Compiling Symbolic Execution with Staging and Algebraic Effects

GUANNAN WEI, OLIVER BRAČEVAC, SHANGYIN TAN, and TIARK ROMPF, Purdue University, USA

Building effective symbolic execution engines poses challenges in multiple dimensions: an engine must *correctly* model the program semantics, provide *flexibility* in symbolic execution strategies, and execute them *efficiently*.

This paper proposes a principled approach to building *correct*, *flexible*, and *efficient* symbolic execution engines, directly rooted in the semantics of the underlying language in terms of a high-level definitional interpreter. The definitional interpreter induces algebraic effects to abstract over semantic variants of symbolic execution, e.g., collecting path conditions as a state effect and path exploration as a nondeterminism effect. Different handlers of these effects give rise to different symbolic execution strategies, making execution strategies orthogonal to the symbolic execution semantics, thus improving flexibility. Furthermore, by annotating the symbolic definitional interpreter with binding-times and specializing it to the input program via the first Futamura projection, we obtain a "symbolic compiler", generating efficient instrumented code having the symbolic execution semantics. Our work reconciles the interpretation- and instrumentation-based approaches to building symbolic execution engines in a uniform framework.

We illustrate our approach on a simple imperative language step-by-step and then scale up to a significant subset of LLVM IR. We also show effect handlers for common path selection strategies. Evaluating our prototype's performance shows speedups of 10~30x over the unstaged counterpart, and 2x over KLEE, a state-of-the-art symbolic interpreter for LLVM IR.

CCS Concepts: • **Software and its engineering** → **Interpreters**; **Compilers**; **Automated static analysis**.

Additional Key Words and Phrases: symbolic execution, definitional interpreters, multi-stage programming, algebraic effects

ACM Reference Format:
Guannan Wei, Oliver Bračevac, Shangyin Tan, and Tiark Rompf. 2020. Compiling Symbolic Execution with Staging and Algebraic Effects. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 164 (November 2020), 33 pages. https://doi.org/10.1145/3428232

1 INTRODUCTION

Symbolic execution is a popular and useful technique for software testing, bug finding, security, and verification [Baldoni et al. 2018]. The underlying idea is to simultaneously explore multiple execution paths in a given program, with some inputs being left symbolic rather than concrete. The essential job of a symbolic execution *engine* is translating an input program to logical formulae and constraints. Those logical formulae and constraints enable automatic test case generation

Authors' address: Guannan Wei, guannanwei@purdue.edu; Oliver Bračevac, bracevac@purdue.edu; Shangyin Tan, tan279@purdue.edu; Tiark Rompf, tiark@purdue.edu, Department of Computer Science, Purdue University, 610 Purdue Mall, West Lafayette, IN, 47907, USA.

2475-1421/2020/11-ART164
https://doi.org/10.1145/3428232

and verification of the program's liveness or safety properties. This process relies on automated theorem provers such as SMT/FOL solvers in modern symbolic execution engines.

However, building a correct and efficient symbolic execution engine is hard — this task is often considered "*the most difficult aspect of creating solver-aided tools*" [Torlak and Bodik 2014]. We recognize three major challenges in building symbolic execution engines: 1) *correctly* modeling the semantics of symbolic execution for the target language, 2) converting the model to an *efficient* implementation that collects and solves constraints, and 3) *flexibility* in supporting different symbolic execution strategies and heuristics. More recently, other communities have recognized the efficiency problem. For instance, in a security context, Poeplau and Francillon [2020] proposed to use the LLVM infrastructure to generate instrumented code that performs symbolic execution. Our goal is to provide a principled, semantics-driven foundation for building symbolic execution engines, including and especially those that incorporate a degree of compilation.

This paper's main contribution is a novel approach to constructing symbolic execution engines from a programming language's definitional interpreter [Reynolds 1972] while retaining a high level of abstraction, and achieving flexibility and efficiency through staging and algebraic effects. To the best of our knowledge, no previous work on symbolic definitional interpreters meets all of these requirements simultaneously.

Semantics First: Starting from Definitional Interpreters. A symbolic execution engine must simulate concrete execution of programs while collecting path constraints of symbolic variables. In other words, the target language's concrete semantics informs its symbolic semantics. This observation leads to a *semantics-first approach* to building symbolic execution engines: Starting from the definitional interpreter, systematically refine it to operate on symbolic values and collect symbolic path constraints, resulting in a *symbolic definitional interpreter*.

The semantics-first approach has been demonstrated in detail for small functional languages or imperative languages (e.g. [Darais et al. 2017; Mensing et al. 2019; Schwartz et al. 2010]). Furthermore, Rosette [Torlak and Bodik 2014] provides a domain-specific language based on Racket that enables lifting regular interpreters into symbolic ones. Widely-used practical symbolic execution engines such as KLEE [Cadar et al. 2008] and Symbolic Java PathFinder [Anand et al. 2007; Păsăreanu and Rungta 2010] also build upon a symbolic interpreter at their core. However, these are arguably less "definitional."

Considering that the definitional interpreter is an executable, "ground truth" specification of a language's semantics, the semantics-first approach gives developers a higher degree of confidence in correctness of the symbolic counterpart, compared to approaches that treat symbolic execution engines as pure engineering artifacts. We follow this approach but further extend the symbolic interpreter with staging and algebraic effects to address the efficiency and flexibility challenges.

Efficiency: From Symbolic Interpreter to Compiler. A common perception is that interpreters are slow due to the overhead of indirection and dispatch. Some symbolic execution engines use static code instrumentation to avoid the overhead, transforming the input program by inserting special statements for monitoring symbolic values and symbolic constraints. For instance, CREST [Burnim 2014] and EXE [Cadar et al. 2006] use the CIL framework [Necula et al. 2002] for transforming C programs in this way. SymCC [Poeplau and Francillon 2020] implements the instrumentation as a transformation pass in the LLVM framework. Instrumentation-based symbolic execution engines can significantly outperform interpretation-based counterparts [Kapus and Cadar 2017]. However, while instrumentation-based approaches have performance advantages, they usually cannot serve as semantic specifications and are not as easy to understand and implement as interpreters.

Our work reconciles the interpretation- with instrumentation-based approaches using the well-known first Futamura projection [Futamura 1971, 1999] as the missing conceptual link. We *specialize*

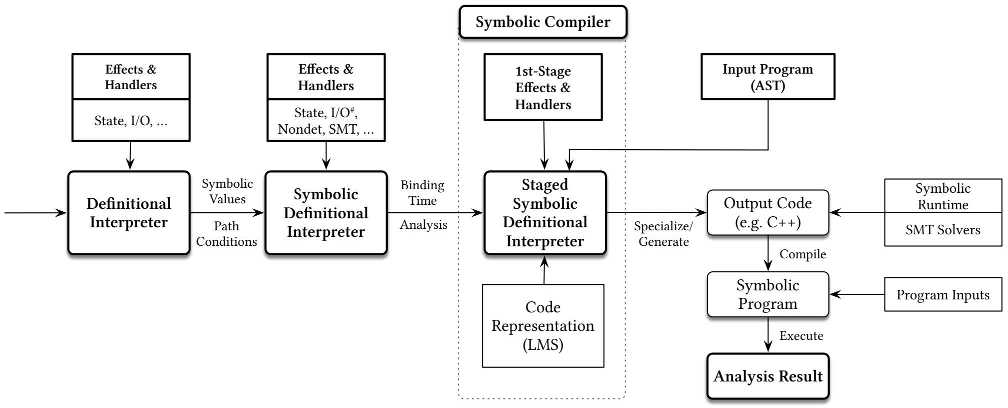

Fig. 1. The workflow of the proposed approach to symbolic execution engine construction

a symbolic interpreter to the input program using the first Futamura projection. The result is a program that executes the same symbolic computation without the interpretation overhead. We use multi-stage programming (MSP) [Taha and Sheard 1997], a form of partial evaluation [Jones et al. 1993] guided by the programmer, and the Lightweight Modular Staging framework (LMS) [Rompf and Odersky 2010] to realize the first Fumatura projection. Therefore, we obtain performant instrumented symbolic programs by staging the definitional symbolic interpreter.

Flexibility: Effects and Handlers for Custom Execution Strategies. While an exhaustive symbolic execution in King's style [King 1976] is sound and complete (with regard to the underlying theories), it is impractical for larger programs due to the path- and state-explosion problem. Many execution strategies make symbolic execution non-exhaustive, using heuristics that trade soundness for a higher likelihood of triggering certain paths in a reasonable amount of time [Baldoni et al. 2018; Burnim and Sen 2008; Chipounov et al. 2011; Godefroid et al. 2005a; Kuznetsov et al. 2012]. Nevertheless, there is no single best symbolic execution strategy that works equally well for all programs. The construction of symbolic execution engines should strive for the extensibility and customizability of execution strategies.

How can we modularly specify these execution strategies while keeping the symbolic interpreter concise and high-level? We need to write the definitional interpreter using a level of control-flow abstraction at the meta-level. A variety of such techniques exist, including explicit monads and computational effects [Moggi 1989, 1991], explicit continuation passing style (CPS), or direct-style control operators such as control/prompt [Felleisen 1988] or shift/reset [Danvy and Filinski 1990] in languages that support them [Flatt et al. 2007; Koppel et al. 2018; Rompf et al. 2009]. While any such control abstraction technique should work in principle, we found it helpful and elegant to rely on the notion of algebraic effects and handlers [Plotkin and Power 2003; Plotkin and Pretnar 2013].

We view execution strategies as particular computational effects of the execution engine, and those effects should allow for user-defined interpretations. For example, path selection strategies implement different notions of nondeterminism and control flow. The symbolic definitional interpreter declares abstract effect operations in its signature, representing customizable aspects of the symbolic execution semantics. This way, our approach exhibits flexibility and modularity by decoupling the execution strategies from the semantics of symbolic execution. Different execution strategies take the form of first-class effect handlers as composable components for specifying and implementing these strategies orthogonally to the symbolic execution. Programmers may extend and adapt the symbolic execution engine with custom effects and handlers as needed.

Overview of the Approach. Figure 1 shows the individual steps of our approach to building symbolic execution engines for imperative languages.

First, one implements a high-level definitional interpreter for a given programming language. This activity also includes modeling and interpreting the semantics' computational effects in terms of algebraic effects and handlers. Then, one refines the definitional interpreter to a symbolic definitional interpreter, including symbolic values, path constraints, and additional effects, e.g., for integrating nondeterminism and SMT solving. Finally, by annotating the binding-times in this symbolic interpreter, it becomes a *symbolic compiler* via the first Futamura projection [Futamura 1971, 1999]. Notably, we use a *heterogeneous* staging framework, which permits compilation into different target languages, thus allowing the symbolic program's execution on different platforms.

We split the symbolic execution into two stages. In the first stage, given an input program and a particular execution strategy expressed with effect handlers, the symbolic compiler generates output code in a target language (e.g., C++) for the symbolic program. This stage eliminates the interpretation overhead of both AST traversal and effect handling. Then, using an external target-language compiler, compiling and executing the symbolic program yields an analysis result. In this way, one can utilize off-the-shelf compilers to optimize symbolic programs further, even though the symbolic execution engine is initially written in a high-level managed language (e.g., Scala).

Previous work (e.g., [Torlak and Bodik 2014]) has used the terminology "symbolic compilers" for tools that translate a program to logical constraints and "symbolic virtual machines" for tools that compile selective bits to constraints and invoke solvers on the fly. From that perspective, one may consider our approach as both a *symbolic compiler-compiler* and a *symbolic VM compiler* in the sense that it translates a program to another program that produces logical constraints (and orchestrates solver invocations) when run.

Furthermore, our approach is economical. The staged symbolic interpreter is developed in a high-level language (in our case, Scala) and derived from the unstaged symbolic interpreter. Using stage polymorphism [Ofenbeck et al. 2017; Stojanov et al. 2019], the staged symbolic interpreter can serve both as an "interpreter" and as a "compiler", enabling a high degree of code reuse.

Contributions. In summary, we make the following contributions:

- Using an imperative language (Imp) as example, we illustrate our approach step-by-step using a meta-language with algebraic effects and a binding-time notion:
  1. Specify the concrete execution semantics of Imp in terms of a definitional interpreter, mapping Imp programs to computations inducing algebraic effects (Section 2).
  2. Extend the definitional interpreter to a symbolic definitional interpreter, adding additional effects specific to symbolic execution (Section 3).
  3. Finally, conduct a binding-time analysis of the symbolic execution interpreter from the previous step, and annotate its parts as either dynamic or static (Section 4).
- We describe the implementation of our approach using the Lightweight Modular Staging (LMS) framework in Scala. The algebraic effects are embedded in Scala using free monads (Section 5).
- We investigate the use of effect handlers to express different path selection strategies in symbolic execution, all in a modular and flexible fashion (Section 6).
- We extend our approach to a subset of the LLVM intermediate representation (IR), discuss challenges and solutions, and build a prototype symbolic compiler for LLVM IR (Section 7).
- We evaluate the performance of the prototype LLVM IR symbolic execution engine and compare the path traversal performance with the unstaged version and KLEE, showing that the generated code significantly outperforms its unstaged counterparts and is ~2x faster than KLEE (Section 8).

$$\begin{array}{lll} v \in \mathsf{Value} & ::= \mathbb{Z} \mid \mathbb{B} \qquad op_2 \in \{+, -, \times, =, \neq, \dots\} \qquad op_1 \in \{-, \dots\} \\ e \in \mathsf{Expr} & ::= v \mid x \mid op_1(e) \mid op_2(e, e) \mid \mathsf{read} \qquad x \in \mathsf{Var} \\ s \in \mathsf{Stmt} & ::= \mathsf{assign}(x, e) \mid \mathsf{cond}(e, s, s) \mid \mathsf{while}(e, s) \mid \mathsf{seq}(s, s) \mid \mathsf{skip} \mid \mathsf{write}(e) \end{array}$$

Fig. 2. The abstract syntax of Imp.

2 DEFINITIONAL INTERPRETERS WITH ALGEBRAIC EFFECTS

Our approach to building symbolic execution engines begins with implementing the definitional interpreter [Reynolds 1972] of a given object language in a metalanguage, i.e., a host programming language. A definitional interpreter defines a "ground-truth" executable artifact of the object language's semantics, as a practical semantics-first approach.

Since our focus lies on imperative object languages, a definitional interpreter must account for side-effecting features. We employ algebraic effects [Plotkin and Power 2003] and effect handlers [Plotkin and Pretnar 2013] as a general abstraction for modeling object language effects in the metalanguage in a functional manner.

2.1 Metalanguage

In the interest of keeping the presentation at a high level, we assume a standard call-by-value polymorphic λ-calculus as the metalanguage, with data types, pattern matching, general recursion, higher-kinded types, and ad-hoc polymorphism. We will use the latter two features only indirectly for embedding algebraic effects, handlers, and effect typing. We leave universal quantification in types implicit, and use square brackets to apply constructors and polymorphic functions to type parameters, e.g.,

$$\mathsf{map} : (A \to B) \to \mathsf{List}[A] \to \mathsf{List}[B]$$

is the standard map function on lists, where A and B are universally quantified. For multi-stage programming used in the later steps of our approach, we adopt a library-based staging approach in Scala [Rompf and Odersky 2010]. We defer the discussion of staging extensions to Section 4.

2.2 Object Language: Imp

We consider the imperative language Imp (Figure 2), which is small yet sufficient to exemplify all essential aspects of symbolic execution.

Imp has expressions e and statements s. Expressions consist of integer and boolean values, variables, binary and unary operators, and an I/O primitive read, which reads an integer from standard input. Statements contain standard syntactic constructs for assignments, conditionals, loops, sequential statements, a no-op statement skip, and an I/O statement write for writing integers to standard output. The read and write forms exemplify interactions with the program's environment.

We stipulate that the different syntax categories in Figure 2 represent data type definitions in the metalanguage. Var, Value, Expr, and Stmt denote data types, and their syntax forms represent data type constructors.

2.3 Algebraic Effects and Handlers

Algebraic effects [Plotkin and Power 2003] and effect handlers [Plotkin and Pretnar 2013] enable modular and extensible programming with user-defined effects (e.g., state, exceptions, nondeterminism, I/O). To keep the presentation at a high level, we stipulate a syntax of algebraic effects and handlers in style similar to native language implementations, e.g., Koka [Leijen 2017b] and

$$\begin{array}{ll} \textbf{effect}\ \mathsf{State}[S] := & \textbf{effect}\ \mathsf{IO} := \\ \quad \mathsf{get} : S & \quad \mathsf{readInt} : \mathsf{Int} \\ \quad \mathsf{put} : S \longrightarrow \mathsf{Unit} & \quad \mathsf{writeInt} : \mathsf{Int} \longrightarrow \mathsf{Unit} \end{array}$$

Fig. 3. Example: Effect signatures for State and IO.

$$\begin{array}{ll} \text{StateHandler} : S \longrightarrow \langle \mathsf{State}[S], E\rangle A \longrightarrow \langle E\rangle(S \times A) & \text{IOHandler} : \langle \mathsf{IO}, E\rangle A \longrightarrow \langle E\rangle A \\ \text{StateHandler} = \textbf{handler}(s) & \text{IOHandler} = \textbf{handler} \\ \quad \textbf{return}(x) \mapsto \textbf{return}\ (s, x) & \quad \textbf{return}(x) \mapsto \textbf{return}\ x \\ \quad \mathsf{get}(k) \mapsto \textbf{return}\ k(s)(s) & \quad \mathsf{readInt}(k) \mapsto \textbf{return}\ k(\mathsf{StdIn.readInt}()) \\ \quad \mathsf{put}(s', k) \mapsto \textbf{return}\ k(())(s') & \quad \mathsf{writeInt}(n, k) \mapsto \{\ \_ \longleftarrow \mathsf{StdOut.print}(n); \\ & \qquad \textbf{return}\ k(())\ \} \end{array}$$

Fig. 4. Example: Effect handlers for State and IO.

Eff [Pretnar 2015]. In principle, these can be macro-translated into our metalanguage on top of a free(r) monad data type, higher-kinded types and ad-hoc polymorphism.

Effect signatures and effect types define an interface of abstract effect operations. For specifying the concrete definitional interpreter for Imp, we define corresponding effect signatures of the State and IO effects in Figure 3. Named interfaces group effect operations together (as is custom, we call all of these entities "effects"). The State effect is parametric in the type of underlying state values S and has two abstract effect operations get and put for retrieving and updating (respectively) the current state. Similarly, the IO effect defines operations for reading and writing integer values. At this point, the meaning of these operations is underspecified.

We annotate *effect rows*, i.e., type-level lists of effect interfaces, to the type of effect operations. For instance, effect operations, such as put : $S \longrightarrow \langle \mathsf{State}[S], E\rangle\mathsf{Unit}$ (Figure 3), represent metalanguage functions where its effect row $\langle \mathsf{State}[S], E\rangle$ states that invocations induce the $\mathsf{State}[S]$ effect. Furthermore, it is *effect polymorphic*, indicated by the type parameter E. Effect polymorphism ensures the composability of computations in usage contexts requiring potentially more than the $\mathsf{State}[S]$ effect. We stipulate that E always ranges over effect rows. Intuitively, $\langle E\rangle T$ is a monad of computations returning type T indexed by effect rows E. Pure computations have empty effect rows $\langle\rangle$. Furthermore, we do not distinguish between $\langle\rangle B$ and B.

We adopt the well-known comprehension syntax $\{\ x \leftarrow c; \ldots\ \}$ for computations c with algebraic effects, analogous to Haskell's **do**-notation or Scala's **for**-comprehensions. Within this syntactic scope, we write **return** v for a computation that directly yields a value v. For instance:

$$\begin{array}{l} \mathsf{c} : \mathsf{Int} \longrightarrow \langle \mathsf{State}[\mathsf{Int}], \mathsf{IO}, E\rangle\mathsf{Unit} \\ \mathsf{c} = \lambda s.\{x \leftarrow \mathsf{put}\ s;\ s_1 \leftarrow \mathsf{get};\ s_2 \leftarrow \mathsf{get};\ y \leftarrow \mathsf{writeInt}\ (s_1 + s_2);\ \textbf{return}\ y\}. \end{array}$$

For brevity, we may also use **return** c as syntactic sugar for $\{\ v \leftarrow c;\ \textbf{return}\ v\ \}$ whenever possible, and omit bindings of unused values in sequences, e.g.,

$$\mathsf{c} = \lambda s.\{\mathsf{put}\ s;\ s_1 \leftarrow \mathsf{get};\ s_2 \leftarrow \mathsf{get};\ \textbf{return}\ \mathsf{writeInt}\ (s_1 + s_2)\}.$$

To assign meanings to effect operations, we need to define *effect handlers*, which are first-class, composable values that define a user-defined interpretation of the effect operations. The decoupling of effect signatures and interpretations has advantages concerning modular abstraction and instantiation over the more conventional monad transformers approach (cf. [Kammar et al. 2013]). Various means for implementing effects and handlers exist, such as delimited control/CPS

$$\begin{aligned}
\mathrm{eval} : \mathrm{Expr} &\longrightarrow \langle \mathsf{State}[\mathsf{Store}], \mathsf{IO}\rangle \mathsf{Value} \\
\mathrm{eval}(v) &= \textbf{return } v \\
\mathrm{eval}(x) &= \{\ \sigma \leftarrow \mathsf{get}; \textbf{return } \sigma(x)\ \} \\
\mathrm{eval}(op(e)) &= \{\ v \leftarrow \mathrm{eval}(e);\ \textbf{return } \mathrm{primEval}_1(op, v)\ \} \\
\mathrm{eval}(op(e_1, e_2)) &= \{\ v_1 \leftarrow \mathrm{eval}(e_1);\ v_2 \leftarrow \mathrm{eval}(e_2);\ \textbf{return } \mathrm{primEval}_2(op, v_1, v_2)\ \} \\
\mathrm{eval}(\mathrm{read}) &= \textbf{return } \mathsf{readInt}
\end{aligned}$$

Fig. 5. Definitional interpreter for expressions. The definitions of $\mathrm{primEval}_{\{1,2\}}$ are elided.

transformation [Hillerström et al. 2017; Kammar et al. 2013] or free(r) monads [Kiselyov and Ishii 2015; Swierstra 2008; Wu et al. 2014]. We implement these in Scala using the latter approach (cf. Section 5). We discuss concrete handlers while introducing the definitional interpreter of Imp in the rest of this section.

2.4 A Definitional Interpreter for Imp

We define definitional interpreters in terms of pure functions from abstract syntax on expressions and statements into computations with effects. The definitional interpreters for expressions (Figure 5) and statements (Figure 6) have the following type signatures:

$$\begin{aligned}
\mathrm{eval} : \mathrm{Expr} &\longrightarrow \langle \mathsf{State}[\mathsf{Store}], \mathsf{IO}\rangle \mathsf{Value} \\
\mathrm{exec} : \mathrm{Stmt} &\longrightarrow \langle \mathsf{State}[\mathsf{Store}], \mathsf{IO}\rangle \mathsf{Unit}.
\end{aligned}$$

We represent all the effects of the object language in terms of effect interfaces appearing in the effect row of the definitional interpreter(s). In our case, Imp's mutation and I/O primitives (Figure 2) translate to the State effect carrying a Store type and the IO effect, respectively (Figure 3).

The Store type represents the memory and corresponds to partial function of type Var $\longrightarrow$ Value. If σ is a Store, we write $\sigma[x \mapsto v]$ to denote the store σ' such that $\sigma'(z) = v$ if $z = x$ and $\sigma'(z) = \sigma(z)$ otherwise.

We also assume two primitive evaluators for arithmetic and boolean operations:

$$\begin{aligned}
\mathrm{primEval}_1 &: \mathsf{Op}_1 \times \mathsf{Value} \longrightarrow \mathsf{Value} \\
\mathrm{primEval}_2 &: \mathsf{Op}_2 \times \mathsf{Value} \times \mathsf{Value} \longrightarrow \mathsf{Value}.
\end{aligned}$$

With the given effect operations and primitive evaluators, Figures 5 and 6 implement the standard store-transformer semantics of Imp [Nielson and Nielson 2007]. When executing a conditional statement cond, we use the conditional clause **if**/**else** from the metalanguage for branching. We execute loops by unrolling the syntactic constructs while(e, s), one step each time.

2.5 Handlers for Concrete Execution

For completing the object language's definitional interpreter, one must define concrete interpretations of the abstract effects in the interpreter's effect row via effect handlers. Figure 4 shows the handlers for Imp's State and IO effects. Intuitively, effect handlers (or simply handlers) are control-flow abstractions behaving similarly to exception handlers, but they may resume the underlying computation that "threw" an effect operation. Formally, handlers are computation transformers/homomorphisms of type

$$\langle \overline{X}, E\rangle A \longrightarrow \langle \overline{Y}, E\rangle B$$

$$
\begin{aligned}
\mathsf{exec} &: \mathsf{Stmt} \to \langle \mathsf{State}[\mathsf{Store}], \mathsf{IO}\rangle \mathsf{Unit} \\
\mathsf{exec}(\mathsf{assign}(x, e)) &= \{\ v \leftarrow \mathsf{eval}(e);\quad \sigma \leftarrow \mathsf{get};\ \textbf{return}\ \mathsf{put}(\sigma[x \mapsto v])\ \} \\
\mathsf{exec}(\mathsf{cond}(e, s_1, s_2)) &= \{\ b \leftarrow \mathsf{eval}(e);\quad \textbf{return if}\ (b)\ \mathsf{exec}(s_1)\ \textbf{else}\ \mathsf{exec}(s_2)\ \} \\
\mathsf{exec}(\mathsf{while}(e, s)) &= \{\ b \leftarrow \mathsf{eval}(e);\quad \textbf{return if}\ (b)\ \mathsf{exec}(\mathsf{seq}(s, \mathsf{while}(e, s)))\ \textbf{else}\ \mathsf{exec}(\mathsf{skip})\ \} \\
\mathsf{exec}(\mathsf{write}(e)) &= \{\ v \leftarrow \mathsf{eval}(e);\quad \textbf{return}\ \mathsf{writeInt}(v)\ \} \\
\mathsf{exec}(\mathsf{seq}(s_1, s_2)) &= \{\ \_ \leftarrow \mathsf{exec}(s_1);\ \textbf{return}\ \mathsf{exec}(s_2)\ \} \\
\mathsf{exec}(\mathsf{skip}) &= \textbf{return}\ ()
\end{aligned}
$$

Fig. 6. Definitional Interpreter for statements.

that resolve a finite prefix $\overline{X}$ in an effect row and possibly induce effects from the list $\overline{Y}$ while doing so. They may also transform the computation's return type A into type B. For example, the handlers for Imp's effects (Figure 4) have types

$$
\begin{aligned}
\mathrm{StateHandler} &: S \to \langle \mathsf{State}[S], E\rangle A \to \langle E\rangle (S \times A) \\
\mathrm{IOHandler} &: \langle \mathsf{IO}, E\rangle A \to \langle E\rangle A,
\end{aligned}
$$

that interpret the State and IO effect (respectively) to another effect computation, explained further below. Handlers have clauses for handling the specified effect operation, which bind the operation's parameters as specified by the operation's interface and the continuation of the computation, usually designated by k. The special clause return defines the behavior when the handled computation returns with a result. We often omit this clause in the case of identity functions.

Functional State Handler. Given an initial state s of type S, the StateHandler (Figure 4, left) handles the State effect (Figure 3) by transforming the handled computation into a tuple of type $S \times A$, returning the answer A of the computation along with the final state at the end of the computation. For example, we have that

$$(\mathrm{StateHandler}(21)\ \{s_1 \leftarrow \mathsf{get};\ s_2 \leftarrow \mathsf{get};\ \mathsf{put}\ 0;\ \textbf{return}\ (s_1 + s_2)\}) \equiv \{\textbf{return}\ (0, 42)\}.$$

StateHandler is a parameterized/indexed handler in Koka's style [Leijen 2017b]. It binds the current state value s as a local parameter. The continuations in the clauses additionally take an updated state value used for handling subsequent state operations. In contrast, the IOHandler (Figure 4) does not have a parameter.

Handling I/O with Real Side Effects. The StateHandler above defines a user-defined functional simulation of a state, but not an actual mutation side effect. Indeed, we may handle algebraic effects with "real" side effects if present in the metalanguage. Assuming we have access to actual I/O primitives, we define the IOHandler (Figure 4) to resolve the abstract IO effect in the definitional interpreter by delegating the effect operations readInt and writeInt to these primitives.

3 SYMBOLIC DEFINITIONAL INTERPRETERS WITH ALGEBRAIC EFFECTS

Given an object language's definitional interpreter with algebraic effects (Section 2), the next step is refining the interpreter to a *symbolic definitional interpreter*, which specifies a generic symbolic execution semantics for the object language. The refinement introduces additional effects to cater to symbolic execution. The recipe for refinement is as follows:

- Extend the Value domain with symbolic values.
- Lift primitive evaluators so that they can operate on symbolic values.

$$
\begin{array}{lll}
\textbf{effect}\ \mathsf{Nondet} := & \textbf{effect}\ \mathsf{SMT} := & \textbf{effect}\ \mathsf{IO}^{\#} := \\
\quad \mathsf{choice}[X] : \mathsf{List}[X] \to X & \quad \mathsf{sat?} : \mathsf{PC} \to \mathbb{B} & \quad \mathsf{readInt} : \mathsf{Value} \\
& \quad \mathsf{concretize} : \mathsf{PC} \times \mathsf{Var} & \quad \mathsf{writeInt} : \mathsf{Value} \to \mathsf{Unit} \\
& \qquad \to \mathsf{Option}[\mathsf{Value}] &
\end{array}
$$

Fig. 7. Effects during symbolic execution.

- Introduce a nondeterminism effect to abstract over different exploration strategies for branching constructs.
- Maintain a path condition (i.e., set of Boolean expressions expressing the path constraint) using another state effect.
- Introduce an SMT effect to interact with external solvers, if required.

This section continues the Imp example, refining its definitional interpreter (Figures 5 and 6) to a symbolic version that implements the semantics from [King 1976].

3.1 Symbolic Values and Path Conditions

The first step is to extend the value domain with *symbolic values*:

$$v \in \mathsf{Value} ::= \mathbb{Z} \mid \mathbb{B} \mid \mathsf{sym}(e).$$

In contrast to the value domain defined in Figure 2, we additionally have $\mathsf{sym}(e)$, which holds an expression e with symbolic values/unknowns. The sym form represents unknown inputs of the target program, as well as symbolic computation on symbolic inputs. For instance, $\mathsf{sym}(x)$ is a symbolic value for a fresh variable $x \in \mathsf{Var}$. A value $v \in \mathsf{Value}$ is *symbolic* if it is of the form $\mathsf{sym}(e)$ and *concrete* otherwise. Analogously, an expression e is symbolic if it contains a subexpression $\mathsf{sym}(e')$.

We represent *path conditions* by sets of (Boolean) expressions

$$\pi \in \mathsf{PC} = \mathcal{P}(\mathsf{Expr}),$$

where each set π represents the logical conjunction of its elements, expressing the path constraint posed on the symbolic inputs during the symbolic execution. Path conditions are initially empty when starting the execution, thus vacuously true. Whenever the symbolic definitional interpreter evaluates a conditional statement $\mathsf{cond}(e, s_1, s_2)$ where e involves symbolic values, it may separately execute s_1 under a new path condition containing e and execute s_2 under a new path condition containing $\neg e$, i.e., path conditions are path-sensitive. The branch choice depends on the concrete path exploration strategy.

3.2 Effects for Symbolic Execution

Symbolic execution requires slight modifications to the effect interfaces (Section 2.4), which introduce additional effects.

State Effect for Path Conditions. A path condition is an additional piece of state in symbolic execution next to the Store (Section 2.4). The state for symbolic execution now consists of a product

$$\mathbb{S} = \mathsf{Store} \times \mathsf{PC},$$

and the symbolic definitional interpreter induces a $\mathsf{State}[\mathbb{S}]$ effect.

Branching as Nondeterminism Effect. If the condition at a branching point is a symbolic value, then the symbolic definitional interpreter may explore multiple paths; the Nondet effect interface (Figure 7) abstracts over concrete strategies for choosing such paths. The polymorphic operation choice offers a finite number of candidates to choose from to the handling context. The caller of this operation determines the concrete type X of choices.[1]

SMT Solver Interactions and Symbolic I/O Effects. Some symbolic execution scenarios invoke a decision procedure to discharge path conditions/logical formulae. For example, we may need to query an SMT solver for the current path conditions' satisfiability at a branching point. Depending on the answer, we may either disregard a path (unsatisfiable) or commit to a satisfiable path.

Another use of SMT solvers is *concretization* of symbolic expressions, i.e., generating a concrete value satisfying a given path condition. E.g, concretization is necessary for calling external functions that cannot take symbolic arguments. Finally, the eventual goal of symbolic execution is generating test cases, which are also obtained by concretization. We cover all of these uses cases with the SMT effect interface (Figure 7).

The operation sat? takes a path condition and produces a Boolean value, indicating the condition's satisfiability. The operation concretize accepts the path condition argument and the variable for concretization. It returns an optional Value, which is either Some(v) for a concrete value v satisfying the path condition, or None if no such value exists. Usually datatypes in the object language can be encoded as finite-sized bit-vectors, SMT arrays, or other theories supported by the solver. We elide the details of encoding these here and take a high-level view of SMT solver interactions.

We also change the previous IO effect to produce or yield symbolic values (i.e., sym(x) where x is fresh and unconstrained): the effect's operations now operate on the Value type. We denote the new effect signature for symbolic input/output operations by $\mathsf{IO}^\#$ (Figure 7).

3.3 Symbolic Definitional Interpreter

The type signatures of the symbolic definitional interpreters for expressions and statements only differ in their effect row from their non-symbolic counterparts in Figures 5 and 6:

$$\mathsf{eval} : \mathsf{Expr} \rightarrow \langle \mathsf{State}[\mathbb{S}], \mathsf{IO}^\# \rangle \mathsf{Value}.$$

$$\mathsf{exec} : \mathsf{Stmt} \rightarrow \langle \mathsf{State}[\mathbb{S}], \mathsf{IO}^\#, \mathsf{SMT}, \mathsf{Nondet} \rangle \mathsf{Unit}.$$

As motivated above, the state effect now carries a store and path conditions, and we use the refined $\mathsf{IO}^\#$ effect for I/O with symbolic values. Furthermore, the interpreter for statements induces the Nondet and SMT effects, abstracting over branch exploration, and SMT solver interactions.

The definitions of the interpreter functions remain, for the most part, unchanged from Figures 5 and 6. In the following, we elaborate on the differences:

3.3.1 *Interpretation of Primitives.*

Because symbols and expressions are now part of the values domain, we must lift the primitive evaluators primEval$_1$ and primEval$_2$ accordingly. For example, we would need to lift integer addition (and similarly, other cases) as follows:

$$\begin{array}{rcll} \mathrm{primEval}_2^\#(+, n_1, n_2) & = & \mathrm{primEval}_2(+, n_1, n_2) & \text{if } n_1, n_2 \in \mathbb{Z} \\ \mathrm{primEval}_2^\#(+, e_1, e_2) & = & \mathsf{sym}(+(e_1, e_2)) & \text{otherwise.} \end{array}$$

3.3.2 *Interpretation of Branches and Loops.*

A significant departure from the previous definition of exec (Figure 6) is the evaluation of conditional statements and loops: we must record path

[1]For a handler, the type X is an abstract/existential type. Supplying the empty list of choices is equivalent to a failure/exception, because the handler cannot resume the computation if the list of available Xs is empty.

conditions and explore branches according to some strategy, e.g., nondeterministically. This change affects the cases for cond and while in the definition of exec.

In what follows, we rely on helper functions that act on the individual components of the Store and path condition states:

$$\begin{aligned}
\text{getPathCond} &: \langle \mathsf{State}[\mathbb{S}] \rangle \mathsf{PC} \\
\text{getPathCond} &= \{\ (\sigma, \pi) \leftarrow \mathsf{get};\ \textbf{return}\ \pi\ \} \\
\text{putPathCond} &: \mathsf{Expr} \rightarrow \langle \mathsf{State}[\mathbb{S}] \rangle \mathsf{Unit} \\
\text{putPathCond}(e) &= \{\ (\sigma, \pi) \leftarrow \mathsf{get};\ \textbf{return}\ \mathsf{put}((\sigma, \{e[\sigma]\} \cup \pi))\ \} \\
\text{getStore} &: \langle \mathsf{State}[\mathbb{S}] \rangle \mathsf{Store} \\
\text{getStore} &= \{\ (\sigma, \pi) \leftarrow \mathsf{get};\ \textbf{return}\ \sigma\ \} \\
\text{putStore} &: \mathsf{Var} \times \mathsf{Value} \rightarrow \langle \mathsf{State}[\mathbb{S}] \rangle \mathsf{Unit} \\
\text{putStore}(x, v) &= \{\ (\sigma, \pi) \leftarrow \mathsf{get};\ \textbf{return}\ \mathsf{put}((\sigma[x \mapsto v], \pi))\ \}
\end{aligned}$$

The helper function putPathCond will add expression e to path condition, where e is substituted with concrete values in the current store, if there is any, denoted by $e[\sigma]$ in its definition.

Furthermore, we define a helper function exec′ that updates path conditions and then executes the statement, shown below:

$$\begin{aligned}
\text{exec}' &: \mathsf{Stmt} \times \mathsf{Expr} \rightarrow \langle \mathsf{State}[\mathbb{S}], \mathsf{IO}^{\#}, \mathsf{SMT}, \mathsf{Nondet} \rangle \mathsf{Unit} \\
\text{exec}'(s, e) &= \{\ \_ \leftarrow \text{putPathCond}(e);\ \textbf{return}\ \text{exec}(s)\ \}.
\end{aligned}$$

Depending on the symbolic semantics we would like to follow, there are different ways to handle conditionals and loops. The naive one is to nondeterministically execute both branches, using the Nondet effect (Figure 7), no matter the value of the condition:

$$\text{exec}(\mathsf{cond}(e, s_1, s_2)) = \{\ \_ \leftarrow \text{eval}(e);\ (s', e') \leftarrow \mathsf{choice}((s_1, e), (s_2, \neg e));\ \text{exec}(s', e')\ \}.$$

However, naively executing both branches is neither economical nor correct, as spurious execution paths would lead to false errors. The refined version would mix concrete evaluation and symbolic execution (not to be confused with concolic execution[2] [Godefroid et al. 2005b]), by only nondeterministically executing both branches when the value of condition e is indeterminate. That is, in the case of symbolic conditions, we invoke the choice effect operation and execute s_1 and s_2 nondeterministically:

$$\begin{aligned}
\text{exec}(\mathsf{cond}(e, s_1, s_2)) = \{\ b \leftarrow \text{eval}(e);& \\
&\textbf{return if}\ (b \equiv \text{true})\ \text{exec}'(s_1, e) \\
&\qquad \textbf{else if}\ (b \equiv \text{false})\ \text{exec}'(s_2, \neg e) \\
&\qquad \textbf{else}\ \{\ (s', e') \leftarrow \mathsf{choice}((s_1, e), (s_2, \neg e));\ \text{exec}'(s', e')\ \}\ \}
\end{aligned}$$

Using the same syntactic unrolling idea in concrete execution, we define loop execution by desugaring to cond and while. However, online unrolling of loops during the symbolic execution may result in the nontermination of the engine. While the user may intervene and halt this at any point, we choose a more straightforward way that statically unfolds the loop a constant number of times:

$$\text{exec}(\mathsf{while}(e, s)) = \text{exec}(\text{unfold}(k, \mathsf{while}(e, s)))$$

[2]Concolic execution uses concrete inputs to guide symbolic execution and usually does not have nondeterministic exploration.

$$
\begin{aligned}
&\text{NondetHandler} : \langle \mathsf{Nondet}, E\rangle A \longrightarrow \langle E\rangle \mathsf{List}[A] \\
&\text{NondetHandler} = \textbf{handler} \\
&\qquad \textbf{return}(x) \mapsto \textbf{return}\ \mathsf{List}(x) \\
&\qquad \mathsf{choice}(xs, k) \mapsto \textbf{return}\ \text{fold}(xs, \{\textbf{return}\ \mathsf{List}()\}, f) \\
&\qquad\quad \text{where } f : \langle E\rangle \mathsf{List}[T] \times T \longrightarrow \langle E\rangle \mathsf{List}[T] \\
&\qquad\qquad f(acc, x) = \{\ ys \leftarrow acc;\ \ xs \leftarrow k(x);\ \ \textbf{return}\ ys \mathbin{++} xs\ \}
\end{aligned}
$$

Fig. 8. Effect handler for Nondet.

$$
\begin{aligned}
&\text{SMTHandler} : \langle \mathsf{SMT}, E\rangle A \longrightarrow \langle E\rangle A \\
&\text{SMTHandler} = \textbf{handler} \\
&\qquad \mathsf{sat?}(\pi, k) \mapsto \{\ b \leftarrow \text{oracle}(\pi);\quad \textbf{return}\ k(b)\ \} \\
&\qquad \mathsf{concretize}(\pi, x, k) \mapsto \{\ v \leftarrow \text{oracle}(\pi, x);\ \textbf{return}\ k(v)\ \}
\end{aligned}
$$

Fig. 9. Effect handler for SMT.

where unfold is defined as

$$
\begin{aligned}
&\mathsf{unfold}(0, \mathsf{while}(e, s)) = \mathsf{skip}; \\
&\mathsf{unfold}(k, \mathsf{while}(e, s)) = \mathsf{cond}(e, \mathsf{seq}(s, \mathsf{unfold}(k-1, \mathsf{while}(e, s))), \mathsf{skip}).
\end{aligned}
$$

However, syntactic unrolling may lead to undesirable exponential code duplication. In Section 4.4, we address generating dynamic loops in the symbolic compiler.

3.4 Handlers for Symbolic Execution

To put all the pieces together, we must still define the interpretation of the Nondet and SMT effects, and redefine a symbolic interpretation for the $\mathsf{IO}^{\#}$ effect (Figure 7). The handler of State effect remains the same as in concrete execution (Figure 4).

3.4.1 Handling Nondeterminism. We handle the nondeterminism effect in the following fashion (Figure 8): In the return case, we lift the value x into a singleton list, indicating that this is the only "choice". In the choice case, we resume the continuation k of the computation requesting the choice with all the supplied candidates, and concatenate all the resulting lists into a single list (by folding over the candidate list xs from left to right).

While the handler in Figure 8 exhaustively explores every candidate, one may implement other strategies, which we will discuss in Section 6.

3.4.2 Handling SMT Operations. The most convenient way to handle the *SMT* effect is to invoke an external SMT solver, such as Z3 or CVC4. This requires first to encode the path condition into a format (such as smt-lib) that can be recognized by the chosen solver, and then invoke the solver with the encoded formulae. We elide this part's details, but assume an oracle will produce the desired answer for satisfiability and concretizing symbolic values, as sketched in Figure 9.

3.4.3 Handling IO, Symbolically. The remaining piece of symbolic execution is to make the IO effect symbolic. readInt can now return a symbolic value $\mathsf{sym}(x)$, where x is a fresh and unconstrained variable. We have more choices of handling writeInt: (1) if the argument value is symbolic, we may concretize it to a concrete value and output that to standard output, or (2) instead of outputting values to the system's standard output, the analysis could collect all symbolic outputs and their associated constraints during the execution in the fashion of a writer monad. Here we favor simplicity and show the handler of the first approach in Figure 10.

$$\begin{aligned}
&\mathrm{IO}^{\#}\mathrm{Handler} : \langle \mathsf{IO}^{\#}, E\rangle A \longrightarrow \langle E\rangle A \\
&\mathrm{IO}^{\#}\mathrm{Handler} = \textbf{handler} \\
&\qquad \mathsf{readInt}(k) \mapsto \{\ \textbf{return}\ k(\mathsf{sym}(x))\ \} \quad \text{where } x \text{ is fresh} \\
&\quad \mathsf{writeInt}(e, k) \mapsto \{\ n \leftarrow \mathsf{concretize}(e);\ \_ \leftarrow \mathsf{StdOut.print}(n);\ \textbf{return}\ k(())\ \}
\end{aligned}$$

Fig. 10. Effect handler for IO#.

3.5 Assembling Symbolic Definitional Interpreters

Composing handlers of State, Nondet, SMT and IO# effects with the refined eval and exec functions (cf. Section 3.3.2) yields an artifact for symbolically executing object language programs. The order in which we compose the handlers produces different types of results. Due to the fact that in symbolic execution the states are path-sensitive, we must interpret the state effect as a “local” state, in the sense that each path has its own state, instead of sharing a single “global” state. Hence, we should apply the StateHandler handler first, and Nondet handler afterwards:

$$\begin{aligned}
&\mathrm{SymExeHandler} : \langle \mathsf{State}[\mathbb{S}], \mathsf{SMT}, \mathsf{IO}^{\#}, \mathsf{Nondet}\rangle \mathsf{Unit} \longrightarrow \mathsf{List}[\mathbb{S} \times \mathsf{Unit}] \\
&\mathrm{SymExeHandler} = \mathrm{NondetHandler} \circ \mathrm{SMTHandler} \circ \mathrm{IO}^{\#}\mathrm{Handler} \circ \mathrm{StateHandler}(s_0)
\end{aligned}$$

where s_0 is an initial value of the symbolic state.

With this combination of handlers, the eventual result value after effect-handling is a list, where each element contains the symbolic state $\mathbb{S}$ of an execution path. Given an Imp program p, we shall further compose SymExeHandler and exec to obtain the result of symbolic execution:

$$(\mathrm{SymExeHandler} \circ \mathrm{exec})(p) : \mathsf{List}[\mathbb{S} \times \mathsf{Unit}]$$

We could also generate inputs for the explored paths by invoking SMT operations on the obtained set of path conditions after applying exec but before effect handling.

Given a program of finite paths, the symbolic execution as presented here is both sound and complete, in the following sense [Baldoni et al. 2018]:

Definition 3.1 (Soundness). A symbolic execution is sound if it can find all unsafe inputs.

Definition 3.2 (Completeness). A symbolic execution is complete if its reported unsafe inputs are indeed unsafe.

Proof sketch. Soundness and completeness stem from the fact that we explore every path, under the assumption that the underlying decision procedure is also sound and complete. □

Soundness implies that the symbolic execution explores all feasible paths, and, as demonstrated later in Section 6, practical symbolic executions often have to drop this property by using different heuristics and execution strategies.

4 STAGED SYMBOLIC EXECUTION: INTERPRETER AND COMPILER

Having derived a symbolic definitional interpreter (Section 3) the next step is eliminating the overhead of interpretation and effect handling for efficient symbolic execution. We apply the first Futamura projection [Futamura 1971, 1999] using heterogeneous multi-stage programming. This yields a staged symbolic interpreter, i.e., a symbolic compiler. The required steps are as follows:

- Conduct a manual binding-time analysis and add stage annotations to the symbolic interpreter. I.e., which parts of the interpreter’s computation happen *statically* (i.e., at compile-time, first stage) versus *dynamically* (i.e., at run time, second stage).

- Handle the interpreter's effects strictly at compile-time, generating code which is free of handlers. This step requires refining the effect handlers to transform computations into second-stage code representations instead of first-stage values.

4.1 A Two-Stage Binding-Time Notation

We borrow the binding-time notation from two-level λ-calculus [Nielson and Nielson 1985, 1992] for our metalanguage, where an underlined term $\underline{e}$ or type $\underline{T}$ indicates a second-stage (dynamic) computation or type, and an undecorated term or type indicates a first-stage (static) computation or type. We interpret the underlying evaluation of the two-stage binding-time notation in the way of LMS [Rompf 2016], which preserves the call-by-value evaluation order of our metalanguage in the second stage.

Example 4.1 (Functions). $\underline{\mathsf{Int} \rightarrow \mathsf{Int}}$ is a second-stage function type, and $\underline{\lambda x.x + 1}$ is a term of this type. However, $\underline{\mathsf{Int}} \rightarrow \underline{\mathsf{Int}}$ is a first-stage function type whose domain and codomain are second-stage integers. An example term of that type is $\lambda \underline{x}.\underline{x + 1}$, where λ is not underlined.

Example 4.2 (Applications). If $\underline{f}$ is second-stage function and $\underline{x}$ a second-stage value, then $\underline{f(x)}$ is a second-stage application; whereas $(\lambda \underline{x}.\underline{x + 1})(\underline{y})$ is a first-stage application, which reduces to $\underline{y + 1}$ at the first stage.

Example 4.3 (Lifting). A first-stage available term can be lifted into the second stage. For example, from $v : \mathsf{Int}$, to $\underline{v} : \underline{\mathsf{Int}}$.

Well-formedness. The two-stage binding-time notation also poses a *well-formedness* requirement on types and terms. Base types at either stage are well-formed. However, if a function type is second stage, zsh:1: command not found: Make is ill-formed. This rule applies to other type constructors such as product or list types, e.g., $\underline{\mathsf{List}}[\mathsf{Int}]$ is ill-formed.

Similarly, second-stage conditionals $\underline{\textbf{if}}$ require second-stage conditions and branches:

$$\frac{\underline{e_1} : \underline{\mathbb{B}} \qquad \underline{e_2} : \underline{T} \qquad \underline{e_3} : \underline{T}}{\underline{\textbf{if}\ (e_1)\ e_2\ \textbf{else}\ e_3} : \underline{T}} \qquad (\text{WF-}\underline{\text{IF}})$$

In the case of first-stage conditionals, the condition must be first-stage too, but may have second-stage branches, e.g., $\textbf{if}\ (\mathsf{true})\ \underline{e_2}\ \textbf{else}\ \underline{e_3}$ is well-formed.

First-Stage Effect Computation and Handlers Only. We prohibit algebraic effect computations and handlers in the second stage for performance reasons. Hence, underlined computation types $\underline{\langle E\rangle A}$ are ill-formed for any non-empty or variable effect row E, whereas $\underline{\langle\rangle A} \equiv \langle\rangle\underline{A} \equiv \underline{A}$ (effect-free computations) and $\langle E\rangle\underline{A}$ (first-stage computations producing second-stage values) are permitted. We also rule out second-stage handlers $\underline{\langle E_1\rangle A \rightarrow \langle E_2\rangle B}$, but permit $\langle E_1\rangle\underline{A} \rightarrow \langle E_2\rangle\underline{B}$ (first-stage handlers transforming second-stage values). Similarly, we forbid underlining effect operations so that no algebraic effect leaks into the second stage, e.g., $\underline{\mathsf{put}\ 5}$ is ill-formed, whereas $\mathsf{put}\ \underline{5}$ is well-formed.

4.2 Two-Stage Symbolic Definitional Interpreters

Starting from the symbolic definitional interpreters (Section 3), we first add the required binding-time annotations, which is straightforward (Figures 11 and 12). Their type signatures become:

$$\mathsf{eval} : \mathsf{Expr} \rightarrow \langle \mathsf{State}[\underline{\mathbb{S}}], \mathsf{IO}^{\#}\rangle \underline{\mathsf{Value}}$$

$$\mathsf{exec} : \mathsf{Stmt} \rightarrow \langle \mathsf{State}[\underline{\mathbb{S}}], \mathsf{IO}^{\#}, \mathsf{SMT}, \mathsf{Nondet}\rangle \underline{\mathsf{Unit}}.$$

$$
\begin{aligned}
\text{eval} &: \text{Expr} \longrightarrow \langle \text{State}[\underline{\mathbb{S}}], \text{IO}^{\#} \rangle \underline{\text{Value}} \\
\text{eval}(v) &= \textbf{return } \underline{v} \\
\text{eval}(x) &= \{\ \underline{\sigma} \leftarrow \text{getStore};\ \textbf{return } \underline{\sigma(x)}\ \} \\
\text{eval}(op(e)) &= \{\ \underline{v} \leftarrow \text{eval}(e);\ \textbf{return } \text{primEval}^{\#}_1(op, \underline{v})\ \} \\
\text{eval}(op(e_1, e_2)) &= \{\ \underline{v_1} \leftarrow \text{eval}(e_1);\ \underline{v_2} \leftarrow \text{eval}(e_2);\ \textbf{return } \text{primEval}^{\#}_2(op, \underline{v_1}, \underline{v_2})\ \} \\
\text{eval}(\text{read}) &= \textbf{return } \text{readInt}
\end{aligned}
$$

$$
\begin{aligned}
\text{primEval}^{\#}_1 &: \text{Op}_1 \times \underline{\text{Value}} \longrightarrow \underline{\text{Value}} \\
\text{primEval}^{\#}_2 &: \text{Op}_2 \times \underline{\text{Value}} \times \underline{\text{Value}} \longrightarrow \underline{\text{Value}}
\end{aligned}
$$

Fig. 11. The 2-stage symbolic definitional interpreter for expressions. Definitions of $\text{primEval}^{\#}_{\{1,2\}}$ elided.

$$
\begin{aligned}
\text{exec} &: \text{Stmt} \longrightarrow \langle \text{State}[\underline{\mathbb{S}}], \text{IO}^{\#}, \text{SMT}, \text{Nondet} \rangle \underline{\text{Unit}} \\
\text{exec}(\text{assign}(x, e)) &= \{\ \underline{v} \leftarrow \text{eval}(e);\ \underline{\sigma} \leftarrow \text{getStore};\ \textbf{return } \text{putStore}(\underline{\sigma[x \mapsto v]})\ \} \\
\text{exec}(\text{write}(e)) &= \{\ \underline{v} \leftarrow \text{eval}(e);\ \textbf{return } \text{writeInt}(\underline{v})\ \} \\
\text{exec}(\text{seq}(s_1, s_2)) &= \{\ \_ \leftarrow \text{exec}(s_1);\ \textbf{return } \text{exec}(s_2)\ \} \\
\text{exec}(\text{skip}) &= \textbf{return } \underline{()} \\
\text{exec}(\text{cond}(e, s_1, s_2)) &= \{\ \underline{b} \leftarrow \text{eval}(e);\ \underline{s} \leftarrow \text{get} \\
&\qquad \textbf{return if}^{\uparrow\downarrow}{}_{\underline{s}}\ (\underline{b \equiv \text{true}})\ \text{exec}'(s_1, e) \\
&\qquad\qquad \textbf{else if}^{\uparrow\downarrow}{}_{\underline{s}}\ (\underline{b \equiv \text{false}})\ \text{exec}'(s_2, \neg e) \\
&\qquad\qquad \textbf{else}\ \{\ (s', e') \leftarrow \text{choice}((s_1, e), (s_2, \neg e));\ \text{exec}'(s', e')\ \}\ \} \\
\text{exec}(\text{while}(e, b)) &= \{\ \underline{s} \leftarrow \text{get};\ \text{reflect}(\underline{\text{loop}(s)})\ \} \\
\text{where } \underline{\text{loop}} &: \underline{\mathbb{S} \longrightarrow \text{List}[\mathbb{S} \times \text{Unit}]} \\
\underline{\text{loop}}(\underline{s}) &= \text{reify}(\underline{s})\{\ \underline{c} \leftarrow \text{eval}(e);\ \underline{v} \leftarrow \text{continue?}(\underline{c}); \\
&\qquad \textbf{return if}^{\uparrow\downarrow}{}_{\underline{s}}\ (\underline{v})\ \{\ \_ \leftarrow \text{exec}'(b, e);\ \underline{s'} \leftarrow \text{get};\ \textbf{return } \text{reflect}(\underline{\text{loop}(s')})\ \} \\
&\qquad\qquad \textbf{else } \text{exec}'(\text{skip}, \neg e)\ \}
\end{aligned}
$$

Fig. 12. Definition of the 2-stage symbolic definitional interpreter for statements.

The object language syntax (expressions and statements) is first stage, hence it is not underlined. We classify runtime constructs as second stage, including values $\underline{v} \in \underline{\text{Value}}$, stores $\underline{\sigma} \in \underline{\text{Store}}$, and path conditions $\underline{\pi} \in \underline{\text{PC}} = \underline{\mathcal{P}(\text{Expr})}$. Although we represent the path conditions as syntactic expressions, we classify them as second stage, as their construction happens there. The path conditions can be transformed to proper SMT encodings, therefore will not pose interpretation overhead.

The functions eval and exec are first stage, but return second-stage $\underline{\text{Value}}$s, respectively $\underline{\text{Unit}}$ with effects. As explained earlier, algebraic effects should not leak into the second stage and hence remain first-stage only. However, the effect operations and handlers do compute second-stage terms, e.g., the State effect carries now a second-stage symbolic state (Section 3.2) of type $\underline{\mathbb{S}}$.

Modulo binding times, the definition of the staged eval remains identical to the version from Section 3. This holds for most cases of exec, too. However, the cases for cond and while are more intricate and we discuss them in the following sections.

$$\mathsf{reify} : \underline{\mathbb{S}} \to \langle \mathsf{State}[\underline{\mathbb{S}}], \mathsf{SMT}, \mathsf{IO}^{\#}, \mathsf{Nondet}\rangle \underline{\mathsf{Unit}} \to \underline{\mathsf{List}[\mathbb{S} \times \mathsf{Unit}]}$$
$$\mathsf{reify}(\underline{s_0}) = \mathrm{NondetHandler} \circ \mathrm{SMTHandler} \circ \mathrm{IO}^{\#}\mathrm{Handler} \circ \mathrm{StateHandler}(\underline{s_0})$$
$$\mathsf{reflect} : \underline{\mathsf{List}[\mathbb{S} \times \mathsf{Unit}]} \to \langle \mathsf{State}[\underline{\mathbb{S}}], \mathsf{SMT}, \mathsf{IO}^{\#}, \mathsf{Nondet}\rangle \underline{\mathsf{Unit}}$$
$$\mathsf{reflect}(\underline{ss}) = \{\ \underline{s} \leftarrow \mathsf{choice}(\underline{ss});\ \_ \leftarrow \mathsf{put}(\underline{\mathrm{fst}(s)});\ \textbf{return}\ \underline{\mathrm{snd}(s)}\ \}$$

Fig. 13. Implementation of reify and reflect.

4.3 Staging Conditionals

Attempt 1 (ill-formed). A first attempt might look as follows (simplified without considering symbolic values and nondeterminism):

$$\mathrm{exec}(\mathrm{cond}(e, s_1, s_2)) = \{\ \underline{b} \leftarrow \mathrm{eval}(e);\ \underline{\textbf{if}}\ (\underline{b})\ \mathrm{exec}(s_1)\ \underline{\textbf{else}}\ \mathrm{exec}(s_2)\ \}$$

Unfortunately, this is not a well-formed two-stage binding-time notation, because $\underline{\textbf{if}}$ requires its branches to be both second-stage, but $\mathrm{exec}(s_i)_{i\in\{1,2\}}$ produces first-stage effect computations. We could delay $\mathrm{exec}(s_i)$ into the second stage, but that requires second-stage effects and handlers, which is undesirable for performance.

Attempt 2 (ill-behaved). There is an "obvious" well-formed way that fixes of the problem:

$$\mathrm{exec}(\mathrm{cond}(e, s_1, s_2)) = \{\ \underline{b} \leftarrow \mathrm{eval}(e);\ \underline{c_1} \leftarrow \mathrm{exec}(s_1);\ \underline{c_2} \leftarrow \mathrm{exec}(s_2);\ \underline{\textbf{if}\ (b)\ c_1\ \textbf{else}\ c_2}\ \},$$

which sequentially executes the first-stage effects of each branch, but produces ill-behaved second-stage code. To see why, consider a simplified example that applies an indexed state handler StateHandler maintaining a second-stage state value:

$$\mathrm{StateHandler}(\underline{0})\{\ \underline{c_1} \leftarrow (\mathsf{put}(\underline{1});\ \mathsf{get});\ \underline{c_2} \leftarrow (\mathsf{put}(\underline{2});\ \mathsf{get});\ \underline{\textbf{if}\ (\mathrm{true})\ c_1\ \textbf{else}\ c_2}\ \}$$

Evaluating it results in $(\underline{2}, \underline{1})$, where $\underline{2}$ is the state and $\underline{1}$ is value returned from the "then" branch. However, since the condition is $\underline{\mathrm{true}}$, we would expect to observe the effects of the "then" branch only, resulting in $(\underline{1}, \underline{1})$. This discrepancy comes from the fact that the underlying monad computation is executed sequentially – binding $\underline{c_2}$ has changed the handler state left after binding $\underline{c_1}$. In future work, we would like to investigate if handlers for arrow or idiom computations [Lindley 2014; Pieters et al. 2020] are more adequate in this situation.

Moreover, the metalanguage's $\underline{\textbf{if}}$ is a pure *expression* that cannot capture the branch effects encapsulated in the effect computation. In contrast, the object language Imp's conditional is a *statement* where each branch only performs effects (yielding $\underline{\mathsf{Unit}}$ values), and the effect of a branch should carry over to the join point of the conditional. We need to rule out branch interference when handling first-stage effects and ensure that handlers update their state based on a proper first-stage representation of $\underline{\textbf{if}}$'s second-stage join point.

Attempt 3 (well-formed and well-behaved). To properly stage cond with algebraic effects, we need a bridge that connects *second-stage representation of pure values* and *first-stage effectful computation.* We adapt the terminology "reflection" and "reification" from the metaprogramming literature [Friedman and Wand 1984]: reflection transforms second-stage representations of data to first-stage effectful computation, and reification transforms effectful computation into second-stage values.

Luckily, we already have the reification function, which is the composed symbolic execution handler operating on first-stage effect computations (although NondetHandler requires minor changes, which will be discussed in Section 4.5). The implementations of reify/reflect are shown in Figure 13.[3] The reflect function takes a second-stage list of $\underline{\mathbb{S}}$-and-$\underline{\mathsf{Unit}}$, which are the representation

[3]To be precise, $\mathrm{StateHandler}(\underline{s_0})$ yields a value of type $\langle E\rangle\underline{\mathbb{S}} \times \underline{\mathsf{Unit}}$, which is implicitly converted to $\langle E\rangle\underline{\mathbb{S} \times \mathsf{Unit}}$.

of the symbolic execution results of multiple paths, and uses the nondeterminism effect syntax choice to lift them back into a first-stage effectful computation. It is worth noting that this choice operation takes a second-stage list of symbolic execution results; in Section 4.5, we will discuss the stage-polymorphic behavior of the choice operation. After obtaining each single $\underline{\mathbb{S}}$-and-$\underline{\mathsf{Unit}}$ pair, we re-install the state $\underline{\mathbb{S}}$ using put and return the $\underline{\mathsf{Unit}}$ value.

With the reify and reflect functions in hand, we can now correctly stage cond (Figure 12). We first define a syntactic sugar $\mathbf{if}^{\uparrow\downarrow}$ which takes the state $\underline{s}$ before branching, the condition value $\underline{c}$ of type $\underline{\mathbb{B}}$, and two first-stage effectful computations e_1 and e_2. The desugared form applies reify with state s in each branch, resulting in values of type $\underline{\mathsf{List}[\mathbb{S} \times \mathsf{Unit}]}$, and uses second-stage $\underline{\mathbf{if}}$ to make a choice over the "then" list or "else" list based on the value of $\underline{c}$, and finally uses reflect to recover the whole second-stage $\underline{\mathbf{if}}$ expression to first-stage effect computation.

$$\frac{s : \underline{\mathbb{S}} \quad c : \underline{\mathbb{B}} \quad e_1 : \langle E \rangle \underline{\mathsf{Unit}} \quad e_2 : \langle E \rangle \underline{\mathsf{Unit}} \quad E = \mathsf{State}[\underline{\mathbb{S}}], \mathsf{SMT}, \mathsf{IO}^{\#}, \mathsf{Nondet}}{\mathbf{if}^{\uparrow\downarrow}{}_{\underline{s}}\ (\underline{c})\ e_1\ \mathbf{else}\ e_2 \triangleq \mathsf{reflect}(\underline{\mathbf{if}}\ (\underline{c})\ \mathsf{reify}(\underline{s})(e_1)\ \underline{\mathbf{else}}\ \mathsf{reify}(\underline{s})(e_2)) : \langle E \rangle \underline{\mathsf{Unit}}} \quad (\mathbf{if}^{\uparrow\downarrow})$$

We can now define the staged symbolic execution of the cond case using $\mathbf{if}^{\uparrow\downarrow}$ (Figure 12, case cond), which is similar to the unstaged counterpart. Additionally, note that the last branch again uses the stage-polymorphic nondeterminism effect syntax choice to choose over *first-stage* statements s_i and their corresponding path condition.

4.4 Staging Loops

It is relatively straightforward to stage loops if we have pre-unfolded while for a constant number of times into several conds, which we addressed in the previous section. More sophisticated execution strategies for loops may depend on next-stage dynamic heuristics. Such dynamic heuristics decide when to stop executing a loop, for example, based on the second-stage total number of explored paths. We assume an effect operation continue? returning a $\underline{\mathbb{B}}$ value, indicating if the loop can continue. This effect can be handled by generating a second-stage native effect operation.

To actually execute loops in the second stage, an intuitive idea is to generate a recursive function that invokes itself at the second stage. However, it is not immediately clear what the type of such a function would be. Again, we need to use the reify/reflect function and $\mathbf{if}^{\uparrow\downarrow}$ to ensure such looping functions act only on pure, second-stage values.

We show in Figure 12 the $\underline{\mathsf{loop}}$ function. Note that the function itself is a second-stage function, indicated by its type $\underline{\mathbb{S} \to \mathsf{List}[\mathbb{S} \times \mathsf{Unit}]}$. The function takes an initial state $\underline{s}$ of type $\underline{\mathbb{S}}$, which the state will be used to reify the following computation: first, the condition is evaluated to a value $\underline{c}$; second, we query if the loop can continue (which is an effect operation that results in generating code with native effects), and get a value $\underline{v}$ of $\underline{\mathbb{B}}$; finally, we use $\mathbf{if}^{\uparrow\downarrow}$ to insert a "branch" between two effect computations, where the first branch executes the loop body b and recursively invokes $\underline{\mathsf{loop}}$ with the state $\underline{s'}$ after executing the loop body, and the second branch is simply executing skip.

After defining $\underline{\mathsf{loop}}$, the execution of while can be simply derived (Figure 12): we get the current state $\underline{s}$, invoke $\underline{\mathsf{loop}(s)}$ yielding a second-stage application, and reflect the second-stage representation back to effectful computation.

4.5 Nondeterminism Effect and Handler for Staging

When implementing the reflect function (Figure 13), we use the effect operation choice in a stage-polymorphic manner: It can take either a second-stage list of choices, or a first-stage list of choices. We refine the effect signature of Nondet to be explicitly stage-polymorphic (Figure 14, left). The choice with first-stage candidates is used to nondeterministically execute multiple statements at

$$\begin{aligned}
&\textbf{effect}\ \mathsf{Nondet} := \\
&\quad \mathsf{choice}[X] : \mathsf{List}[X] \to X \\
&\quad \mathsf{choice}[\underline{X}] : \underline{\mathsf{List}[X]} \to \underline{X}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{NondetHandler} : \langle \mathsf{Nondet} \rangle \underline{A} \to \langle\rangle \underline{\mathsf{List}[A]} \\
&\mathrm{NondetHandler} = \textbf{handler} \\
&\qquad \textbf{return}(\underline{x}) \mapsto \textbf{return}\ \underline{\mathrm{List}(x)} \\
&\qquad \mathsf{choice}(\underline{xs}, k) \mapsto \textbf{return}\ \underline{\mathrm{fold}(xs, \mathrm{List}(), f)} \\
&\qquad\qquad \text{where } \underline{f} : \underline{\mathsf{List}[T] \times T \to \mathsf{List}[T]} \\
&\qquad\qquad\quad \underline{f}(\underline{acc}, \underline{x}) = \underline{acc}\ \underline{++}\ k(\underline{x})
\end{aligned}$$

Fig. 14. Stage-polymorphic nondeterminism effect (left) and handler of choice with second-staged data (right)

a *fork* point, as the syntactic AST is indeed statically known; while the choice with second-stage candidates is used to reflect an unknown number of results back to effect computation at a *join* point, as in general we do not have the information of how many paths will be explored.

If the candidates are formed in a second-stage list, its corresponding handler used in reify also needs to be marginally adapted (Figure 14, right): The Nondet effect now must come last in the effect row, as the type indicates. Permitting more effects to come after Nondet interacts badly with choice invocations having arguments whose length is unknown in the first stage. In that case, we cannot handle Nondet without delaying the effect computation or handlers into the second stage. During the development of our approach, we have found this limitation is not an issue at all, as the Nondet effect is indeed the last effect for symbolic execution.

If we restrict NondetHandler's effect row in the above way, then its unstaged and staged versions can have the same piece of code, modulo the binding-time annotations. Thus, NondetHandler in Figure 14 should be in fact considered stage-polymorphic (although the figure specifically shows two-stage binding times).

4.6 Assembling Staged Symbolic Definitional Interpreters

Now, we can combine exec and reify to derive the staged symbolic definitional interpreter. Given a program p and initial state $\underline{s_0}$, we have a second-stage representation of the result,

$$(\mathsf{reify}(\underline{s_0}) \circ \mathrm{exec})(p) : \underline{\mathsf{List}[\mathbb{S} \times \mathsf{Unit}]}$$

which can be readily used for code generation if they are implemented with an actual MSP system.

It is worth noting that here we have achieved the first Futamura projection by specializing the symbolic interpreter exec. The result of type $\mathsf{List}[\mathbb{S} \times \mathsf{Unit}]$ represents the symbolic execution without the interpretation and effect handling overhead. It is also reusable and composable, in the sense that we can reuse the specialization result under other program contexts.

5 IMPLEMENTATION

We implement our approach by embedding an algebraic effect system using freer monads [Kiselyov and Ishii 2015] into Scala. We utilize the Lightweight Modular Staging framework (LMS) [Rompf and Odersky 2010] for staging, which allows expressing binding times at the type level. With LMS, we can specialize the staged symbolic interpreter given static programs and generates second-stage programs in C++. This section describes the implementation and code generation.

5.1 Algebraic Effects using Free Monads

We follow Kiselyov and Ishii [2015]'s freer monad to implement algebraic effects and handlers with open unions for type-level effect rows in Scala (Figure 15). Computations of type `Comp[R, A]` return values of type `A`, and induce user-defined algebraic effects described by the effect row `R`, which is a subtype of the type-level list `Eff`. A computation finishes either with an answer value

```
sealed trait Eff // type-level list representing an effect row
trait ∅ extends Effs;  trait ⊗[A[_], TL <: Eff] extends Eff
abstract class Comp[R <: Eff, +A] // freer monads
case class Return[R <: Eff, A](a: A) extends Comp[R, A]
case class Op[R <: Eff, A, X](op: U[R, X], k: X ⇒ Comp[R, A]) extends Comp[R, A]
```

Fig. 15. Freer monad definition in Scala.

(`Return`), or an impure effect operation invocation (`Op`), carrying the effect operation `op`, which is contained in the effect row `R` and has response (continuation) type `X`, along with a continuation `k` of type `X ⇒ Comp[R, A]`, defining the next computation step after handling the operation.

We also use `T ∈ R` to represent that an effect operation `T` exists in an effect row `R`. The effect operations (e.g., get, put), are `case classes` and will be lifted into the desired effect row via `inj`.

```
trait ∈[T[_], R <: Eff] {
  def inj[X](sub: T[X]): U[R, X]
  def prj[X](u: U[R, X]): Option[T[X]]
}
```

For illustration, we now define the effect signature of symbolic execution `SymEff`, which follows our description in Section 3. `State[S, *]` is the *State* effect, parameterized by `S` and represented by a higher-kinded type in Scala. `IO`, `SMT`, and `Nondet` are similarly defined, and ∅ is the void effect.

```
type S = (Store, PC);   type SymEff = State[S, *] ⊗ (IO ⊗ (SMT ⊗ (Nondet ⊗ ∅)))
```

In order to define effect-polymorphic computations, one may write programs against an abstract effect row `R` and implicitly require `T ∈ R` as a type-level constraint, which specifies which effects must be present in `R`. For example, one may require that `eval` perform the state effect, as follows:

```
def eval[R <: Eff](e: Expr)(implicit I: State[S, *] ∈ R): Comp[R, Value].
```

5.2 Multi-Stage Programming with LMS

Lightweight Modular Staging (LMS) [Rompf and Odersky 2010] is an extensible Scala framework for multi-stage programming. The framework utilizes the implicit conversions and overloading mechanisms in Scala [Rompf et al. 2012] to allow writing stage-polymorphic programs [Amin and Rompf 2018; Carette et al. 2009; Hofer et al. 2008; Ofenbeck et al. 2017]. In contrast to syntactic MSP languages, LMS transforms the front-end programs to an intermediate representation, which will be further transformed to target code. A built-in ANF transformation [Flanagan et al. 1993] avoids unnecessary code duplication and preserves the call-by-value evaluation order [Rompf 2016].

The core interface provided by LMS is a type constructor `Rep[T]`. A value `v` of `Rep[T]` represents a next-stage computation whose value is unknown in the current stage. To minimize syntactic burden, it is idiomatic to use Scala's implicit conversions to lift operations on values of `Rep[T]` to operations on a class defined elsewhere, whose operations build up the intermediate representations. The following code snippet shows the idea of lifting a map lookup `m(k)` to the IR:

```
implicit class MapOps[K, V](m: Rep[Map[K, V]]) {
  def apply(k: Rep[K]): Rep[V] = reflect(IRMapApply, m, k) // Other ops omitted
}
```

Using LMS, we can annotate a function's dynamic arguments and return value with `Rep[_]` type, and the binding-times of body terms will be inferred by Scala's local type inference, which corresponds to a local binding-time analysis.

LMS provides a `Base` class to be extended, which exposes the `Rep` type and minimal infrastructure for users. When extending `Base`, one may also mix in other classes that provide extensions, such as overloaded representations for data structures. The metaprograms should be written in a class that extends `Base`. Additionally, LMS supplies a `virtualize` annotation which macro-transforms the control primitives (e.g., `if`) as well into the IR, therefore we have the ability to naturally write those second-stage primitives (e.g., **if** in Section 4). The `virtualize` annotation can be used at either class-level or method-level in Scala.

Internally, LMS uses a graph-based intermediate representation [Click and Paleczny 1995]. The execution of current-stage programs with `Rep` types constructs the IR of next-stage programs. Then the IR can be sent out to a code generation backend, which determines the output code. The code generation can be highly customized, for example, generating code in different languages. During both graph construction and code generation, we may apply rewritings rules to optimize the code.

5.3 The Organization of Staged Symbolic Execution with LMS

Our staged symbolic execution engine is organized as follows. The trait `StagedSymbolicImp` is extended from `Base` and other supporting classes, where our staged symbolic executor is defined with algebraic effect signature `Comp[SymEff, Rep[Unit]]`.

```
@virtualize trait StagedSymbolicImp extends Base with .../*Other mixins omitted*/{
  def eval(e: Expr): Comp[SymEff, Rep[Value]] = ...
  def exec(s: Stmt): Comp[SymEff, Rep[Unit]] = ...
}
```

The definition of `Comp` is already parameterized to the answer type: we thus provide a `Rep[T]` to it (i.e., `Comp[R, Rep[T]]`). With this type, the free monads are first-stage data structures that represent the manipulation of the representations of second-stage computations at the effect level.

To use (specialize) an Imp program with regard to the staged symbolic execution engine, there is a `specialize` function that applies `reify` (i.e. handlers) to the result of `exec`:

```
  def specialize(s: Stmt): Rep[List[(S, Unit)]] = reify(state0)(exec(s))
```

Because `exec` yields first-stage effect computation, we can also apply handlers to resolve all effects in the same stage, which ensures the overhead of effect interpretation will not be residualized into the generated code. Given the returned value of `specialize`, LMS's back-end will further generate C++ code from the representations.

5.4 Generating Symbolic Execution Code in C++

LMS allows us to manipulate intermediate representations and customize their code generation as compiler passes. Therefore, we have the flexibility to choose among different target languages. Mainly for performance reasons, we elect to generate C++ code after specialization. We may also leverage sophisticated optimizations in existing C++ compilers.

Since we handle the effects in a purely functional way (i.e., state effects transformed to functions that take a state argument), we obtain generated C++ code free from side-effects. We also develop simple runtime representations of symbolic values, memories, and path conditions in C++, consisting of ~200 LOC. The generated code that manipulates these runtime constructs uses persistent data structures and shared pointers. We use a C++ library Immer [Puente 2017] to represent and manipulate high-level data structures such lists, maps and sets. Due to the fact that our generated code is also functional, potentially it can be parallelized without too much effort. Certainly, there is a design space of how to represent runtime symbolic values, stores, and constraints; we elect for simplicity. The generated code directly orchestrates SMT invocations with their C++ API. Our prototype implementation uses the STP SMT solver [Ganesh and Dill 2007].

```
CoinChoice : ⟨Nondet, E⟩A → ⟨E⟩List[A]
CoinChoice(p) = handler
    return(x) ↦ return List(x)
  choice(xs, k) ↦
      return k(uniformSample(xs))
```

```
CoinChoice : ⟨Nondet⟩A̲ → ⟨⟩List[A]
CoinChoice(p) = handler
    return(x) ↦ return List(x)
  choice(x̲s̲, k) ↦ unchanged, see Figure 14
  choice(xs, k) ↦ {
    val r̲ = uniformSample(0 . . . |xs| − 1)
    return buildSelection(r̲, xs, k) }
```

Fig. 16. Left: general single-stage random sampling handler for choice; Right: restricted two-stage handler that generates random sampling code.

5.5 A Staged Symbolic Executor Is a Symbolic Compiler

With all the tools in place, we are able to connect everything and construct a symbolic execution engine that generates code. The obtained staged symbolic execution engine acts as a compiler, which, given an input program, outputs instrumented C++ code where the interpretation overhead from the symbolic definitional interpreter and effect handling are eliminated.

We have turned a high-level definitional symbolic interpreter into a "symbolic compiler", by specializing it w.r.t. input programs, via the first Futamura projection [Futamura 1971, 1999].

6 FLEXIBLE PATH-SELECTION STRATEGIES VIA EFFECT HANDLERS

If a symbolic execution encounters a branching point, then the branching condition might depend on symbolic information, leaving the resulting Boolean value indeterminate. A symbolic execution engine then proceeds by choosing a branch according to a given path-selection strategy/heuristic. For instance, KLEE [Cadar et al. 2008] supports depth-first, breadth-first, random path selection strategies, etc. This section shows that our approach, based on effects and handlers, uniformly and extensibly abstracts over these strategies, which are particular interpretations of a nondeterminism effect. The handlers presented here apply to the choice (Figure 14, left) operation with first-stage known candidates, as the branch statements are syntactic and static objects. The handler for the choice operation with second-stage candidates (Figure 14, right) remains unchanged.

6.1 Random Path Sampling

One may also explore paths of nondeterministic computations by random selection at branches, which is straightforward to implement with effect handlers. Figure 16 (left) shows the general single-stage handler that uniformly selects one element from the candidates *xs*, i.e., only one path will be executed. The selection is delegating to a real random source

$$\text{uniformSample} : \mathsf{List}[A] \to A,$$

which invokes a system library implementing a fair random selection.

The probability that each element $x \in xs$ being selected is $1/|xs|$. We shall still commit to resolve the nondeterminism effect to a list of execution results, which can be convenient when performing multiple times of sampling.

To integrating the random sampling handler with our staged symbolic interpreter, we have two choices to move forward: 1) execute the sampling procedure in the first stage; the generated code is determined by this choice; 2) execute the sampling procedure in the second stage; each time we execute the generated code, we may have a different sampling result.

The former is obviously less useful, so we commit to the latter approach (Figure 16, right). The handler clause for choice(xs, k) first defines a second-stage random number $\underline{r} : \underline{\mathsf{Int}}$, uniformly

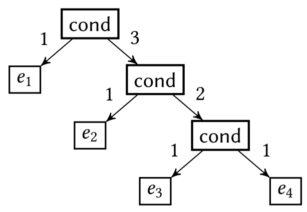

Fig. 17. Preprocessing step for fair random path selection in a loop-unrolled source program with three branches, propagating branch counts along edges bottom up.

$$
\begin{aligned}
&\text{Explore} = \textbf{shallow\_handler} \\
&\qquad \textbf{return}(x) \mapsto \{\ \text{acc}(sol, x);\ \text{schedule}()\ \} \\
&\quad \text{choice}(xs, k) \mapsto \{\ insert(ps,\ ks);\ \text{schedule}()\ \} \\
&\text{where} \qquad ks = \text{map}(xs, \lambda x.\lambda().k(x)) \\
&\quad \text{schedule}() = \{\ \textbf{if}\ (\text{nonEmpty?}(ps))\ \{c \leftarrow \text{next}(ps);\ \text{Explore}(c)\}\ \textbf{else}\ \{\ \textbf{return}\ sol\ \}\}
\end{aligned}
$$

Fig. 18. Single-stage generic nondeterminism handler for exhaustive path exploration.

sampled from the interval $[0, |xs| - 1]$, and then uses an auxiliary function

$$\text{buildSelection} : \underline{\mathsf{Int}} \to \mathsf{List}[X] \to (X \to \underline{\mathsf{List}[A]}) \to \underline{\mathsf{List}[A]}$$

to build a second-stage nested branching program using $\underline{\textbf{if}}$, where the i-th branch corresponds to the computation of applying continuation k to the i-th element of xs.

6.2 Fair Random Path Sampling

We further generalize random search so that all possible paths starting at a given branch are chosen with equal probability. This strategy requires counting the number of following branches at any given branch, in order to compute the correct probability parameter for a biased coin flip. We assume that the input program has a finite number of paths and a preprocessing step (e.g., [Zhang et al. 2019]) attaches the number of paths to the AST. Figure 17 illustrates path counting on an unbalanced binary AST.

We also extend the effect operation choice to pchoice, such that the probability information is accompanied with each candidate:

$$\text{pchoice} : \mathsf{List}[X \times \mathsf{Float}] \to X.$$

Then we can handle pchoice in the similar fashion of the uniformly random sampling, but using a biased random sample function. The code snippet below shows the handler clause with a second-stage sampling function biasedSample that additionally takes the weight of each candidate:

$$\text{pchoice}(xps, k) \mapsto \{\ \textbf{val}\ \underline{r} = \underline{\text{biasedSample}(0 \ldots |ps| - 1, ps)};\ \textbf{return}\ \text{buildSelection}(\underline{r}, xs, k)\ \}$$

where ps is the list of probabilities and xs is the list of candidates, each projected from the xps list.

6.3 Depth- and Breadth-First Search

The nondeterminism handler in Section 3.4 implements an exhaustive exploration of all paths in a depth-first manner. With a few generalizations, we obtain a handler (Explore in Figure 18) that is variable in the search strategies and accumulation of explored results.

The handler is parametric over a data structure ps, which collects and selects branches of the choice effect. Accordingly, insert, nonEmpty?, and next, are operations for inserting the branches into ps, checking for its non-emptiness, and selecting the next unexplored branch from it. For instance, we obtain the depth-first exploration of paths by choosing a stack for *ps*, and the breadth-first strategy by choosing a queue. Furthermore, it is parameterized over an additional collection *sol* and an operation acc, for inserting the result of terminated paths into it. If all result values should be returned, we can take the type of *sol* to be a list or a set.

Explore is a *shallow handler* [Hillerström and Lindley 2018]. The intuitive difference between standard deep and shallow handlers is analogous to a complete fold over a tree (i.e., deep, Church encoding) versus a single fold (i.e., shallow, Scott encoding). Shallowness is useful to implement cooperative concurrency via handlers, because we can just store the continuations of the choice effect as they are in the *ps* collection. Specialized variants of Explore have been used for implementations of asynchrony operations, and cooperative interleaving concurrency in the algebraic effects literature [Bračevac et al. 2018; Dolan et al. 2017; Leijen 2017a].

Similar to the two-stage CoinHandler, it is appealing to generate code that makes the scheduling happen at the second-stage execution. We sketch the two-stage version based on Figure 18: First, we will have to transform the candidates $\underline{xs}$ to a second-stage list of thunks (functions) $\underline{ks}$, where the body of each thunk $k(\underline{x})$ is a first-stage application yielding second-stage representations. Second, it is necessary to have runtime support for the data collection *ps*, *sol* and operations insert, next, etc. Finally, those operations and collections can store, retrieve, or invoke the thunks, all in the second stage, which implement different scheduling schemes.

7 SCALING UP: FROM IMP TO LLVM IR

In the previous sections, we have presented our approach to building symbolic execution engines for the Imp language, which is small but sufficient to show how algebraic effects and multi-stage programming can improve the flexibility and performance. We now discuss extending our approach to a real-world language: a subset of the LLVM IR language, for which we implemented a prototype symbolic execution engine based on our approach.

We model a number of essential instructions, including `br`, `ret`, `switch`, `unreachable`, `alloca`, `store`, `load`, `getelementptr`, `icmp`, `sext`, `zext`, `call`, `phi`, `select`, and a few arithmetics. These instructions are sufficient to cover a large set of programs and many non-trivial language features, e.g., dynamic allocation and memory indirection. We devote the remainder of this section to necessary adaptations for new challenges.

Block-based Semantics and Jumps. LLVM IR features a block-based, static single assignment form. Each block has a label and consists of a sequence of statements and a terminal statement. For example, a terminal statement can be a direct jump or a conditional jump to other blocks (both using the `br` instruction). Other forms of control flow transfers are function calls `call` and multiway conditional jumps `switch`.

To effectively generate code for these control structures, we generate next-stage functions for each LLVM function and its contained blocks. Jumps and function calls are translated to invocations of the second-stage functions. However, if a jump target is dynamically computed, we must also generate code that computes the target (i.e., the address to which to jump). Since the execution of blocks are generated as functions, loops are straightforward to handle — as they are indirectly translated into function calls representing jumps.

Memory. Compared to Imp, LLVM also has a fully-featured, machine-level, memory model, which introduces a notion of pointers. A notion of global store is similar to what we have in Imp. In addition, locally-used memory can be dynamically allocated using the `alloca` instruction. The LLVM IR program can `store` values into locations, and `load` values from locations. Low level memory layout details, such as alignments, are also introduced. To model this behavior in addition to function calls, we use two contiguous regions of memory for representing the "heap" and "stack" respectively; they are carried by the state effect. Their second-stage representation in C++ is the `flex_vector` data structure from Immer (cf. Section 5.4), which can be accessed using integer indices. Meanwhile, locations are also values (i.e., pointers) that can be passed or returned. Our prototype uses a concrete memory model, therefore it has to concretize the jump address if it is a symbolic value.

Implementation. According to LLVM IR's syntactic structure, we have the following top-level functions, where the effect row `E` contains the state, nondeterminism, IO, and SMT effect.

```
def execTerm(inst: Terminator):           Comp[E, Rep[Value]]
def execBlock(bb: BasicBlock):            Comp[E, Rep[Value]]
def execInst(inst: Instruction):          Comp[E, Rep[Unit]]
def execValueInst(inst: ValueInstruction): Comp[E, Rep[Value]]
def eval(v: LLVMExpr):                    Comp[E, Rep[Value]]
```

The function `execTerm` executes terminator instructions, whose result will be used as the value of executing its containing block in `execBlock`. Instructions such as `assign` and `store` are handled by `execInst`, whose return value in the free monad encoding is `Rep[Unit]`, as those instructions only perform side effects. Evaluating value instructions (such as `load` and arithmetics) and expressions (such as constants) both yield `Rep[Value]` in the effect encoding, and is handled by `execValueInst` and `eval` respectively. When implementing those functionalities, we are able to reuse the same effects, handlers, as well as the reify/reflect function introduced in the previous sections.

The generated code of our prototype is again purely functional. States are passed among the generated C++ functions, which represent the execution of LLVM blocks and functions.

8 PERFORMANCE EVALUATION

In this section, we present the performance evaluation brought by staging. We build the prototype staged symbolic execution engine for a subset of LLVM IR, as described in the last section, and compare the performance with the unstaged baseline (Scala) and KLEE [Cadar et al. 2008], which are both interpretation-based engines. We aim to answer the following two questions on performance:

(1) Compared with a high-level definitional symbolic interpreter written with algebraic effects in Scala, how much performance improvement can be brought by eliminating the interpretation and effect handling overhead? To answer this question, we measure the path traversal time of the unstaged version and staged version.
(2) Compared with a state-of-the-art symbolic interpreter written in C++, KLEE, how is the performance of our prototype build upon our staging and algebraic effect approach? To answer this question, we use micro benchmarks which involves building SMT expressions and solver invocations. Our prototype and KLEE both use the STP[4] solver and query the solver for generating concrete inputs.

Environment. We run all the experiments on Ubuntu 20.04 (kernel 5.4.0), with an 8-core AMD 3700X CPU and 32GB RAM. We compile the C++ programs generated by our prototype together with Immer using `g++` version 9.3.0, with compilation option `-std=c++17 -O3`. The benchmarks and input to the symbolic execution engines are LLVM IR programs. The LLVM IR files are generated by `clang` [5] from C programs. Running times of Scala code are measured by recording the time of function-level executions. Running times of executables are measured by the `time` command. For all experiments, we set the timeout for 1 hour execution.

[4]https://stp.github.io/
[5]clang version 10.0.0; LLVM IR generated with `-O0 -S -fno-discard-value-names -emit-llvm`

Table 1. Description of the benchmarks, including the number of static instructions (#inst, excluding invocations of `klee_make_symbolic`), the number of total executed instructions (#exec inst), the number of paths (#path), and a list of used instructions in the program.

name	#inst	#exec inst	#path	IR instructions and features used
`multipath-1`	146	29733	1024 (2^{10})	`alloca`, `store`, `load`, `br`, `icmp`, `add`, `ret`
`multipath-2`	230	2687035	65536 (2^{16})	same as above
`multipath-3`	285	51380299	1048576 (2^{20})	same as above
`maze`	156	65210	309	above + `sext`, `call`, `getelementptr`, `switch`, array

Table 2. Running time of the multi-path benchmarks. 1) "unstaged" reports pure execution time of the definitional symbolic interpreter written in Scala, which accumulates path conditions but will not invoke external solvers. 2) "KLEE" reports the running time of KLEE (ver. 2.1) with `-max-memory=100000`. 3) "staged" reports the running time of generated C++ programs, staged from our prototype. Both KLEE and staged code invokes STP for generating one input. The staging time (i.e., specialization time) and the size of generated programs are also reported.

benchmark	unstaged (Scala)	KLEE	staged (C++)	staging time	generated LOC
`multipath-1`	0.58 s	0.087 s	0.055 s	0.9 s	1741
`multipath-2`	63.37 s	10.41 s	4.80 s	1.67 s	2731
`multipath-3`	timeout	350.35 s	96.26 s	2.08 s	3391

Benchmarks. Table 1 shows the description of the benchmarks we used. The `multipath-*` series of benchmarks are made to be small but have exponential number of paths, involving basic memory operations, branching, and simple arithmetic instructions. For a benchmark with 2^n paths, the number of symbolic variables in this benchmark is n. We also collect a program `maze` found in a KLEE tutorial [Manzano 2010], which further exhibits more involved instructions (e.g. `getelementptr`) and data structures (e.g. arrays). All execution engines are expected to exhaustively explore every path in the space.

Evaluation Question 1. Table 2 shows the running time of the high-level definitional symbolic interpreter (unstaged) and the generated programs (staged). We additionally report the staging time spent on specialization and the line of numbers of generated C++ programs. We can observe that 1) the staging time is relatively small, proportional to the number of instructions. Compared with the actual execution time, staging time can be negligible when the search space becomes larger. 2) The size of the generated code (generated LOC in Table 2) is also proportional to the number of instructions. 3) The running time of staged programs (i.e., the running time of the second stage) significantly outperforms their unstaged counterparts. From the preliminary result, we may conclude that staging can bring 10~30x running time improvement. As the number of paths increases, we can expect the ratio of performance improvement also increases. Note that the unstaged interpreter does not invoke SMT solvers, but the staged one invokes STP for generating one test case. Thus, the performance gap in fact is even greater.

Evaluation Question 2. The results are also shown in Table 2. Both KLEE and our generated code invokes STP for querying one test case on `multipath-*`. In addition, we evaluate an end-to-end symbolic execution on `maze`, in the sense that both KLEE and our prototype not only explores every path, but also invokes STP for generating inputs of every reachable path. KLEE finishes 309 paths in 0.27 seconds, and our generated code finished the same number of paths in 0.14 seconds.

We still observe that our staged tool outperforms KLEE, approximately 2~3x faster. We expect that the performance of our prototype can be further improved, by 1) using low-level data structures to represent the program states at runtime, instead of using functional data structures, 2) utilizing the LMS framework to perform staging-time optimizations and generating smaller programs.

9 RELATED WORK

Symbolic Execution. Originally proposed in the 1970s [Boyer et al. 1975; Howden 1977; King 1976], symbolic execution is widely used for program testing and debugging. Overviews and extensive surveys of modern symbolic execution can be found in Baldoni et al. [2018]; Cadar and Sen [2013]; Yang et al. [2019]. The symbolic execution technique considered in this paper is forward symbolic execution; Schwartz et al. [2010] discussed this technique in a security context, using a compact low-level intermediate language, and presented a formalization of forward symbolic execution in small-step operational semantics. That work also discusses existing challenges of symbolic execution and potential mitigations.

Widely used symbolic execution engines are mostly interpretation-based [Anand et al. 2007; Cadar et al. 2008; Păsăreanu and Rungta 2010] or instrumentation-based [Burnim 2014; Cadar et al. 2006]. Commonly, assembly- or binary-level symbolic execution engines are implemented based on instrumentation [Luk et al. 2005]. Nelson et al. [2019] used the Rosette DSL [Torlak and Bodik 2014] to build symbolic interpreters and verifiers for low-level languages and ISAs, including RISC-V, LLVM, and x86-32. Very recently, SymCC uses LLVM's infrastructure to instrument LLVM IR and generate binary code that performs symbolic execution [Poeplau and Francillon 2020], obtaining a compiler for symbolic execution. However, SymCC's approach is substantially different from this paper in several aspects: 1) SymCC does not use the Futamura projection, i.e., deriving compilers by specializing interpreters. It is implemented as an LLVM pass, which is closer to a traditional instrumentation-based approach. 2) The generated code by SymCC performs single-path concolic execution and does not involve nondeterministic execution or path selection strategies.

In a functional programming context, Mensing et al. [2019] described an approach that derives symbolic execution engines from definitional interpreters. Similar to our implementation, the definitional interpreter is written with free monads. But Mensing et al. [2019]'s symbolic execution engine does not use an external SMT solver; instead, a tiny reasoning engine in the fashion of miniKanren [Byrd et al. 2017; Friedman et al. 2018] is embedded. Therefore, the symbolic executor can be used to generate test cases, albeit its scalability is limited. Darais et al. [2017] derived a King-style symbolic executor from a big-step monadic abstract interpreter, notably with a termination guarantee. Higher-order symbolic execution has been applied to contract verification [Nguyen et al. 2018, 2014]. Targeting modern dynamic and interpretation-based languages such as Python and Lua, Bucur et al. [2014] proposed to leverage the existing interpreter as the semantic specification and turn it into a symbolic execution engine by instrumenting the interpreter itself.

Our staging technique is conceptually similar to, but should not be confused with, Siddiqui and Khurshid [2012]'s staged symbolic execution, where the first stage generates abstract symbolic inputs that will be shared and used across other methods in later stages. This paper instead leverages metaprogramming techniques such as MSP [Taha and Sheard 1997] and runtime code generation.

Effects in Functional Programming. Using monads to uniformly structure the denotational semantics of computational effects was pioneered by Moggi [1991]. Shortly after that, monads became popular as a programming abstraction in functional languages, representing effects in an immutable, referentially transparent manner [Peyton Jones and Wadler 1993; Wadler 1992]. The predominant approach to composing multiple effects is monad transformers [Liang et al. 1995; Steele 1994], which, however do not compose as well as algebraic effects and handlers [Kammar et al. 2013]. Resolving their composability issues is an active area of research.

Algebraic effects stem from Plotkin and Power [2003] investigating the equational laws behind specific monadic effects, eventually leading Plotkin and Pretnar [2013] to propose effect handlers as modular denotations for effect algebras. A similar but slightly less modular approach appeared in [Cartwright and Felleisen 1994], which delegates effects to a monolithic, central authority. There are plenty of new language implementations of algebraic effects and handlers [Chandrasekaran et al. 2018], due to its convenience for defining and interpreting effects.

Besides modeling the semantics of (imperative) languages, monads are used to formalize and understand non-standard semantics such as abstract interpretation [Sergey et al. 2013] and query languages [Breazu-Tannen et al. 1992]. We argue that symbolic execution should not be an exception.

Program Specialization. The idea of program specialization can be traced back to Kleene's S_n^m theorem [Kleene 1938]. Futamura [1971] introduced a hierarchy of specializations of self-interpreters for automatically deriving compilers and compiler generators, known as the Futamura projections [Futamura 1971, 1999]. The two-level binding-time notation and the two-level λ-calculus are due to [Nielson 1989; Nielson and Nielson 1988, 1992]. Partial evaluation [Jones et al. 1993] and multi-stage programming (MSP) [Taha 1999; Taha and Sheard 1997] are two related realizations to achieve program specialization systematically: the former relies on automatic binding-time separation (online or as an offline analysis), while the latter allows programmers to control which part of the program should be specialized using explicit annotations. The classical approach of MSP uses syntactic quotations and splices and has been widely adopted in functional languages, including Lisp and Scheme [Bawden 1999], MetaML [Taha and Sheard 2000], F#, MetaOCaml [Calcagno et al. 2003; Kiselyov 2014], Template Haskell [Sheard and Jones 2002], as well as Dotty [Stucki et al. 2018]. This paper uses Lightweight Modular Staging (LMS) [Rompf and Odersky 2010], a Scala library providing type-based stage annotations and evaluation-order preserving specialization [Rompf 2016]. Due to LMS's flexible IR design, we can use different back-ends to achieve heterogeneous staging, i.e., generate programs in languages other than the metalanguage. A survey of metaprogramming languages can be found in [Lilis and Savidis 2019].

Multi-stage programming has shown success in several performance-critical scenarios [Rompf et al. 2015], such as query compilation in databases [Essertel et al. 2018; Klonatos et al. 2014; Rompf and Amin 2015; Tahboub et al. 2018; Tahboub and Rompf 2020; Tahboub et al. 2019], stencil and matrix computation in HPC [Aktemur et al. 2013; Kiselyov 2018; Ofenbeck et al. 2013], etc. Metaprogramming and generative programming have been applied to build and optimize program analyzers [Torlak and Bodik 2014], abstract interpreters in particular [Boucher and Feeley 1996; Johnson et al. 2013; Wei et al. 2019, 2018], as well as to program verification [Amin and Rompf 2017].

10 CONCLUSION

The paper proposes the use of algebraic effects and program specialization techniques to derive flexible and efficient symbolic execution engines. Algebraic effects model aspects of the symbolic execution semantics as an effectful definitional interpreter, and handlers can be plugged in to implement various different execution strategies. Guided by the two-stage binding-time annotations, the symbolic interpreter is turned to a symbolic compiler that reliably eliminates the interpretation overhead and abstraction layers introduced for flexibility.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their valuable feedback. We also thank James Decker and Fei Wang for proofreading the early drafts. This work was supported in part by NSF awards 1553471, 1564207, 1918483, 1910216, DOE award DE-SC0018050, as well as gifts from Google, Facebook, and VMware.

REFERENCES

Baris Aktemur, Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2013. Shonan Challenge for Generative Programming: Short Position Paper. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation* (Rome, Italy) *(PEPM '13)*. ACM, New York, NY, USA, 147–154. https://doi.org/10.1145/2426890.2426917

Nada Amin and Tiark Rompf. 2017. LMS-Verify: abstraction without regret for verified systems programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 859–873. http://dl.acm.org/citation.cfm?id=3009867

Nada Amin and Tiark Rompf. 2018. Collapsing towers of interpreters. *PACMPL* 2, POPL (2018), 52:1–52:33. https://doi.org/10.1145/3158140

Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2007. JPF–SE: A Symbolic Execution Extension to Java PathFinder. In *Tools and Algorithms for the Construction and Analysis of Systems*, Orna Grumberg and Michael Huth (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 134–138.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657

Alan Bawden. 1999. Quasiquotation in Lisp. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1*, Olivier Danvy (Ed.). University of Aarhus, 4–12.

Dominique Boucher and Marc Feeley. 1996. Abstract Compilation: A New Implementation Paradigm for Static Analysis. In *Proceedings of the 6th International Conference on Compiler Construction (CC '96)*. Springer-Verlag, London, UK, UK, 192–207. http://dl.acm.org/citation.cfm?id=647473.727587

Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT - a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California). ACM, New York, NY, USA, 234–245. https://doi.org/10.1145/800027.808445

Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. 2018. Versatile event correlation with algebraic effects. *Proc. ACM Program. Lang.* 2, ICFP (2018), 67:1–67:31. https://doi.org/10.1145/3236762

Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. 1992. Naturally embedded query languages. In *Database Theory — ICDT '92*, Joachim Biskup and Richard Hull (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 140–154.

Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping symbolic execution engines for interpreted languages. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, Rajeev Balasubramonian, Al Davis, and Sarita V. Adve (Eds.). ACM, 239–254. https://doi.org/10.1145/2541940.2541977

Jacob Burnim. 2014. *CREST: Concolic test generation tool for C.*

J. Burnim and K. Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, USA, 443–446. https://doi.org/10.1109/ASE.2008.69

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 8 (Aug. 2017), 26 pages. https://doi.org/10.1145/3110252

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '06)*. Association for Computing Machinery, New York, NY, USA, 322–335. https://doi.org/10.1145/1180405.1180445

Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2830)*, Frank Pfenning and Yannis Smaragdakis (Eds.). Springer, 57–76. https://doi.org/10.1007/978-3-540-39815-8_4

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

Robert Cartwright and Matthias Felleisen. 1994. Extensible denotational language specifications. In *Theoretical Aspects of Computer Software*, Masami Hagiya and John C. Mitchell (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 244–272.

Sivaramakrishnan Krishnamoorthy Chandrasekaran, Daan Leijen, Matija Pretnar, and Tom Schrijvers. 2018. Algebraic Effect Handlers go Mainstream (Dagstuhl Seminar 18172). *Dagstuhl Reports* 8, 4 (2018), 104–125. https://doi.org/10.4230/DagRep.8.4.104

Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. *SIGPLAN Not.* 46, 3 (March 2011), 265–278. https://doi.org/10.1145/1961296.1950396

Cliff Click and Michael Paleczny. 1995. A Simple Graph-Based Intermediate Representation. *SIGPLAN Not.* 30, 3 (March 1995), 35–49. https://doi.org/10.1145/202530.202534

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990.* ACM, 151–160. https://doi.org/10.1145/91556.91622

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *PACMPL* 1, ICFP (2017), 12:1–12:25. https://doi.org/10.1145/3110256

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. 2017. Concurrent System Programming with Effect Handlers. In *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10788)*, Meng Wang and Scott Owens (Eds.). Springer, 98–117. https://doi.org/10.1007/978-3-319-89719-6_6

Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 799–815. https://www.usenix.org/conference/osdi18/presentation/essertel

Matthias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and P. Mager (Eds.). ACM Press, 180–190. https://doi.org/10.1145/73560.73576

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. 2007. Adding delimited and composable control to a production programming environment. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 165–176. https://doi.org/10.1145/1291151.1291178

Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer, Second Edition.* MIT Press.

Daniel P. Friedman and Mitchell Wand. 1984. Reification: Reflection without Metaphysics. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, August 5-8, 1984, Austin, Texas, USA.* ACM, 348–355.

Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Computers, Controls* 25 (1971), 45–50.

Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (01 Dec 1999), 381–391. https://doi.org/10.1023/A:1010095604496

Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification*, Werner Damm and Holger Hermanns (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 519–531.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005a. DART: Directed Automated Random Testing. *SIGPLAN Not.* 40, 6 (June 2005), 213–223. https://doi.org/10.1145/1064978.1065036

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005b. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 415–435. https://doi.org/10.1007/978-3-030-02768-1_22

Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK (LIPIcs, Vol. 84)*, Dale Miller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:19. https://doi.org/10.4230/LIPIcs.FSCD.2017.18

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic embedding of dsls. In *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*, Yannis Smaragdakis and Jeremy G. Siek (Eds.). ACM, 137–148. https://doi.org/10.1145/1449913.1449935

W. E. Howden. 1977. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering* SE-3, 4 (July 1977), 266–278. https://doi.org/10.1109/TSE.1977.231144

J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. 2013. Optimizing Abstract Abstract Machines. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. ACM, New York, NY, USA, 443–454. https://doi.org/10.1145/2500365.2500604

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice Hall.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 145–158. https://doi.org/10.1145/2500365.2500590

T. Kapus and C. Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 590–600. https://doi.org/10.1109/ASE.2017.8115669

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

Oleg Kiselyov. 2018. Reconciling Abstraction with High Performance: A MetaOCaml approach. *Foundations and Trends® in Programming Languages* 5, 1 (2018), 1–101. https://doi.org/10.1561/2500000038

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. ACM, New York, NY, USA, 94–105. https://doi.org/10.1145/2804302.2804319

S. C. Kleene. 1938. On Notation for Ordinal Numbers. *J. Symbolic Logic* 3, 4 (12 1938), 150–155. https://projecteuclid.org:443/euclid.jsl/1183385485

Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building Efficient Query Engines in a High-Level Language. *Proc. VLDB Endow.* 7, 10 (2014), 853–864. https://doi.org/10.14778/2732951.2732959

James Koppel, Gabriel Scherer, and Armando Solar-Lezama. 2018. Capturing the future by replaying the past (functional pearl). *Proc. ACM Program. Lang.* 2, ICFP (2018), 76:1–76:29. https://doi.org/10.1145/3236771

Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 193–204. https://doi.org/10.1145/2254064.2254088

Daan Leijen. 2017a. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDeICFP 2017, Oxford, UK, September 3, 2017*, Sam Lindley and Brent A. Yorgey (Eds.). ACM, 16–29. https://doi.org/10.1145/3122975.3122977

Daan Leijen. 2017b. Type directed compilation of row-typed algebraic effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 486–499. http://dl.acm.org/citation.cfm?id=3009872

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 333–343. https://doi.org/10.1145/199448.199528

Yannis Lilis and Anthony Savidis. 2019. A Survey of Metaprogramming Languages. *ACM Comput. Surv.* 52, 6, Article 113 (Oct. 2019), 39 pages. https://doi.org/10.1145/3354584

Sam Lindley. 2014. Algebraic effects and effect handlers for idioms and arrows. In *Proceedings of Workshop on Generic Programming, (WGP)*.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

Felipe Andres Manzano. 2010. The Symbolic Maze! https://web.archive.org/web/20200810223740/https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/. Accessed: 2020-08-10.

Adrian D. Mensing, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. 2019. From Definitional Interpreter to Symbolic Executor. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection* (Athens, Greece) *(META 2019)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/3358502.3361269

E. Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science* (Pacific Grove, California, USA). IEEE Press, 14–23.

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*. Springer-Verlag, Berlin, Heidelberg, 213–228.

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 225–242. https://doi.org/10.1145/3341301.3359641

Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2018. Soft contract verification for higher-order stateful programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 51:1–51:30. https://doi.org/10.1145/3158139

Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 139–152. https://doi.org/10.1145/2628136.2628156

Flemming Nielson. 1989. Two-level semantics and abstract interpretation. *Theoretical Computer Science* 69, 2 (1989), 117 – 242. https://doi.org/10.1016/0304-3975(89)90091-1

Flemming Nielson and Hanne Riis Nielson. 1985. Code generation from two-level denotational meta-languages. In *Programs as Data Objects, Proceedings of a Workshop, Copenhagen, Denmark, October 17-19, 1985 (Lecture Notes in Computer Science, Vol. 217)*, Harald Ganzinger and Neil D. Jones (Eds.). Springer, 192–205. https://doi.org/10.1007/3-540-16446-4_11

Flemming Nielson and Hanne Riis Nielson. 1988. Two-level semantics and code generation. *Theoretical Computer Science* 56, 1 (1988), 59 – 133. https://doi.org/10.1016/0304-3975(86)90006-X

Flemming Nielson and Hanne Riis Nielson. 1992. *Two-level Functional Languages.* Cambridge University Press, New York, NY, USA.

Hanne Riis Nielson and Flemming Nielson. 2007. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag, Berlin, Heidelberg.

Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2017. Staging for Generic Programming in Space and Time. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Vancouver, BC, Canada) *(GPCE 2017)*. ACM, New York, NY, USA, 15–28. https://doi.org/10.1145/3136040.3136060

Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. 2013. Spiral in scala: towards the systematic construction of generators for performance libraries. In *GPCE*. ACM, 125–134.

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. ACM, New York, NY, USA, 71–84. https://doi.org/10.1145/158511.158524

Ruben P. Pieters, Exequiel Rivas, and Tom Schrijvers. 2020. Generalized monoidal effects and handlers. *J. Funct. Program.* 30 (2020). https://doi.org/10.1017/S0956796820000106

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

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

Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19 – 35. https://doi.org/10.1016/j.entcs.2015.12.003 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 179–180. https://doi.org/10.1145/1858996.1859035

Juan Pedro Bolívar Puente. 2017. Persistence for the Masses: RRB-Vectors in a Systems Language. *Proc. ACM Program. Lang.* 1, ICFP, Article 16 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110260

John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2.*

Tiark Rompf. 2016. The Essence of Multi-stage Evaluation in LMS. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 318–335. https://doi.org/10.1007/978-3-319-30936-1_17

Tiark Rompf and Nada Amin. 2015. Functional Pearl: A SQL to C Compiler in 500 Lines of Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. ACM, New York, NY, USA, 2–9. https://doi.org/10.1145/2784731.2784760

Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. 2012. Scala-Virtualized: linguistic reuse for deep embeddings. *High. Order Symb. Comput.* 25, 1 (2012), 165–207. https://doi.org/10.1007/s10990-013-9096-9

Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle Olukotun. 2015. Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 238–261. https://doi.org/10.4230/LIPIcs.SNAPL.2015.238

Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, Graham Hutton and Andrew P. Tolmach (Eds.). ACM, 317–328. https://doi.org/10.1145/1596550.1596596

Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, Eelco Visser and Jaakko Järvi (Eds.). ACM, 127–136. https://doi.org/10.1145/1868294.1868314

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 317–331. https://doi.org/10.1109/SP.2010.26

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic Abstract Interpreters. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 399–410. https://doi.org/10.1145/2491956.2491979

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. *SIGPLAN Not.* 37, 12 (Dec. 2002), 60–75. https://doi.org/10.1145/636517.636528

Junaid Haroon Siddiqui and Sarfraz Khurshid. 2012. Staged Symbolic Execution. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (Trento, Italy) *(SAC '12)*. ACM, New York, NY, USA, 1339–1346. https://doi.org/10.1145/2245276.2231988

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

Alen Stojanov, Tiark Rompf, and Markus Püschel. 2019. A stage-polymorphic IR for compiling MATLAB-style dynamic tensor expressions. In *GPCE*. ACM, 34–47.

Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*, Eric Van Wyk and Tiark Rompf (Eds.). ACM, 14–27. https://doi.org/10.1145/3278122.3278139

Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Walid Taha. 1999. *Multi-stage programming: Its theory and applications.* Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology.

Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97), Amsterdam, The Netherlands, June 12-13, 1997*, John P. Gallagher, Charles Consel, and A. Michael Berman (Eds.). ACM, 203–217. https://doi.org/10.1145/258993.259019

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 307–322. https://doi.org/10.1145/3183713.3196893

Ruby Y. Tahboub and Tiark Rompf. 2020. Architecting a Query Compiler for Spatial Workloads. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2103–2118. https://doi.org/10.1145/3318464.3389701

Ruby Y. Tahboub, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Towards compiling graph queries in relational engines. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages, DBPL 2019, Phoenix, AZ, USA, June 23, 2019*, Alvin Cheung and Kim Nguyen (Eds.). ACM, 30–41. https://doi.org/10.1145/

3315507.3330200

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 530–541. https://doi.org/10.1145/2594291.2594340

Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/143165.143169

Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged Abstract Interpreters: Fast and Modular Whole-program Analysis via Meta-programming. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 126 (Oct. 2019), 32 pages. https://doi.org/10.1145/3360552

Guannan Wei, James Decker, and Tiark Rompf. 2018. Refunctionalization of Abstract Abstract Machines: Bridging the Gap Between Abstract Abstract Machines and Abstract Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 2, ICFP, Article 105 (July 2018), 28 pages. https://doi.org/10.1145/3236800

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 1–12. https://doi.org/10.1145/2633357.2633358

Guowei Yang, Antonio Filieri, Mateus Borges, Donato Clun, and Junye Wen. 2019. Chapter Five - Advances in Symbolic Execution. Advances in Computers, Vol. 113. Elsevier, 225 – 287. https://doi.org/10.1016/bs.adcom.2018.10.002

Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 137:1–137:31. https://doi.org/10.1145/3360563