Flow Graphs. Lastly, we represent *mcfg*s, i.e. sets of mutually recursive *cfg*s. The main task is tying the recursive knot of function calls, similar to the *cfg* blocks. However, the `iter` combinator falls short this time: calls are not necessarily tail recursive. We therefore rely on a more general `mrec` combinator, to tie the knot for us by dynamically unrolling function calls. Conveniently, LLVM IR is a first order language: all (internal) functions are defined at the top-level, as part of the *mcfg*. We can therefore statically know their global identifiers, and build an association list[7] of type $fundefs : \texttt{list}\ (\mathcal{V} * (\texttt{list}\ (\mathcal{V}_u) \to \texttt{itree}\ \texttt{virE}\ \mathcal{V}_u))$ mapping each function address to its ITree representation. As shown below, the body passed to `mrec` can therefore simply query this list to know if the function being called is internal, in which case it returns its representation. Otherwise, it triggers back the call, this time explicitly classified as external. As alluded to in Section 4.1,

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

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 5, the full VIR semantic model is given by a "tower of interpreters" which interpret events to different levels. Level 0 corresponds to the uninterpreted ITree. Each subsequent level handles some events using an appropriate instance of `interp`. For example, the interpreter from Level 0 to Level 1 handles intrinsic events only, whereas by Level 2 both intrinsic events and global events have been handled. As will be developed in Section 5, we want to be able to

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

[7] Constructing this list happens when initializing the global, top-level state. See Section 4.4.

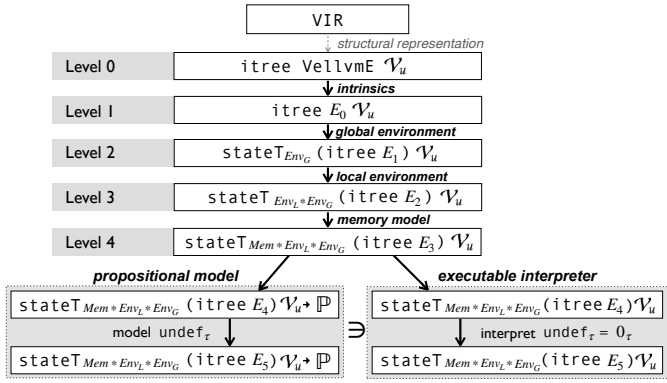

Fig. 5. Levels of interpretation

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

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

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

Fig. 6. Handlers for Interpretation Levels

establish that a program p_1 refines a program p_2 in the simplest monad allowing the refinement to be established.

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 5) and an *executable* VIR *interpreter* (the right path). As explained below, the model accounts for nondeterminism in the VIR semantics by interpreting some events *propositionally* (*i.e.*, into sets characterized by Coq predicates), making them suitable for specification but not extraction, whereas the executable interpreter concretizes the nondeterminism, which is useful for testing and debugging. The two semantics share most of the interpretation levels, allowing us to easily prove that the implementation refines the model (see Section 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."

$\mathcal{I}$: Intrinsics. VIR, like LLVM, supports *intrinsic functions* that extend its core semantics (for instance to allow for the implementation of new "primitive" arithmetic operations). Such intrinsics are defined by a map associating each name to a semantic function of type list $(\mathcal{V}) \to \mathcal{V} + \text{err}$, *i.e.*, a pure Coq function that takes a list of $\mathcal{V}$s and produces either an error or a $\mathcal{V}$ as a result. The handler for intrinsics looks up the name, and runs the semantic function on the arguments, returning the result (or raising an error if it fails).[8]

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

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

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

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

$\mathcal{M}$: Memory. The handler for VIR's memory events is far more complex than the handlers described above. Lack of space prohibits us from describing it in full detail. The VIR implementation is closest to the *quasi-concrete* model proposed by Kang, *et al.* [Kang et al. 2015]. Briefly, the quasi-concrete model has a "logical" memory, represented by an integer map to blocks, where each block is an integer map to *symbolic bytes* that contain actual bytes or representation information, including the possibility of undefinedness. Logical addresses are represented as a pair of integers; the first being the index in the map of blocks, and the second representing the offset of the first byte of the value within the block. $\mathcal{M}$ events are handled by interpreting them into a state monad containing this map of logical blocks, as well as a list of stack frames.

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

This memory model, though sound, is a source of misalignment between our semantics and LLVM IR's semantics. Indeed, as described previously, we have taken care of introducing under-defined values in order to make sure that successive reads to an instance of `undef` could lead to different results: it behaves as a random variable. However, this memory model is only able to store a defined value: it collapses the non-determinism via the $\texttt{Pick}^{\mathcal{V}}(\_)$ event when interacting with the memory. This behavior prevents proving the correctness of certain optimizations, such as store forwarding. Other proposed memory models, such as the "twin allocation semantics" by Lee *et al.* [Lee et al. 2018] permit store forwarding, but prohibit other desirable optimizations (such as dead allocation elimination). It remains an open research question how best to fully model the LLVM's complex memory semantics, but the modularity of our handlers should make it easier to adapt VIR as the technology improves.

$\mathcal{P}$: Pick. When implementing the handlers for a $\texttt{Pick}^{\mathcal{V}}(u)$ event, which resolve nondeterminism, there is a bifurcation: The "true" semantic model, which aims to capture *all* the legal behaviors, uses a handler that interprets behaviors into a monad $\texttt{propT}_{\texttt{E}}\ \texttt{A} \triangleq \texttt{itree}\ E\ A \to \mathbb{P}$. This monad represents *sets* of ITrees as Coq predicates, allowing us to use logical quantifiers to express the allowable

nondeterministic behaviors. On the other hand, for *executable* versions of the VIR semantics, we can use any handler that implements *one* of the allowable behaviors, but provides a way to run VIR programs. We will see in Section 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:

$$\texttt{model\_handle}_{\mathcal{P}}\ \texttt{Pick}^{\mathcal{V}}(u) : \texttt{propT}_{\texttt{E}_5}\ \mathcal{V} = \{t \mid \begin{cases} t \approx \texttt{fail} & [\![u]\!]_C = \emptyset \\ t \approx \texttt{ret}\ v & dv \in [\![u]\!]_C \ \wedge\ dv \neq \texttt{poison} \\ t \approx \texttt{raiseUB} & dv \in [\![u]\!]_C \ \wedge\ dv = \texttt{poison} \end{cases} \}$$

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

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

$$\texttt{exec\_handle}_{\mathcal{P}}\ \texttt{Pick}^{\mathcal{V}}(u) : \texttt{propT}_{\texttt{E}_5}\ \mathcal{V}\ =\ \texttt{ret default(u)}$$

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

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

$$\texttt{model\_handle}_{\mathcal{U}}\ \texttt{UB}^{\emptyset} : \texttt{propT}_{\texttt{E}_5}\ \mathcal{V}\ =\ \{t \mid t : \texttt{itree}\ E_5\ \mathcal{V}\}$$

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

$$\texttt{exec\_handle}_{\mathcal{U}}\ \texttt{UB}^{\emptyset} : \texttt{propT}_{\texttt{E}_5}\ \mathcal{V}\ =\ \texttt{fail}$$

4.4 Stitching the Semantics Together

Having represented our syntax as ITrees, and having defined handlers for each event type, we combine them with interp (see Section 3) to obtain intepreters over complete ITrees as depicted in Figure 5. The order in which we compose these interpreters is chosen to keep "simpler" semantics (such as the pure intrinsics) earlier and delay as far down the chain as possible the introduction of the prop monad.

At the top-level, an LLVM program is parsed into a VIR representation containing the declarations of globals[9], the *mcfg*, and the name of the main from which to start the execution. The set of internal functions is fixed and known statically, which allows us to build the association list of function addresses to denotations required by $[\![\_]\!]_{\text{mcfg}}$:

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

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

General Monad Laws

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

General Interpreter Laws

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

ITree-specific Structural Laws

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

Fig. 7. Core equational theory of ITrees.

Finally, we obtain the full semantic model for VIR, `model`, as $\texttt{interp\_vir}(\llbracket \_ \rrbracket_{\text{VIR}})$. If, rather than composing all the layers of intepretation, we instead define $\texttt{interp\_vir}_4$, stopping at the fourth level, we obtain a semantics that does not introduce the `prop` monad—we return to this idea in Section 5. Finally, we can also interpret all stages, but using different handlers: the left path on Figure 5 defines the *propositional model*, where the right path leads to an *executable interpreter* for VIR that we refer to as `interpreter`.

5 VIR EQUIVALENCES AND REFINEMENT

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

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

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 $\approx_R$ (see [Xia et al. 2020; Zakowski et al. 2020] for details), instead focusing on its relevant properties. Technically, $\approx_R$ is an equivalence relation only when R is; the usual notion of weak bisimulation is recovered as the instance $\approx_{\text{eq}}$, where the relation is chosen to be Coq's Leibnitz equality, `eq`, and we leave off the subscript in this case. The $\approx$ relation plays a particular role in that it can be used as a rewriting rule in any $\approx_R$ goal. When R is a preorder (i.e. reflexive and transitive), so is $\approx_R$, and we can think of this relation as a form of tree refinement; in this case we write $t_1 \gtrsim_R t_2$ to emphasize the (potential) asymmetry and think of t_2 as refining t_1.

The ITrees equational theory is defined in terms of $\approx$. Figure 7 shows the key equivalences that allow us to exploit the monadic structure and semantics of interpretations. The general interpreter laws hold for any monad that supports a suitable implementation of the `iter` combinator, which includes ITrees and many monads built from them—especially important for the VIR semantics are

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

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

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

Fig. 8. Relational reasoning principles

the state and propositional monad transformers.[10] The figure also shows laws specific to ITrees, which explain how `Tau` and `Vis` interact with `bind`. The first of these laws, $(\texttt{Tau}\ t) \approx t$, lets us ignore any (finite number) of `Tau`'s, which is where $\approx$ gets the name "equivalence up to taus" from.

Figure 8 shows (selected) *relational* reasoning principles that hold for $\approx_R$, for an arbitrary relation R. In the case of refinements, the ERet rule establishes the basic relation between values returned by the computation, and reflexivity of R ensures that the computation refines itself. In the ETrans rule, we write $R_1 \circ R_2$ for relation composition. For refinements, we have $R \circ R = R$ by transitivity, so indeed tree refinement is also transitive. Moreover, ETrans implies that rewriting with the monad and interpretation laws is sound for refinement: since $\texttt{eq} \circ R = R = R \circ \texttt{eq}$ for any relation R. This means that we can string refinements and equivalences together to reach a desired conclusion. For instance, from $t_1 \approx t_2 \gtrsim_R t_3 \gtrsim_R t_4 \approx t_5$ we can conclude $t_1 \gtrsim_R t_5$.

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

5.2 Interpretation into $\mathbb{P}$

Recall that a predicate $S : A \to \mathbb{P}$ can be thought of as a (propositionally-defined) *set* of values of type A. We write $a \in S$ for the proposition $S\ a$, which indicates that a is an element of S. Similarly, the type $\texttt{propT}_\texttt{E}\ \texttt{A}$, defined as $\texttt{itree}\ E\ A \to \mathbb{P}$, represents a *set* of ITrees, where we additionally treat set membership modulo $\approx$. We use this type in the VIR semantics to model nondeterminism in the language definition. The type $\texttt{propT}_\texttt{E}$ `is` *nearly* a monad,[11] where, intuitively, `ret` x is the singleton set $\{x\}$ corresponding to a deterministic result, but `bind` *spec* k_{spec} must take the union over all possible nondeterministic behaviors allowed by *spec*, when each of those might itself continue via any one of a set of possible behaviors characterized by k_{spec}. The unions are implemented in Coq by existentially quantifying over the possibilities. Formally, we have:

Definition 5.1 ($\texttt{propT}_\texttt{E}\ \texttt{A}$ *operations).*

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

[10] The Coq code uses typeclasses to characterize such monads and to overload $\approx_R$ with suitable notions of refinement.
[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 [Maillard et al. 2020].

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

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

Definition 5.2 (`Returns` *t).*
$(\texttt{Returns}\ t) : A \to \mathbb{P}$ is the smallest set such that

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

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

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

Definition 5.3 (`interp_prop`*).* Let $h_{spec} : E \rightsquigarrow \texttt{propT}_F$ be a (propositional) handler and $R : A \to B \to \mathbb{P}$ be a relation, then $\texttt{interp\_prop}_R\ h_{spec}$ has type $\texttt{itree}\ E\ A \to \texttt{propT}_F\ B$ and is defined as a coinductive predicate satisfying:

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

This definition of `interp_prop` satisfies the general interpreter laws in Figure 7.[12] 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 5.4 (Handler Correctness). A handler $h : E \rightsquigarrow \texttt{itree}\ F$ is *correct* with respect to a specification $h_{spec} : E \rightsquigarrow \texttt{propT}\ F$, written as $h \in h_{spec}$, if and only if $\forall\ T\ e, (h\ T\ e) \in h_{spec}\ T\ e$.

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

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

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

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

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

Fig. 9. Structural VIR equations (excerpt)

Lemma 5.5 (interp_prop correct). *For any handler $h \in h_{spec}$, any reflexive relation R, and any tree t : itree E A it is the case that* (interp h t) $\in$ $\texttt{interp\_prop}_R\ h_{spec}\ t$.

A significantly less trivial property—the proof is fairly tricky and we refer interested readers to the Coq development for details—establishes that the analog of the EInterp rule from Figure 8 also holds when we interpret into the PropT monad.

Lemma 5.6 (interp_prop respects refinement). *For any h_{spec} and any partial order R, if $t_1 \gtrsim_R t_2$, then* $(\texttt{interp\_prop}_R\ h_{spec}\ t_1) \supseteq (\texttt{interp\_prop}_R\ h_{spec}\ t_2)$.

With the above results established, our development uses $\texttt{interp\_prop}_R$ in two ways. In the *definition* of the VIR propositional model (see the left path of Figure 5), we use $R = \texttt{eq}$ (Coq's equality), in which case $\texttt{interp\_prop}_{\texttt{eq}}$ gives us the desired "sets of ITrees" semantics for modeling nondeterminism. On the other hand, we use Lemma 5.6 to reason about that model—in particular, to establish refinement properties, where we pick R to be nontrivial (see Section 5.4).

5.3 Equational Theory for VIR

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

It is a common compiler optimization to perform systematic rewriting of equivalent expressions, often associated with clever mechanisms used to find the optimal sequence of rewriting according to some cost function. Since expressions depend on the state, but (most) do not cause side effects,[13] the correctness of rewriting expression e into f can usually be established sound with respect to a strong notion of equivalence: they are bisimilar, *and* compute exactly the same states, *i.e.*:

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

This equivalence, much stronger than the notion of refinement that completely disregards the computed states (see Section 5.4), can be easily lifted to all contexts without any syntactic conditions about variables in scope. Naturally, this strong equivalence also entails the refinement relation: substitution of f for e is always sound, in any piece of syntax.

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

The equations pertaining to the denotation of open *cfg*s are of particular interest: we highlight a couple of them in Figure 9. Suppose that we are interested in the semantics of a *cfg* composed of

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

two components cfg_1 and cfg_2. Equation `SubCFG1` allows us to simply disregard cfg_1 granted that we *syntactically* check that we are jumping into cfg_2, and that it cannot jump back into cfg_1: we can reason about sequence. Similarly, equation `Flow` helps to reason about branches: if cfg_1 and cfg_2 are (syntactically) completely independent, the semantics of their union is simply the semantics of whichever subgraph we enter. Finally, equation `SubCFG2` states that we can always temporarily forget about cfg_2 by starting execution in cfg_1 (without cfg_2) and then proceeding afterward with the whole graph in scope.

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

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

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

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

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

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

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

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

5.4 VIR Refinements

The refinement machinery defined in Sections 5.1 and 5.2 lets us give a clean semantics to VIR's underspecified values and undefined behaviors. Moreover, we can straightforwardly define appropriate refinement relations that work at any level of interpretation shown in Figure 5 such that refinement at one level implies refinement at the next. This arrangement means that we can prove the correctness of program transformations at whatever level is most suited to the task.

Uvalue refinements. In order to prove refinements between programs we need to know what it means for a value to be a refinement of another in VIR. For concrete values, this is straightforward: refinement is reflexive and anything can refine `poison`. However, as we have established, LLVM makes

use of (typed) under-defined values values, which can represent arbitrary (typed) sets of concrete values. The refinement relation is thus given by inclusion between the sets of concrete values that can be represented by a $\mathcal{V}_u$. At the base case we have $[\![\texttt{undef}_\tau]\!]_C = \{v \mid v \textit{ is a concrete value of type } \tau\}$ where the notation $[\![x]\!]_C$ represents the set of concrete values of x. For instance $[\![\texttt{undef}_{i64}]\!]_C$ is the set of all 64-bit integers. Since $\mathcal{V}_u$ contains "delayed" computations like $2 \times \texttt{undef}_{i64}$, the sets are nontrivial. In this case, we have that $[\![2 \times \texttt{undef}_{i64}]\!]_C$ is the set of *even* 64-bit integers.

Definition 5.7 (Uvalue refinement). We say that $a \in \mathcal{V}_u$ *refines* $u \in \mathcal{V}_u$ precisely when $[\![u]\!]_C \supseteq [\![a]\!]_C$ and we write $u \geq a$ for that relation.

Uvalue refinement, namely $t_1 \approx_{\geq} t_2$, gives us the base notion of what it means for VIR programs to be related at the structural level L_0 in which none of the LLVM events have yet been interpreted. Levels 1-4 introduce pieces of state: the compiler has no responsibility to preserve them, as long as programs exhibit the same series of external calls and returns values are related by $\geq$. Thus, at L_1 we use the refinement $t_1 \approx_{\mathsf{T} \times \geq} t_2$, where T is the total relation. Each subsequent state interpretation adds another $\mathsf{T} \times -$ to the relation.

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

Lemma 5.8 (L_0 to L_1 refinement).
For any global state g, if $t_1 \approx_{\geq} t_2$ then $(\texttt{interp\_global}\ t_1\ g) \approx_{\mathsf{T} \times \geq} (\texttt{interp\_global}\ t_2\ g)$.

For $\mathcal{P}$ and $\mathcal{U}$, this approach falls short as they lift their events into the $\texttt{propT}_\texttt{E}$ monad. We instead use Lemma 5.6 to lift a tree refinement to set inclusion, which gives us the desired definition of top level refinement for VIR programs.

Soundness of the executable interpreter. A pleasing—and very useful—consequence of the above refinement lemmas is that it is almost trivial to prove that the executable VIR interpreter's program behaviors are permitted by the VIR semantics. The following theorem follows directly from Lemma 5.5 by showing that the executable handlers for $\mathcal{P}$ and $\mathcal{U}$ are correct with respect to their propositional specifications (which is entirely straightforward, since for $\mathcal{P}$ the only requirement is that the handler choose a concrete value of the appropriate type and $\mathcal{U}$ allows any behavior at all).

Theorem 5.9 (VIR interpreter soundness). *For any program p,* $(\texttt{interpreter}\ p) \in \texttt{model}\ p$.

5.5 Floyd-Hoare-Style Forward Relational Reasoning

From the point of view of reasoning, we can think of the R of $\approx_R$ as a *relational postcondition* satisfied by two bisimilar computations. Heterogeneous relations $R : A \to B \to \mathbb{P}$, which relate ITrees of *different* return types, are useful when connecting the behaviors of two ITrees, as is typical when reasoning about a compiler's or program transformation's correctness—we describe such a case study in Section 6.

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

When working with VIR in particular, we apply the same recipe but change the granularity of our "atomic" computations: we do not bubble up `trigger`s but instead we bubble up denotations

of individual instruction, keeping their representation opaque and relying on axiomatizations of their behaviors.

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

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

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

An important missing aspect is the ability to express (and use in refinement proofs) functional properties of specific programs. To this end, we introduce a unary interpretation of `eutt`: given $t : \texttt{itree}\ E\ X$ and $P : X \to \mathbb{P}$, we write $t \hookrightarrow P \triangleq t \approx_{(\lambda(x,y)=>P\ x)} t$.[14]

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

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

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

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

6 CASE STUDY: HELIX

Besides allowing us to reason about LLVM IR program transformations and to prove the adequacy of the executable interpreter, it is also important that the VIR semantics be usable for applications like compiler correctness proofs. To that end, we have (in a parallel project) verified the correctness of HELIX, a compiler that targets LLVM IR. HELIX synthesizes high-performance implementations of numerical algorithms, providing a certified compiler for a formally-specified DSL [Zaliva and Franchetti 2018; Zaliva and Sozeau 2019; Zaliva et al. 2020]. HELIX is inspired by SPIRAL [Franchetti et al. 2018], an automated program synthesis tool that generates efficient implementations of computational kernels across a variety of platforms.

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

The full details of the verification effort are beyond the scope of this paper, so we refer interested readers to the HELIX repository for more details[15]. Here we sketch the process of using ITree relational reasoning principles to prove the correctness of the last stage of the compiler, which translates an intermediate language called FHCOL to VIR. Proving the correctness of this translation is an excellent stress test for VIR's infrastructure, as the 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 [Leroy et al. 2012]. In contrast, VIR is a general-purpose, low-level language, equipped with a more complex memory model, as described in Section 4.3. 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. [16]

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 10. 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 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 5.5.

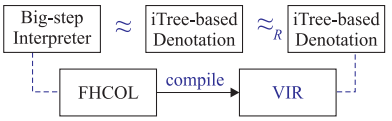

Fig. 10. Verification Methodology

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 op, an evaluation context σ and an initial memory mem, ($\texttt{eval}_{\texttt{FHCOL}}$ op σ *mem*) either fails, or returns the final memory. A distinguished pointer `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 op is represented as an ITree $[\![\texttt{op}]\!]^{\sigma}_{FHCOL}$ acting over a HELIX 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, resulting in an interpretation function `interp_mem`. Finally, we have formally proven that this denotation is a sound refinement of FHCOL's big-step semantics:

$$(\texttt{eval}_{\texttt{FHCOL}}\ \texttt{op}\ \sigma\ \texttt{mem} = \texttt{Some mem}') \Rightarrow (\texttt{interp\_mem}\ ([\![\texttt{op}]\!]^{\sigma}_{FHCOL})\ \texttt{mem} \approx \mathit{Ret}\ (\texttt{mem}', \texttt{tt}))$$

The proof proceeds by induction over FHCOL's syntax, leveraging the equational reasoning described in Figure 7. 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[17], except for the encoding of FHCOL *for loops* into ITrees's iterators.

6.2 Verification of the FHCOL to VIR Translation

Having ITree representations for FHCOL and VIR, we can now express the correctness of the compiler using the approach developed in Section 5.4. This time, rather than relating one VIR

[15] https://github.com/vzaliva/helix

[16] For context, HELIX as a whole is roughly 45 kloc. FHCOL itself contains a cumulative total of 11 operators, and 22 built-in functions. The compiler from FHCOL to VIR is over 1 kloc.

[17] The complete definition of the new semantics and the proof of it equivalence to big-step semantics is only 843 loc

program to another, we instead relate a HELIX op to a VIR *cfg* by interpreting away the different memory events to obtain state-monad transformed ITrees expressible in the same event signatures. On the HELIX side, we write $[\![-]\!]_H^\sigma$ for the resulting semantics. On the VIR side we can use $\texttt{interp\_vir}_4([\![-]\!]_{\text{cfg}})$ because compilation of FHCOL's operators does not exhibit under-defined values and does not cause undefined behavior, so the equivalence can be established with VIR's fourth level of interpretation, and we can conveniently reason without introducing the prop monad.

The theorem below states the core correctness result. It shows that if a HELIX op successfully compiles and is well defined, then running the source op starting from a memory m_H and environment σ is equivalent to running the resulting VIR control-flow graph starting in a related VIR memory m_V, global environment g and local environment l. The invariants also use s_1 and s_2 to track the set of local variables generated by the compiler and ensure that the local environment is modified only in that set.

Theorem 6.1 (FHCOL compilation correctness). *If*

- $\texttt{compile}(\text{op}, s_1) = \texttt{Some}\ (s_2, in, cfg)$ — *successful compilation*
- $\texttt{nofail}([\![\text{op}]\!]_H^\sigma\ m_H)$ — *well-defined source*
- $(\texttt{state\_invariant}\ \sigma\ s_1\ m_H\ (m_V, (l, g)) \wedge \texttt{scoped}\ \sigma\ s_1\ s_2)$ — *state invariant*

then $[\![\text{op}]\!]_H^\sigma\ m_h \approx_Q \texttt{interp\_vir}_4([\![cfg]\!]_{\text{cfg}})(b, in)\ g\ l\ m_V$ *where relation Q is defined such that*

$$Q(m'_H, \_)(m'_V, (l', (g', out))) \Rightarrow (\texttt{state\_invariant}\ \sigma\ s_s\ m'_H\ (m'_V, (l', g')) \wedge \texttt{ok}(out) \wedge \texttt{local\_modif}\ l\ l' \subseteq [s_1, s_2]$$

The backbone of the proof relies on a complex state_invariant predicate 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 $\approx_R$ for R a relation specific to the each case considered and locally strengthening Q. The proofs rely crucially on the relational reasoning principles discussed in Section 5.

7 REFERENCE INTERPRETER VALIDATION

One big benefit of using ITrees as the basis for our Coq semantics is that we are able to extract an executable reference interpreter. To run VIR code, we use a minimal driver, written in OCaml, that traverses an ITrees generated by the semantics, performing the appropriate action for each event. We already defined executable handlers for the nondeterministic events in Section 4.3 and Section 4.3, but there are still $\mathcal{F}$ (failure) and $\mathcal{C}_\mathcal{E}$ (external calls) events that have semantic content. 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.

Our development also 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 hence insert VIR-verified passes into an otherwise unverified compilation chain and interoperate with LLVM IR-native passes.

Testing the interpreter. Executability allows for basic differential testing against another LLVM implementation. The main question is whether our Coq implementation matches the behavior expected for the LLVM IR. Due to the presence of `undef`, an LLVM IR program might have multiple legal behaviors, but an LLVM implementation (and our interpreter) will produce only one, so we include only deterministic test cases in the test suite. Although our interpreter supports external function calls, for simplicity, we tested only programs that don't perform IO; we also considered only programs in the subset of LLVM IR covered by VIR semantics.

Unit Tests. We have developed in conjunction with VIR a battery of unit tests. Each test is a standard `.ll` file containing one or more functions that exercise LLVM IR features and produce numerical outputs. The suite contains 140 total tests, and covers arithmetic operations for various bit-width integers, treatment of `poison`, control-flow operations, phi-nodes, recursion, memory model, aggregate types, *etc.* In all cases, our reference interpreter produces the same output as a corresponding executable compiled via `llc`.[18]

QuickChick. We have also developed QuickChick generators for VIR programs, allowing us to randomly generate programs to use as test cases. The primary property we wish to test is that the VIR interpreter agrees with LLVM proper. Properties in QuickChick are tested by first extracting the property into an ocaml program, making it straightforward to use our extracted interpreter, and additionally it is possible to use Coq's extraction mechanism to expose a function that will serialize our VIR program into a `.ll` file, compile it with clang, and execute it. With this we can perform basic differential testing by comparing the return values from our interpreter and the compiled program, and currently our interpreter agrees across hundreds of generated test cases. At present our QuickChick generators only create fairly straightforward integer programs without loops, but we have demonstrated that it is viable to combine property based testing with the executable interpreter derived from our semantics. We hope to extend this work in the future by generating more interesting programs, but this is beyond the scope of this paper as projects like CSmith[Yang et al. 2011] have demonstrated that this can be a substatial contribution on its own.

HELIX. The HELIX test suite includes eleven tests of varying levels of complexity, obtained by using the verified HELIX compiler described above to produce VIR code. The most complex one calculates a dynamic window monitor for a cyber-physical control system; when compiled it produces 368 lines of LLVM IR code. When run on randomly-generated floating-point vector inputs, our semantics again agrees with that of `llc`.

Although these tests are undoubtedly incomplete, they provide confidence that the VIR semantics agrees with the LLVM IR on a rich enough set of programs to be usable in practice. Moreover, despite being extracted from a purely functional Coq specification, the reference interpreter performs well: it runs the entire test suite in well under 0.1s.

8 RELATED WORK AND DISCUSSION

There is a large literature on formal verification of software artifacts [Ringer et al. 2019]. Here we focus on the works most closely connected to the VIR development.

Verified compilers The CompCert [Leroy 2009] C compiler was a pivotal development in the domain of verified compilation, tackling a real world programming language and nontrivial optimizations formally in the Coq proof assistant [Team 2020]. CompCert's success has fueled numerous projects aiming to expand upon its results. Examples include the addition of concurrency [Ševčík

[18]In our tests, we compared against `llc` LLVM version 11.0.0 compiling for the `x86_64-apple-darwin19.6.0` target running on a 2.4 GHz 8-Core Intel Core i9 processor with 32GB ram.

et al. 2013], the support for linking open programs [Patterson and Ahmed 2019; Song et al. 2019], or the preservation of security properties [Barthe et al. 2020]. Others have developed their own infrastructure in order to tackle different languages: the CakeML [Kumar et al. 2014] project has developed a complete verified chain of compilation for ML

Compositional verification CompCert's original theorem suffered the major restriction of applying only to *whole* programs, thereby disallowing linking. A rich line of works [Kang et al. 2016; Neis et al. 2015; Song et al. 2019; Stewart et al. 2015; Wang et al. 2019] has sought to relax this restriction via compositional simulation techniques. These works have struck different balances between expressiveness and proof obligations. Patterson and Ahmed [Patterson and Ahmed 2019] have recently proposed a framework allowing to compare these result. Another point of comparison comes from CertiKOS'[Gu et al. 2015, 2018] 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.

Non-small step approaches Interaction trees were developed as a general-purpose representation for effectful, interactive, and possibly-divergent code [Xia et al. 2020] and, besides programming language semantics, have been used for specifying network servers[Koh et al. 2019]. One of their distinguishing features is the pervasive use of coinduction, which is crucial to support recursion and iteration, but requires sophisticated proof techniques [Hur et al. 2013; Zakowski et al. 2020]. Leroy and Grall [Leroy and Grall 2009] have experimented with coinduction to model divergence in the operational semantics of a lambda calculus, proving type soundness and verifying a compiler.

Several other exceptions to using relational small-step semantics approach are notable. Chlipala [Chlipala 2010] verifies a compiler for a language shallowly-embedded in Coq. The language in question is total, and hence does not require recursion combinators; nevertheless, this style of semantics admits modular and compositional proof techniques similar to ours. Owens *et al.* advocate for big-step semantics "akin" to an interpreter [Owens et al. 2016]—they use a "clock" for "fuel" to bound recursion, thereby sidestepping the need for coinduction, but requiring proofs to take the fuel into account via step-indexed logical relations. We take this idea a step further and use a true interpreter, embracing the coinductive structure directly. This means that we can more readily reason equationally about ITrees semantics. The tradeoffs between such step-indexed and coinductive approaches deserve more attention.

JSCert The JSCert project [Bodin et al. 2014], which formalizes JavaScript semantics in Coq, uses Charguéraud's "pretty big step" semantics [Charguéraud 2013]. This approach, like interaction trees, promotes compositionality by allowing the semantics to be defined inductively on the syntax; it also uses coinduction to handle diverging terms. Unlike interaction trees, however, "pretty big step" semantics are still defined relationally. The authors implemented a separate executable version of semantics, JSRef, that is intended to serve as a reference implementation. Nontrivial proof effort (we estimate that it takes several thousand lines of Coq code) is required to prove the correspondence of the JSCert pretty-big step relational specification with the JSRef executable version. The authors write: "We believe that both JSCert and JSRef are necessary: JSCert, unlike JSRef, is well-suited for developing inductive proofs about the semantics of JavaScript; JSRef, unlike JSCert, can be used to run JavaScript programs." In contrast, we have shown that interaction trees meet both desiderata: they are well suited both for inductive proofs and for excecutability.

LLVM and C Semantics The Vellvm project [Zhao et al. 2012] has focused its attention on LLVM's intermediate representation and verified complex optimizations over it [Zhao et al. 2013]. The subset of LLVM IR that Vellvm handles is fairly similar to VIR's, albeit marginally outdated and less rich in features. More importantly, their semantics are radically different: Vellvm relies on a traditional small step relation parameterized by the whole *mcfg* considered. Proving any transformation of programs changes the *mcfg* in play and therefore requires to relate two distinct semantics,

which in turn requires heavy invariants. Our approach leads to a significantly cleaner semantics illustrated by the removal of the heavy notion of program counter that Vellvm manipulates.

A number of other projects have formalized various subsets of C [Ellison 2012; Krebbers and Wiedijk 2015; Memarian et al. 2019, 2016], or LLVM IR—such as Crellvm [Kang et al. 2018], K-LLVM [Li and Gunter 2020], and the Alive [Lopes et al. 2015; Menendez and Nagarakatte 2017] projects. The Crellvm project uses the Vellvm semantics internally, so it inherits the same fundamental structure. The K-LLVM framework, implemented in $\mathbb{K}$ [Roşu and Şerbănuţă 2010], is perhaps the most complete executable semantics for the LLVM IR and has been used for extensive testing. There is some work connecting $\mathbb{K}$ specifications to Isabelle/HOL [Li and Gunter 2018], but, to our knowledge, the viability of that approach for formal proofs of, *e.g.*, compiler correctness, remains to be demonstrated.

Alive [Lopes et al. 2015], and its recent successor Alive2 [Lopes et al. 2021], focus on finding bugs in the LLVM IR implementation by using translation validation to check for mis-optimizations. Alive2 is able to run directly on LLVM's source code, and has demonstrated an impressive efficacy. Although their objective, bug-finding, differs from ours, formal verification, both projects share the need for formalizing parts of LLVM IR's semantics. One significant difference from our approach is that Alive2 only formalizes LLVM's semantics *implicitly*, through the encoding its validator performs to check an optimized program. Moreover, the Alive2 semantics properly avoids collapsing `undef`'s non-determinism when interacting with the memory mode—contrary to our current memory model—but it under-approximates its semantics elsewhere (per the paper [Lopes et al. 2021], they "only allow an argument to be either fully `undef` or not `undef` at all"). Moreover, as far as we can tell, Alive2 does not support pointer-to-integer casts. These approximations are sound for bug-finding, only straying Alive2 farther from completeness, but are incompatible with verification. Those differences aside, Alive2 could be a rich source of test cases for VIR. One challenge is that most of the Alive2 test cases aren't executable (they are open program fragments before/after optimization), so it is not clear how we can use them in conjunction with VIR semantics. One could state the expected refinement relation between such program fragments as theorems and try to prove them, but finding a more automatic way of using Alive2 tests, perhaps by making them executable by (randomly) instantiating their free variables, would be desirable.

Others have focused their attention more specifically on characterizing the LLVM's undefined behaviors [Lee et al. 2017] and its concurrency semantics [Chakraborty and Vafeiadis 2017]. Even more specifically, modeling realistic memory models for LLVM is an active area of research in itself [Kang et al. 2015; Lee et al. 2018; Leroy et al. 2012; Mansky et al. 2015], which is closely connected to similar efforts for low-level languages like C [Memarian et al. 2019]. None of these works rely on a mechanized denotational semantics as we do, which constitutes the core of our contribution. Nonetheless, many of these works cover semantic features that VIR does not yet tackle, and as such are major sources of inspirations for the future of VIR. In particular, improving the VIR memory model is an important next step. For example, the memory model presented here does not support storing `undef`, or, more generally, elements of $\mathcal{V}_u$ in the heap, a limitation shared with Vellvm and most of the prior work. This forces the semantics to use a $\texttt{Pick}^{\mathcal{V}}()$ event as part of a `store` operation, which in turn invalidates store-forwarding optimizations (where a `load` following a `store` to the same location is replaced by the stored value). There are similar issues with respect to the proper treatment of `poison` and `undef` with respect to intrinsics and external function calls, that remain to be resolved. VIR currently models the behaviors of `GEP` and pointer-to-integer casts via interactions with the memory model, a natural model for these constructions as their semantics depends on details about how data is laid out in the heap; however, this also means that proving correctness of program transformations that move or eliminate such operations is nontrivial. For

instance, in addition to the usual requirements about the scopes of program identifiers, one would have to prove that a call to `alloca` doesn't affect the result of `GEP` in order to move a use of the `GEP` instruction around an `alloca`. The memory model provided in Juneyoung et al. [Lee et al. 2018] addresses this issue, ensuring that `GEP` is truly pure, and should hence be part of the design of the future rework for VIR's memory model. Despite such remaining challenges, we are optimistic that the design of VIR provides the necessary ingredients to model LLVM semantics with higher fidelity—ITrees provide the ability to introduce and handle nondeterministic events at various levels of interpretation, and the use of the $\mathtt{propT}_{\mathtt{E}}$ monad provides a rich semantic space for describing the allowed behaviors.

LLVM IR's semantics is complex, but also evolving: subtle interactions between `poison` and `undef` led to a recent proposal [Lee et al. 2017] to simplify the under-defined values semantics via a `freeze` instruction, which in VIR affects where $\mathtt{Pick}^{V}()$ events occur; it was extremely straightforward to add support for `freeze` to VIR. Similarly, there is ongoing work on a "provenance" mechanism for specifying which pointer-to-integer casts are allowed, which is also subject to change. Maintaining a formal development of the size of VIR with such evolutions is a major challenge that we believe can be mitigated by the modularity of its semantics.

Executability Our use of monadic interpreters based on interaction trees allowed us to get an executable VIR semantics with very little effort, which enabled testing of the semantics early on. As mentioned above, one main contribution of the JSCert project is the proof of correspondence between the specification and a reference implementation. Similarly, both the Vellvm and CompCert projects have spent substantial efforts during their development to define interpreters and prove them equivalent to the relational semantics. Maintaining two artifacts incurs the cost of synchronizing them, which can become especially painful as a language evolves. Any change to the semantics has to be echoed in the interpreter and the proof fixed. With our approach, almost all of the semantics is shared with the interpreter, with the exception of the implementation of the non-deterministic effects of the language.

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

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

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

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1521539 and the ONR under Grant No. N00014-17-1-2930. 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 or the ONR.

REFERENCES

Andrew W. Appel. 2011. Verified Software Toolchain. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. https://doi.org/10.1145/3371075

Nick Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 14–25. https://doi.org/10.1145/964001.964003

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 87–100. https://doi.org/10.1145/2535838.2535876

Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the Concurrency Semantics of an LLVM Fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 100–110. https://doi.org/10.5555/3049832.3049844

Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–60. https://doi.org/10.1007/978-3-642-37036-6_3

Adam Chlipala. 2010. A Verified Compiler for an Impure Functional Language. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. ACM, New York, NY, USA, 93–106. https://doi.org/10.1145/1706299.1706312

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

Charles Ellison. 2012. *A formal semantics of C with applications.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Franz Franchetti, Tze-Meng Low, Thom Popovici, Richard Veras, Daniele G. Spampinato, Jeremy Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proceedings of the IEEE, special issue on "From High Level Specification to High Performance Code"* 106, 11 (2018). https://doi.org/10.1109/JPROC.2018.2873289

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 653–669. https://doi.org/10.5555/3026877.3026928

Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. 2018. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 646–661. https://doi.org/10.1145/3192366.3192381

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. ACM, New York, NY, USA, 193–206. https://doi.org/10.1145/2429069.2429093

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

Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-Pointer Casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 326–335. https://doi.org/10.1145/2737924.2738005

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight Verification of Separate Compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 178–190. https://doi.org/10.1145/2837614.2837642

Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: Verified Credible Compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 631–645. https://doi.org/10.1145/3192366.3192377

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. ACM, New York, NY, USA, 234–248. https://doi.org/10.1145/3293880.3294106

Robbert Krebbers and Freek Wiedijk. 2015. A typed C11 semantics for interactive theorem proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*. ACM, 15–27. https://doi.org/10.1145/2676724.2693571

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

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

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling High-Level Optimizations and Low-Level Code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 125 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276495

Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming Undefined Behavior in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, 633–647. https://doi.org/10.1145/3140587.3062343

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

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA. 26 pages. https://hal.inria.fr/hal-00703441

Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Information and Computation* 207, 2 (2009), 284 – 304. https://doi.org/10.1016/j.ic.2007.12.004 Special issue on Structural Operational Semantics (SOS).

Liyi Li and Elsa Gunter. 2020. K-LLVM: A Relatively Complete Semantics of LLVM IR. In *34rd European Conference on Object-Oriented Programming, ECOOP 2020, Berlin, Germany*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.7

Liyi Li and Elsa L. Gunter. 2018. *IsaK: A Complete Semantics of* $\mathbb{K}$. Technical Report. University of Illinois at Urbana-Champaign.

Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. *Proceedings of the 42th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3453483.3454030

Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 22–32. https://doi.org/10.1145/2813885.2737965

Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. 2020. The next 700 Relational Program Logics. *Proceedings of the ACM on Programming Languages* 4, POPL, Article 4 (2020), 33 pages. https://doi.org/10.1145/3371072

William Mansky, Dmitri Garbuzov, and Steve Zdancewic. 2015. An Axiomatic Specification for Sequential Memory Models. In *Computer Aided Verification - 27th International Conference, CAV 2015*. https://doi.org/10.1007/978-3-319-21668-3_24

Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.* 3, POPL, Article 67 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290380

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the de Facto Standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

David Menendez and Santosh Nagarakatte. 2017. Alive-Infer: Data-driven Precondition Inference for Peephole Optimizations in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, 49–63. https://doi.org/10.1145/3140587.3062372

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-Order Imperative Language. In *Proceedings of the 20th ACM SIGPLAN*

International Conference on Functional Programming (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178. https://doi.org/10.1145/2784731.2784764

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

Daniel Patterson and Amal Ahmed. 2019. The next 700 Compiler Correctness Theorems (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 85 (July 2019), 29 pages. https://doi.org/10.1145/3341689

Gordon D. Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. https://doi.org/10.1023/A:1023064908962

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, Zachary Tatlock, et al. 2019. QED at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages* 5, 2-3 (2019), 102–281.

Grigore Roşu and Traian Florin Şerbănuţă. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010), 397 – 434. https://doi.org/10.1016/j.jlap.2010.03.012 Membrane computing and programming.

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3 (2013), 22. https://doi.org/10.1145/2487241.2487248

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-Assembly Linking and Lightweight Modular Verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371091

Guy L. Steele, Jr. 1994. Building Interpreters by Composing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. ACM, New York, NY, USA, 472–492. https://doi.org/10.1145/174675.178068

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 275–287. https://doi.org/10.1145/2676726.2676985

Wouter Swierstra and Tim Baanen. 2019. A Predicate Transformer Semantics for Effects (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 103 (July 2019), 26 pages. https://doi.org/10.1145/3341707

The Coq Development Team. 2020. *The Coq Proof Assistant, version 8.11.0.* https://doi.org/10.5281/zenodo.3744225

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290375

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees. *Proceedings of the ACM on Programming Languages* 4, POPL (2020). https://doi.org/10.1145/3371119

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An Equational Theory for Weak Bisimulation via Generalized Parameterized Coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. https://doi.org/10.1145/3372885.3373813

Vadim Zaliva and Franz Franchetti. 2018. HELIX: A Case Study of a Formal Verification of High Performance Program Generation. In *Workshop on Functional High Performance Computing (FHPC)*. https://doi.org/10.1145/3264738.3264739

Vadim Zaliva and Matthieu Sozeau. 2019. Reification of Shallow-Embedded DSLs in Coq with Automated Verification. In *International Workshop on Coq for Programming Languages (CoqPL)*.

Vadim Zaliva, Ilia Zaichuk, and Franz Franchetti. 2020. Verified Translation Between Purely Functional and Imperative Domain Specific Languages in HELIX. In *Proceedings of the 12th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. https://doi.org/10.1007/978-3-030-63618-0_3

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2103621.2103709

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2013. Formal Verification of SSA-Based Optimizations for LLVM. In *Proc. 2013 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*. https://doi.org/10.1145/2499370.2462164