



Compiling WebAssembly Concolic Execution with Staging, Continuations, and Snapshots

DINGHONG ZHONG, Tufts University, USA

ALEXANDER Y. BAI, New York University, USA

MIKAIL KHAN, Carnegie Mellon University, USA

GUANNAN WEI[✉], Tufts University, USA

Concolic execution is a variant of symbolic execution that runs a program simultaneously with concrete and symbolic inputs. It records the symbolic constraints encountered along a concrete execution path, then solves those constraints to generate inputs that explore new paths. Existing concolic engines generally follow one of two implementation strategies: Interpreter-based systems are comparatively simple to build but incur substantial interpretation overhead, while instrumentation-based systems avoid this overhead but typically re-execute the program from the beginning for each new input.

In this paper, we develop a new approach that achieves the best of both worlds. Starting from the concrete semantics of the target language, we first develop a definitional concolic interpreter and stage it to compile away interpretation overhead while retaining the simplicity of an interpretation-based implementation. By expressing the staged interpreter in continuation-passing style, we can capture execution snapshots at branch points and resume from them when exploring alternative paths, avoiding repeated execution from the program entry. Because snapshot-reuse can itself incur overhead, we further develop a heuristic that favors snapshot-reuse only when it is expected to be beneficial. We instantiate this approach for WebAssembly and implement it in a new concolic-execution compiler GENWASYM. Across 184 benchmarks, GENWASYM with staging alone achieves a 29.4× average speedup over the interpreter-based WASP; heuristic snapshot-reuse further increases the speedup to 44.9×.

CCS Concepts: • **Software and its engineering** → **Source code generation; Software testing and debugging**; • **Theory of computation** → **Program semantics**.

Additional Key Words and Phrases: Concolic execution, Continuation, WebAssembly, Semantics, Staging, Performance

ACM Reference Format:

Dinghong Zhong, Alexander Y. Bai, Mikail Khan, and Guannan Wei. 2026. Compiling WebAssembly Concolic Execution with Staging, Continuations, and Snapshots. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 366 (October 2026), 28 pages. <https://doi.org/10.1145/3839498>

1 Introduction

As a white-box testing technique, modern symbolic execution (SE) [11] automates test generation and bug finding by partitioning the input space of programs and utilizing the power of SMT solvers [6]. By treating some inputs as symbolic values [8, 12, 27, 30], symbolic execution calculates the conditions of program paths symbolically. These conditions represent equivalence classes of inputs that can trigger the execution of specific paths. General nondeterministic symbolic execution

Authors' Contact Information: Dinghong Zhong, Tufts University, Medford, USA, dinghong.zhong@tufts.edu; Alexander Y. Bai, New York University, New York, USA, alexander.bai@nyu.edu; Mikail Khan, Carnegie Mellon University, Pittsburgh, USA, mikaillk@cmu.edu; Guannan Wei, Tufts University, Medford, USA, guannan.wei@tufts.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/10-ART366

<https://doi.org/10.1145/3839498>

(e.g., KLEE [10], GenSym [56]) forks the execution and explores multiple paths simultaneously. However, this approach requires frequent path feasibility checks and quickly leads to path explosion.

Concolic execution [22, 23, 45] alleviates solver overhead and path explosion by constraining itself to run only one path at a time, guided by a concrete input. Since path feasibility is evidenced by that concrete input, it suffices to collect the corresponding path condition without querying solvers when running the program. In this way, concolic execution mixes **concrete** and **symbolic** execution. After executing one path, new inputs are generated by negating parts of the collected path condition and solving satisfiability for the new concrete inputs. These new inputs lead the engine to explore new paths in the program. A driver orchestrates this process iteratively to systematically explore the program space. In practice, concolic execution is often more effective for bug finding and is the building block of widely adopted hybrid fuzzing techniques [34, 48, 60], as concrete inputs can drive execution deep into the program.

Interpretation or Instrumentation? Concolic execution can be viewed as a specific “path scheduling strategy”, where the path to be explored is biased by the concrete inputs provided in each iteration. However, in each iteration with a new input, concolic execution needs to rerun the program from the beginning (i.e., the root of the symbolic execution tree), both concretely and symbolically. Although a prefix of the execution is shared between the paths explored by the new and old inputs, the corresponding path conditions for the new path must still be re-collected through unnecessary re-execution. This redundant re-execution hurts performance and throughput of concolic execution, especially when the negation point lies deep in the execution tree.

Indeed, re-execution can be avoided by reusing the existing symbolic state from the negation point, an optimization known as *snapshot reuse* (e.g., Driller [48]). It has been observed that Driller with snapshot reuse achieves more coverage in 76 out of 126 benchmarks from DARPA’s Cyber Grand Challenge [60]. Realizing snapshot reuse requires copying the program state and saving it at the corresponding branching point, so that in the next iteration, execution can directly jump back to that point and continue the alternative branch. Due to these requirements, implementing snapshot reuse is most straightforward within interpretation-based concolic execution, where the engine has full control over the execution state and control flow at the meta level. However, copying the whole program state can be expensive [60], and interpretation imposes significant overhead [29], such as traversing and dispatching over the program’s static AST/IR. These overheads make interpretation a less ideal choice for performance-critical concolic execution, even with snapshot reuse.

To achieve high performance, state-of-the-art concolic execution engines (e.g., QSYM [60] for binaries, SymCC [38] for LLVM IR, and SymQEMU [39] for VEX IR) rely on source-to-source *instrumentation* to completely avoid interpretation overhead. Since concolic execution explores only one path at a time, instrumentation needs only to insert symbolic instructions that collect path conditions alongside the control flow of the original concrete execution. However, these instrumentation-based engines abandon snapshot reuse at all. It is no coincidence: with instrumentation that shallowly inserts additional instructions, the program itself cannot save its current control at a branching point¹ and later resume it in the next iteration. Therefore, it is challenging to implement snapshot reuse entirely within instrumentation-based concolic execution.

Best of Two Worlds. Although instrumentation enables fast execution, snapshot reuse remains a valuable optimization, especially when branch points occur deep in the execution. To maximize throughput, these two techniques should be combined. Existing implementation strategies, however, force concolic-execution engine designers to choose between them: snapshot reuse is relatively

¹To be precise, it would require the language to have mechanisms such as `call/cc` to capture the current continuation, which is not the case here for typical targets of concolic execution.

straightforward to implement in an interpreter, but incurs substantial interpretation overhead; instrumentation eliminates that overhead, but makes snapshot reuse difficult to realize.

In this paper, we resolve this tension by showing how to obtain instrumentation-like performance while still supporting effective snapshot reuse. Our key idea is to partially evaluate [29] a concolic interpreter written in continuation-passing style (CPS) [4, 42] using staging [43, 50, 51].

- By staging a concolic interpreter, we specialize it with respect to the input program and thereby generate efficient concolic-execution code that no longer suffers from interpretation overhead, achieving instrumentation-like performance. In effect, our approach realizes compilation via the first Futamura projection [19, 20] for a concolic interpreter. At the same time, constructing a concolic interpreter is arguably simpler than building a compiler, so our approach reduces the implementation effort required for high-performance engines.
- By writing the staged concolic interpreter in CPS, we make control flow explicit and manipulable as first-class functions in the generated code, which enables snapshot reuse. At each branch in the source program, we generate code that stores the current state and continuation as a snapshot. When execution later resumes from that branch point, the engine invokes the saved continuation for the alternative branch, completely avoiding re-execution from the program entry to that point.

Applying staging or partial evaluation to program analysis is not new [2, 7, 14, 54–56], nor is the use of continuations in symbolic execution engines [56]. Our work is inspired in part by GenSym’s metaprogramming approach [56], which shows that, for nondeterministic symbolic execution of LLVM IR, continuations can be used to implement search heuristics. We adopt the idea of using continuations to manipulate control flow at the meta level. To the best of our knowledge, however, this is the first work to combine staging and continuations in a concolic execution engine to achieve both high execution performance and snapshot reuse, unifying two optimizations that have previously been deployed only separately in practice.

That said, the effectiveness of snapshot reuse still depends on the target program. When symbolic states are large and execution paths are shallow, saving and restoring snapshots can cost more than simply re-executing from the beginning, since the avoided re-execution may be minimal. To address this trade-off, we propose a heuristic that decides when to use snapshots based on execution depth and symbolic-state complexity (Section 6). Our heuristic also avoids using snapshots for external functions, which may have side effects depending on the concrete inputs. With this heuristic, our approach smoothly combines the best of both worlds: *efficient execution through compilation and adaptive snapshot reuse via continuations when it is likely to pay off*.

Compiled Concolic Execution with Snapshots for WebAssembly. To demonstrate and validate our approach, we use WebAssembly (Wasm) [24] as the target language. As a general-purpose, low-level bytecode language, WebAssembly has emerged as an important target for both compilation and program analysis. WebAssembly is now supported by all major browsers and has also been adopted beyond the browser setting.

Despite WebAssembly’s growing importance, existing concolic or symbolic execution tools for Wasm remain interpreter-based, limiting their performance and scalability. To the best of our knowledge, there is no instrumentation- or compilation-based concolic execution engine for Wasm. Therefore, our focus on Wasm is motivated both by its growing importance as a deployment and analysis target and by the current lack of high-performance concolic execution tools for it.

We present our development through a sequence of mechanical derivation steps, each obtained from the previous one. We begin with a concrete continuation-passing semantics for WebAssembly (Section 2), formulated as a definitional interpreter [41] adapted from [53]. On top of this foundation,

we derive a concolic evaluator in CPS (Section 3), refactor it into a staged concolic interpreter in CPS (Section 4), and finally show how snapshots can be reused in the generated code (Section 5).

Although this paper focuses on WebAssembly as the target language, our approach is not specific to WebAssembly and readily extends to other languages or intermediate representations (Section 9). Our mechanical, derivational methodology helps ensure that each step is correct by construction and makes the overall approach easier to transfer to other languages.

Implementation and Evaluation. We implement our architecture in GENWASYM, a concolic-execution compiler for Wasm written in Scala using LMS [43]. Given a Wasm program, GENWASYM generates a C++ implementation following our snapshot-reusing concolic semantics. The generated program links against a scheduler that invokes the SMT solver to generate new inputs and applies our heuristic (Section 6), which chooses between re-execution and applying snapshots. To the best of our knowledge, GENWASYM is the first concolic-execution compiler for Wasm and the first Wasm concolic engine to support snapshot reuse.

We evaluate GENWASYM against WASP [35], the prior state-of-the-art concolic execution engine for WebAssembly. WASP suffers from two performance limitations due to its design as an interpreter-based engine. First, it is built on the reference small-step reduction semantics [24], which incurs substantial interpretation cost when resolving control flow, such as jumps, through administrative instructions [52]. Second, it does not support snapshot reuse, leading to redundant re-execution. Our work addresses both limitations.

We evaluate GENWASYM on the B-Tree benchmark suite and the Collections-C benchmark suite [47] used by WASP [35]. Across 184 benchmarks, GENWASYM with staging alone achieves an average (geomean) speedup of 29.4×. Further enabling heuristic snapshot reuse yields an average speedup of 44.9×. The results show that staging uniformly eliminates the majority overhead from interpretation, yielding orders-of-magnitude speedups, while on top of that snapshot reuse provides significant additional improvement for programs with deep execution paths.

Contributions and Organization. In this paper, we develop the first concolic-execution architecture to combine high-performance execution with snapshot reuse, enabled by the key insight of using staging with continuations in the concolic-execution engine. We demonstrate our approach with Wasm, implement the snapshot-reusing concolic-execution compiler GENWASYM, and evaluate its performance through extensive experiments. The paper is organized as follows:

- Section 2 briefly reviews concolic execution, Wasm, and Wasm’s concrete CPS semantics.
- Section 3 develops a definitional concolic interpreter based on the concrete CPS semantics. Section 4 derives the staged concolic interpreter by refactoring the single-stage concolic interpreter.
- Section 5 explains how to implement snapshot reuse (*i.e.*, avoiding re-execution) in the generated concolic execution code by manipulating continuations. Section 6 proposes a heuristic to decide when to reuse snapshots to balance the benefit and cost of snapshot resumption.
- Section 7 describes our implementation of GENWASYM, which is implemented in Scala and generates C++ code for concolic execution of WebAssembly programs.
- Section 8 presents the performance evaluation of GENWASYM against WASP [35], showing an average 29.4× speedup. We also evaluate the effectiveness of our snapshot heuristic.
- Section 9 compares our approach with instrumentation and discusses its applicability to other languages; Section 10 reviews related work; and Section 11 concludes the paper.

2 Background

In this section, we provide background on concolic execution, WebAssembly, and its CPS semantics [53] that we build upon.

```

int x = input();
int y = input();
if (x <= 0) return 0;
else if (y <= 0) return 0;
else if (x * x + y * y == 25)
    return assert(false);
else return 0;

```

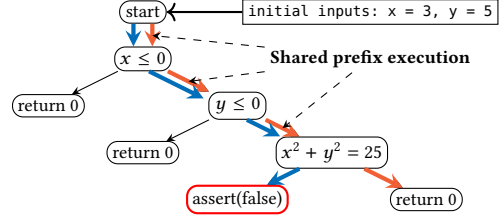


Fig. 1. An example program (left) and its concolic execution tree (right). The orange execution path is traced by initial inputs, while the blue path represents another execution, sharing a prefix with the orange path.

$$\begin{aligned}
 \ell \in \text{Label} &= \mathbb{N} & x \in \text{Identifier} &= \mathbb{N} \\
 t \in \text{ValueType} &::= \text{i32} \mid \text{i64} \mid \dots & ft \in \text{FunctionType} &::= t^* \rightarrow t^* \\
 e \in \text{Instruction} &::= \text{nop} \mid t.\text{const } c \mid t.\{\text{add, sub, eq, } \dots\} \mid \text{local.get } x \mid \text{local.set } x \\
 &\mid \text{block } ft \text{ es} \mid \text{loop } ft \text{ es} \mid \text{if } ft \text{ es es} \mid \text{br } \ell \mid \text{call } x \mid \text{return} \\
 es \in \text{Instructions} &= \text{List}[\text{Instruction}] & f \in \text{Function} &::= \text{func } x \{ \text{type} : ft, \text{ locals} : t^*, \text{ body} : es \} \\
 m \in \text{Module} &::= \text{module } f^*
 \end{aligned}$$

Fig. 2. The abstract syntax of μWasm .

2.1 Concolic Execution

Concolic execution [5, 22, 23, 45] runs program p with concrete input i , traces an execution path π , and collects its path condition cond . Solving negated path condition $\neg \text{cond}$ with an SMT solver generates a new input i' that explores another path. This process repeats until all paths are explored or a limit (e.g., time or coverage) is reached.

Consider the example program and its execution tree shown in Figure 1. Suppose we start concolic execution with initial inputs $x = 3$ and $y = 5$. The execution path in orange collects the path condition $x > 0 \wedge y > 0 \wedge x^2 + y^2 \neq 25$ along the way. To explore a new path, concolic execution negates the condition, e.g., $x > 0 \wedge y > 0 \wedge x^2 + y^2 = 25$ where the last condition is negated, and queries the SMT solver for a new input satisfying the negated condition. The solver returns a new input $x = 4$ and $y = 3$, which explores a new path shown in blue in Figure 1. However, to reach the error node in the blue path, the concolic execution engine needs to re-execute part of the program, which was already executed in the last iteration. If the underlying concolic execution engine can save the snapshot at branch points, it can reuse those symbolic states to avoid re-execution.

The topic of this paper is to develop an approach to concolic execution that is easy to implement, efficient to execute, and able to reuse symbolic states to avoid re-execution.

2.2 WebAssembly

We demonstrate our approach on WebAssembly, an emerging portable stack-based virtual instruction set for safe and efficient execution on the web and beyond. In the formalization, we limit our discussion to μWasm , a subset of WebAssembly that captures the core features relevant to concolic execution. Readers may refer to the official specification [1] for more details about WebAssembly. Figure 2 shows the syntax of μWasm , and Figure 3 shows its concrete semantics. μWasm models the following instructions covering the essential part of local variables, control flow, and function calls in WebAssembly:

- We model a range of arithmetic instructions. For example, $t.\text{const } c$ pushes a constant value c of type t onto the stack, and $t.\text{add}$ pops two operands of type t from the stack, then pushes the addition result back to the stack. Instruction $\text{local.get } x$ pushes the value of variable x onto the stack, and $\text{local.set } x$ sets the value of variable x using the top value on the stack.

- Unlike most other low-level IRs, WebAssembly supports structured control flow with block, loop, and if instructions, containing sequences of instructions as their bodies. Since block-like instructions can be nested, WebAssembly uses the `br l` instruction to transfer control-flow to the target block, where label l is the de Bruijn index identifying the target block. Block-like instructions are annotated with a function type $t^m \rightarrow t^n$, indicating the shape of the stack before and after the execution of the block.
- A top-level program consists of a module of functions. Each function has an identifier, a type, a list of local variables, and a sequence of instructions as its body. The instruction `call x` transfers the control-flow to the callee, and `return` transfers the control-flow back to the caller.

We omit typing of the language, which is standard and can be found in the official WebAssembly specification [1]. We also omit Wasm features such as tables, global variables, and memory, which are not essential to explain our approach. However, our implementation in Section 7 supports a larger subset of WebAssembly and can handle more realistic programs.

2.3 Continuation-Passing Semantics of μ Wasm

Why Continuations? The official WebAssembly specification [1] defines a small-step reduction semantics. However, the semantics has to introduce administrative instructions to manage control flow; these instructions represent evaluation contexts and are materialized only at runtime. This obscures the binding-time distinction between parts of the program that are known at compile time and others at runtime, rendering it difficult to construct a staged interpreter on top of it.

In this paper, we adopt a continuation-passing semantics [53] for WebAssembly that is defined in a compositional way using continuation functions to represent control flow. It is well-known that continuations improve binding-time separation, allowing more computation to be evaluated at compile time [13, 26, 32]. We later exploit this insight and build the staged concolic interpreter on top of the CPS semantics [53]. Moreover, continuations naturally model the snapshots of symbolic states needed for concolic execution, as we will discuss in Section 5.

The correctness of the CPS semantics could be established by relating it to the small-step semantics via defunctionalization [15] and refunctionalization [16, 21]. A detailed investigation of this connection is out of the scope of this work.

CPS Semantics of μ Wasm. Figure 3 shows μ Wasm's concrete CPS semantics [53]. Values are represented as integers, and stack and environment are both lists of values. We call the current stack and environment the program *state*. Continuations κ are represented as functions from states to the final result *Ans*. The *Ans* type is deliberately left abstract and can be instantiated with different types, enabling the semantics to serve different purposes.

The semantics function $\mathbb{E}[\![\cdot]\!]$ is recursively defined over the input list of instructions. Given a list of instructions es , $\mathbb{E}[\![es]\!]$ returns a function taking a stack, environment, continuation, and a trail, and produces *Ans*. If the list is empty, we simply invoke the continuation. If the list is not empty, we pattern-match on the first instruction and define its semantics accordingly, and recursively call $\mathbb{E}[\![rest]\!]$ when necessary. The trail is represented as a list of continuations and serves as a table, modeling block jumps local to a function, which we will explain later with control flow.

In our semantics, we pass the updated state explicitly to recursive calls to $\mathbb{E}[\![\cdot]\!]$ or continuations rather than updating them in place. The `nop` instruction leaves the state unchanged, while `const c` pushes c onto the stack. `local.get x` loads the value at index x in the environment onto the stack, while `local.set x` pops the stack and writes the popped value at index x in the environment. For brevity, several rules pattern-match on the stack. For example, the $t.add$ rule matches $v_1 :: v_2 :: \sigma$, where v_1 and v_2 are the top two stack values and σ is the remainder. We write $\sigma_1 \#_m \sigma_2$ for a stack split into a top segment σ_1 of length m and a remainder σ_2 , and $[\sigma]_n$ for the top n values of σ .

Semantic Domains

$$v \in \text{Value} = \mathbb{Z}$$

$$\rho \in \text{Env} = \text{List}[\text{Value}]$$

$$\sigma \in \text{Stack} = \text{List}[\text{Value}]$$

$$\kappa \in \text{Cont} = \text{Stack} \times \text{Env} \rightarrow \text{Ans}$$

Concrete Continuation Semantics

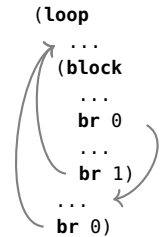
$$\mathbb{E}[\cdot] : \text{List}[\text{Inst}] \rightarrow (\text{Stack} \times \text{Env} \times \text{Cont} \times \text{List}[\text{Cont}]) \rightarrow \text{Ans}$$

$$\begin{aligned} \mathbb{E}[\text{nil}] (\sigma, \rho, \kappa, \theta) &= \kappa(\sigma, \rho) \\ \mathbb{E}[\text{nop} :: \text{rest}] (\sigma, \rho, \kappa, \theta) &= \mathbb{E}[\text{rest}] (\sigma, \rho, \kappa, \theta) \\ \mathbb{E}[\text{t.const } c :: \text{rest}] (\sigma, \rho, \kappa, \theta) &= \mathbb{E}[\text{rest}] (c :: \sigma, \rho, \kappa, \theta) \\ \mathbb{E}[\text{t.add} :: \text{rest}] (v_1 :: v_2 :: \sigma, \rho, \kappa, \theta) &= \mathbb{E}[\text{rest}] (v_1 + v_2 :: \sigma, \rho, \kappa, \theta) \\ \mathbb{E}[\text{local.get } x :: \text{rest}] (\sigma, \rho, \kappa, \theta) &= \mathbb{E}[\text{rest}] (\rho(x) :: \sigma, \rho, \kappa, \theta) \\ \mathbb{E}[\text{local.set } x :: \text{rest}] (v :: \sigma, \rho, \kappa, \theta) &= \mathbb{E}[\text{rest}] (\sigma, \rho[x \mapsto v], \kappa, \theta) \\ \mathbb{E}[\text{block } (t^m \rightarrow t^n) \text{ es} :: \text{rest}] (\sigma_{arg\ m} \# \sigma, \rho, \kappa, \theta) &= \\ \quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1). \mathbb{E}[\text{rest}] (\lfloor \sigma_1 \rfloor_n \# \sigma, \rho_1, \kappa, \theta) \text{ in } \mathbb{E}[\text{es}] (\sigma_{arg}, \rho, \kappa_1, \kappa_1 :: \theta) \\ \mathbb{E}[\text{loop } (t^m \rightarrow t^n) \text{ es} :: \text{rest}] (\sigma_{arg\ m} \# \sigma, \rho, \kappa, \theta) &= \\ \quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1). \mathbb{E}[\text{rest}] (\lfloor \sigma_1 \rfloor_n \# \sigma, \rho_1, \kappa, \theta) \text{ in} \\ \quad \text{fix } \kappa_2 := \lambda(\sigma_2, \rho_2). \mathbb{E}[\text{es}] (\lfloor \sigma_2 \rfloor_m, \rho_2, \kappa_1, \kappa_2 :: \theta) \text{ in } \kappa_2(\sigma_{arg}, \rho) \\ \mathbb{E}[\text{if } (t^m \rightarrow t^n) \text{ es}_1 \text{ es}_2 :: \text{rest}] (v :: \sigma_{arg\ m} \# \sigma, \rho, \kappa, \theta) &= \\ \quad \text{let } es := \text{if } v \equiv 0 \text{ then } es_2 \text{ else } es_1 \text{ in} \\ \quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1). \mathbb{E}[\text{rest}] (\lfloor \sigma_1 \rfloor_n \# \sigma, \rho_1, \kappa, \theta) \text{ in } \mathbb{E}[\text{es}] (\sigma_{arg}, \rho, \kappa_1, \kappa_1 :: \theta) \\ \mathbb{E}[\text{br } \ell :: \text{rest}] (\sigma, \rho, \kappa, \theta) &= \theta(\ell)(\sigma, \rho) \\ \mathbb{E}[\text{call } x :: \text{rest}] (\sigma_{arg\ m} \# \sigma, \rho, \kappa, \theta) &= \\ \quad \text{let } \{\text{type} : t^m \rightarrow t^n, \text{locals} : ts, \text{body} : es\} := \text{lookupFunc}(x) \text{ in} \\ \quad \text{let } \rho_1 := \text{buildEnv}(\sigma_{arg}, ts) \text{ in} \\ \quad \text{let } \kappa_1 := \lambda(\sigma_1, \rho_1). \mathbb{E}[\text{rest}] (\lfloor \sigma_1 \rfloor_n \# \sigma, \rho, \kappa, \theta) \text{ in } \mathbb{E}[\text{es}] ([], \rho_1, \kappa_1, [\kappa_1]) \\ \mathbb{E}[\text{return} :: \text{rest}] (\sigma, \rho, \kappa, \theta) &= \theta.\text{last}(\sigma, \rho) \end{aligned}$$

Fig. 3. The concrete execution semantics of μWasm [53].

Control Flow. Now we go through several control-flow instructions to illustrate how continuations and trail are used. The control-flow instructions `block`, `loop`, and `if` add a new continuation to the control context θ , which defines the behavior of branching instruction `br`. The `block` instruction evaluates the body instructions es with an updated control context that adds a new continuation κ_1 to the trail θ . The continuation κ_1 captures the control flow after the `block`.

To enter this continuation via `br l` instruction, we look up the l -th continuation in the trail θ , and the control flow will jump to the corresponding continuation. On the right, the first `br 0` jumps to the end of the innermost block, which is captured by the continuation at the top of the trail *within* the scope of the block. The `loop` instruction is similar to `block`, but the continuation κ_2 added to the trail θ will re-evaluate the loop body es when branched to via `br l`. This is exemplified by the `br 0` at the end of the example. The `if` instruction first pops the value v from the stack. Then when v is not zero, it recursively evaluates the instructions es_1 with the continuation κ_1 pushes onto trail θ , where κ_1 captures the control flow after the true branch. Otherwise it evaluates es_2 in the false-branch with the same control context.



Function Call and Return. The `call x` instruction retrieves x 's definition using `lookupFunc` and uses `buildEnv` to construct ρ_1 from the argument list σ_{arg} , allocating extra space for local variables.

Semantic Domains

$$\begin{aligned}
 op_2^\# &\in \{+, -, \dots\} & op_1^\# &\in \{-^\#, \dots\} & \sigma^\# &\in \text{Stack}^\# = \text{List}[\text{Value} \times \text{Symbolic}] \\
 x &\in \text{SymVar} & & := \text{Identifier} & \rho^\# &\in \text{Env}^\# = \text{List}[\text{Value} \times \text{Symbolic}] \\
 s &\in \text{Symbolic} & := v \mid \text{Sym}(x) \mid s_1 \text{ } op_2^\# \text{ } s_2 \mid op_1^\# \text{ } s_1 & \kappa^\# &\in \text{Cont}^\# = (\text{Stack}^\# \times \text{Env}^\# \times \text{PC}^\#) \rightarrow \text{Ans} \\
 \pi^\# &\in \text{PC}^\# & = \text{List}[\text{Symbolic}] & \theta^\# &\in \text{Trail}^\# = \text{List}[\text{Cont}^\#]
 \end{aligned}$$

Concolic Continuation Semantics

$$\begin{aligned}
 \mathbb{C}[\cdot] &: \text{List}[\text{Inst}] \rightarrow (\text{Stack}^\# \times \text{Env}^\# \times \text{PC}^\# \times \text{Trail}^\# \times \text{Cont}^\#) \rightarrow \text{Ans} \\
 \mathbb{C}[t.\text{const } c :: \text{rest}] &(\sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\#) = \mathbb{C}[\text{rest}](\langle \langle c, c \rangle :: \sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\# \rangle) \\
 \mathbb{C}[t.\text{add} :: \text{rest}] &(\langle \langle v_1, s_1 \rangle :: \langle v_2, s_2 \rangle :: \sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\# \rangle) = \mathbb{C}[\text{rest}](\langle \langle v_1 + v_2, s_1 +^\# s_2 \rangle :: \sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\# \rangle) \\
 \mathbb{C}[\text{local.get } x :: \text{rest}] &(\sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\#) = \mathbb{C}[\text{rest}](\langle \rho^\#(x) :: \sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\# \rangle) \\
 \mathbb{C}[\text{local.set } x :: \text{rest}] &(\langle \langle v, s \rangle :: \sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\# \rangle) = \mathbb{C}[\text{rest}](\langle \sigma^\#, \rho^\#[x \mapsto \langle v, s \rangle], \pi^\#, \kappa^\#, \theta^\# \rangle) \\
 \mathbb{C}[\text{if } (t^m \rightarrow t^n) \text{ } es_1 \text{ } es_2 :: \text{rest}] &(\langle \langle v, s \rangle :: \sigma_{\text{arg } m}^\# \text{ } \sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\# \rangle) = \\
 &\text{let } es := \text{if } v \equiv 0 \text{ then } es_2 \text{ else } es_1 \text{ in} \\
 &\text{let } \pi_1^\# := \text{if } v \equiv 0 \text{ then } \neg s :: \pi^\# \text{ else } s :: \pi^\# \text{ in} \\
 &\text{let } \kappa_1^\# := \lambda(\sigma_1^\#, \rho_1^\#, \pi_1^\#). \mathbb{C}[\text{rest}](\lfloor \sigma_1^\# \rfloor_n \text{ } \sigma^\#, \rho_1^\#, \pi_1^\#, \kappa^\#, \theta^\#) \text{ in} \\
 &\mathbb{C}[es](\sigma_{\text{arg}}^\#, \rho^\#, \pi_1^\#, \kappa_1^\#, \kappa_1^\# :: \theta^\#) \\
 \mathbb{C}[\text{br } \ell :: \text{rest}] &(\sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\#) = \theta^\#(\ell)(\sigma^\#, \rho^\#, \pi^\#)
 \end{aligned}$$

Fig. 4. The concolic semantics of μWasm (excerpted). The full semantics is in the extended version [62].

Finally, the call x instruction evaluates the function body es under an empty stack, ρ_1 and a control context initialized with continuation κ_1 . This control context corresponds to the target function's scope and is a singleton list containing κ_1 , which captures the control flow after the function call. In the function body, the return instruction invokes the last trail continuation κ_1 , skipping all enclosing blocks. Readers can refer to [53] for more details on Wasm's CPS semantics.

3 Concolic Execution with Continuations

The concrete CPS semantics of μWasm (Section 2.3) forms the basis for developing our compositional concolic execution semantics, which we present in this section. When presenting formalization, we highlight the symbolic extensions relative to the concrete semantics.

3.1 Concrete-Input-Controlled Symbolic Execution

We first extend the semantic domains with symbolic counterparts and then define the concolic semantics as a definitional interpreter $\mathbb{C}[\cdot]$. The new symbolic components collect path conditions during execution without affecting the program's concrete behavior, so the concolic semantics subsumes the concrete one. At the same time, it remains compositional and uses continuations to make control flow explicit.

Symbolic Semantic Domains. Figure 4 highlights the extended semantic domains. Symbolic values s are a superset of concrete values v , additionally including symbols $\text{Sym}(x)$ identified by x , and unary ($op_1^\# s_1$) and binary ($s_1 \text{ } op_2^\# \text{ } s_2$) symbolic expressions.

We use a single stack and environment to store both concrete and symbolic values. Thus, the symbolic stack $\text{Stack}^\#$ and environment $\text{Env}^\#$ are lists of pairs $\langle v, s \rangle$, where v is a concrete value and s is its corresponding symbolic expression under the current path condition. Path conditions PC

are represented as lists of symbolic values, with newly encountered conditions pushed to the front. The conjunction of all expressions in PC is the constraints on symbolic inputs to trigger the current execution path. Finally, continuation functions $\text{Cont}^\#$ take a concolic stack, environment, and path conditions as input. The return type Ans is left polymorphic to allow different instantiations.

Definitional Concolic Interpreter. Figure 4 shows an excerpt of the concolic semantics as a definitional interpreter. Compared to the concrete semantics (Figure 3), the concolic semantics $\mathbb{C}[\![\cdot]\!]$ is extended to take path conditions $\pi^\#$ as additional inputs, and other components are lifted to their concolic counterparts. Most of the changes to the interpreter are structural: the control flow is unchanged and we simply pass around the symbolic components, therefore the cases for *nil*, *block*, *loop*, *call*, and *return* are omitted and can be found in the appendix of the extended version of this paper [62].

We explain concolic execution through representative cases. The symbolic interpretation of numeric operations mirrors their concrete construction. For example, $t.\text{const } c$ creates a concrete-symbolic pair $\langle c, c \rangle$ and pushes it onto the concolic stack, while $t.\text{add}$ builds the corresponding symbolic addition expression. We collect path conditions when encountering conditional instructions. In the $\mathbb{C}[\![\text{if } \dots]\!]$ case, we first pop the concolic condition $\langle v, s \rangle$ from the stack. The concrete value v determines the branch to take, as in the concrete semantics, while the symbolic value s updates the path condition: depending on the branch, s or $\neg s$ is prepended to $\pi^\#$, yielding a new path condition $\pi_1^\#$ that is passed to the evaluation of the chosen branch.

Initial Continuation. To invoke the concolic definitional interpreter, we provide the initial halting continuation $\kappa_{\text{halt}}^\#$. The interpreter's return type Ans is polymorphic over different instantiations of this continuation. For concolic execution, we instantiate it to return the collected path conditions upon completion $\kappa_{\text{halt}}^\#(_, _, \pi^\#) = \pi^\#$. The concolic execution scheduler (described below) uses the returned path conditions to generate new inputs and explore alternative paths.

3.2 Concolic Execution Scheduler

Given an input Wasm program, the concolic scheduler (Algorithm 1) orchestrates the exploration of paths by invoking $\mathbb{C}[\![\cdot]\!]$ with varying inputs that lead to distinct paths. The scheduler takes as input a module, the entry function name f , and a list of initial arguments, which may include both symbolic (*i.e.*, $\text{Sym}(x)$) and concrete values. It maintains a worklist $WL \in \mathcal{P}(\text{PC}^\#)$ as a set of path conditions to explore. Each path condition represents constraints on symbolic inputs leading to an unexplored path if satisfied. Initially, the worklist contains only the empty path condition, since no paths have yet been explored and any input is valid.

In each iteration, the scheduler pops a path condition pc from the worklist and invokes an SMT solver to find a satisfying assignment. We model the solver as a function $\text{solve} : \text{PC} \rightarrow \text{Model} \cup \text{UNSAT}$, where a model is a partial mapping from symbolic variables to concrete values:

$$m \in \text{Model} = \text{SymVar} \rightarrow \text{Value}.$$

If pc is unsatisfiable, the scheduler skips the path as there is no possible input to reach this path. Otherwise, to construct the initial concolic input stack for executing $\mathbb{C}[\![\text{call } f]\!]$, we use an auxiliary function $\text{instantiate} : (\text{List}[\text{Symbolic}], \text{Model}) \rightarrow \text{List}[\text{Value} \times \text{Symbolic}]$ that evaluates each symbolic input under the model and pairs the resulting concrete value with the original symbolic input. The scheduler then invokes $\mathbb{C}[\![\text{call } f]\!]$ with the constructed concolic stack, empty environment, initial continuation trail $[\kappa_{\text{halt}}^\#]$, and halting continuation $\kappa_{\text{halt}}^\#$ to explore a new path.

Upon completion, the interpreter returns the collected path condition π_1 . The scheduler then generates new path conditions by negating (as detailed in [62]) each constraint in π_1 and adds the resulting conditions to the worklist, and proceeds to the next iteration. When the worklist becomes empty, all feasible execution paths have been covered.

Algorithm 1 Concolic Scheduler

```

1: function SCHEDULER( $mod \in \text{Module}, f \in \text{Identifier}, input \in \text{List}[\text{Symbolic}]$ )
2:    $WL \leftarrow \{ [] \}$   $\triangleright WL \in \mathcal{P}(\text{PC}^\#)$ 
3:    $Seen \leftarrow \emptyset$   $\triangleright Seen \in \mathcal{P}(\text{PC}^\#)$ 
4:   while  $WL \neq \emptyset$  do
5:     pop  $\pi$  from  $WL$ 
6:     if  $\pi \in Seen$  then continue
7:      $m \leftarrow \text{solve}(\pi)$   $\triangleright$  invoke SMT solver
8:     if  $m = \text{UNSAT}$  then continue  $\triangleright$  unreachable path condition
9:      $\pi_1 \leftarrow \mathbb{C} \llbracket \text{call } f \rrbracket (\text{instantiate}(input, m), [], [\kappa_{halt}^\#, \kappa_{halt}^\#])$   $\triangleright$  run concolic semantics
10:     $Seen \leftarrow Seen \cup \{\pi_1\}$ 
11:    for all  $\pi' \in \text{NEGATE}(\pi_1)$  do
12:       $WL \leftarrow WL \cup \{\pi'\}$ 
13: function NEGATE( $\pi \in \text{PC}^\#$ )  $\rightarrow \text{List}[\text{PC}^\#]$  ...  $\triangleright$  negate each constraint in  $\pi$  (extended version of the paper)

```

$\text{fix } power : (\text{Int}, \text{Int}) \rightarrow \text{Int}$	$\coloneqq \lambda(x, n). \text{if } n \equiv 0 \text{ then } 1 \text{ else } x * power(x, n - 1)$
$\text{fix } genpower : (\text{Expr}[\text{Int}], \text{Int}) \rightarrow \text{Expr}[\text{Int}]$	$\coloneqq \lambda(x, n). \text{if } n \equiv 0 \text{ then '1}$ $\text{else '}\{ \\$x * \\$\{ genpower(x, n - 1) \} \}$
$\text{let } power2 : \text{Int} \rightarrow \text{Int}$	$\coloneqq \text{run '}\{ \lambda(x). \$\{ genpower('x, 2) \} \}$

Fig. 5. The example of unstaged power function, its staged version, and specialized power wrt $n = 2$.

4 Compiling Concolic Execution with Continuations

The concolic semantics developed in Section 3 can already be implemented as a definitional interpreter. Although conceptually straightforward, this interpreter incurs significant overhead from repeatedly dispatching on instructions, which can greatly impact performance [33, 57, 58]. In this section, we build a concolic-execution compiler to eliminate this overhead by realizing the first Futamura projection [19, 20] through multi-stage programming (MSP) [50]. We refactor the concolic semantics into a compiler that generates efficient concolic execution code without interpretation overhead. Section 4.1 briefly introduces MSP and the notations used later, and Section 4.2 presents the staged concolic semantics for μWasm .

4.1 A Brief Overview of Multi-Stage Programming

Multi-stage programming is a generative programming paradigm that enables programs to construct and manipulate other programs. By dividing computation into multiple stages, programmers annotate code to indicate which parts depend on static input known at earlier stages and which depend on dynamic input available only later. The resulting residual program contains only computations that depend on dynamic input, while computations over static input have been performed in earlier stages. Writing an interpreter in an MSP language allows us to specialize it to the static program representation, effectively achieving programmer-guided partial evaluation [29].

Later, in Section 4.2, we specify the staged concolic semantics using the quasi-quotation and splicing syntax from Scala 3 [49]. A quotation $\text{'}\{ e \}$ (or simply $\text{'}e$) denotes a next-stage expression e , while splicing $\text{'}\{ \dots \$\{ e \} \dots \}$ inserts the value of a current-stage expression e into the surrounding quotation. At the type level, $\text{Expr}[T]$ represents a next-stage type T .

Figure 5 illustrates MSP with the power function example. The single-stage function $power$ computes x^n recursively. When the exponent n is known in advance, we can write a multi-stage version $genpower$, which generates a specialized $power$ function taking only x as input. In $genpower$, the variable x is marked as a next-stage value by type $\text{Expr}[\text{Int}]$, and the result is a next-stage

Staged Module Semantics

$$\begin{aligned}
\mathbb{M}[\![\cdot]\!] &: \text{List}[\text{Function}] \rightarrow \text{Expr}[\text{Memo}] \\
\mathbb{M}[\![\text{nil}]\!] &= \emptyset \\
\mathbb{M}[\![\{\text{type} : t^m \rightarrow t^n, \text{id} : x, \text{locals} : ts, \text{body} : es\} :: \text{rest}]\!] &= \\
&\quad ' \{ \text{let } f := \lambda(\rho^\#, \pi^\#, \kappa^\#, m_1). \$ \{ \mathbb{S}[\![es]\!]('m_1)(\text{lift}([]), ' \rho^\#, ' \pi^\#, ' \kappa^\#, [' \kappa^\#]) \} \text{ in} \\
&\quad ' \{ \mathbb{M}[\![\text{rest}]\!] \} [\$ \text{lift}(x) \mapsto f] \}
\end{aligned}$$

Fig. 6. The staged module semantics of μWasm

expression that computes x^n . When $n = 0$, *genpower* produces the staged constant '1; when $n > 0$, it constructs a staged expression multiplying x by the result of a recursive call with $n - 1$.

By providing a concrete value for the static input n , we can specialize *genpower* and produce a residual program that depends only on x . For instance, we specialize *genpower* to obtain *power2* by applying *run*, a common construct for evaluating staged expressions in MSP languages [31, 49, 51]. Compared to the generic *power* function, *power2* for $n = 2$ unrolls all recursive calls and eliminates the overhead of condition checks.

4.2 Compilation of Concolic Semantics

We now develop the staged concolic semantics $\mathbb{S}[\![\cdot]\!]$ for μWasm . Specializing $\mathbb{S}[\![\cdot]\!]$ to a source program generates efficient concolic execution code and realizes the first Futamura projection [19, 20]. Since the unstaged semantics $\mathbb{C}[\![\cdot]\!]$ (Figure 4) is compositional [28], we derive $\mathbb{S}[\![\cdot]\!]$ in Figure 7 by adding stage annotations to $\mathbb{C}[\![\cdot]\!]$ without altering the concolic evaluation logic. During specialization, the staged semantics mostly unrolls recursive calls over subterms.

In the presentation of staged semantics, variables bound to current stage values are **orange**, while those bound to staged expressions are **blue**. We do not add any color for variables at next stage.

Staging Semantic Domains. The domains of the staged semantics are derived from the concolic semantics (Figure 4) by analyzing whether values in each domain depend on the static program or dynamic input. If they depend on dynamic input, we lift their type to the second stage by wrapping it in *Expr*. The stack, environment, and path condition contain runtime values, so we lift their types to the second stage. Since the generated code invokes continuations with these runtime states, continuations are also lifted. The trail is a list of continuations, but only the continuations within the list are residualized:

$$\theta^\# \in \text{Trail}^\# = \text{List}[\text{Expr}[\text{Cont}^\#]]$$

A careful (manual) binding-time analysis of the concolic semantics shows that the static source-program structure and control flow, such as *br i*, completely determine the trail structure and its access indices at compile time. Because its structure is known statically, the list remains a current-stage value and need not be lifted to *Expr*[*List*[*Cont*[#]]].

Compiling Mutually Recursive Functions. Since WebAssembly function definitions can be mutually recursive, directly unrolling recursive calls could lead to non-terminating specialization. To ensure termination in the staged semantics, we introduce a run-time construct *Memo* that memoizes compiled Wasm function definitions. Instead of unrolling calls, we generate code that, at runtime, looks up the callee's compiled function in *Memo* and invokes it.

$$\text{memo} \in \text{Memo} = \text{Map}[\text{Identifier}, (\text{Env}^\#, \text{PC}^\#, \text{Cont}^\#, \text{Memo}) \rightarrow \text{Ans}]$$

We define a module semantics $\mathbb{M}[\![\cdot]\!]$ (Figure 6) that traverses all functions in a Wasm module only once and produces the *Memo* with their staged function definitions. *Memo* is self-referential: the

Staged Concolic Continuation Semantics

$$\begin{aligned}
\mathbb{S}[\cdot] : \text{List}[\text{Inst}] &\rightarrow \text{Expr}[\text{Memo}] \\
&\rightarrow (\text{Expr}[\text{Stack}^\#] \times \text{Expr}[\text{Env}^\#] \times \text{Expr}[\text{PC}^\#] \times \text{Expr}[\text{Cont}^\#] \times \text{Trail}^\#) \rightarrow \text{Expr}[\text{Ans}] \\
\mathbb{S}[t.\text{const } c :: \text{rest}] &(\text{memo})(\sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\#) = \\
&\mathbb{S}[\text{rest}] (\text{memo})(\langle \{\$ \text{lift}(c), \$ \text{lift}(c) \} :: \$\sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\# \rangle) \\
\mathbb{S}[\text{block } (t^m \rightarrow t^n) \text{ es} :: \text{rest}] &(\text{memo})(\sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\#) = \\
&\langle \{ \text{let } \kappa_1^\# := \lambda(\sigma_1^\#, \rho_1^\#, \pi_1^\#). \$ \{ \mathbb{S}[\text{rest}] (\text{memo})(\langle \{\sigma_1^\# \}_n \uparrow \$\sigma^\#. \text{pop}_m \}, \rho_1^\#, \pi_1^\#, \kappa^\#, \theta^\#) \} \text{ in} \\
&\quad \$ \{ \mathbb{S}[\text{es}] (\langle \{\sigma^\#. \text{take}_m \}, \rho^\#, \pi^\#, \kappa_1^\#, \kappa_1^\# :: \theta^\# \rangle) \} \} \\
\mathbb{S}[\text{loop } (t^m \rightarrow t^n) \text{ es} :: \text{rest}] &(\text{memo})(\sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\#) = \\
&\langle \{ \text{let } \kappa_1^\# := \lambda(\sigma_1^\#, \rho_1^\#, \pi_1^\#). \$ \{ \mathbb{S}[\text{rest}] (\text{memo})(\langle \{\sigma_1^\# \}_n \uparrow \$\sigma^\#. \text{pop}_m \}, \rho_1^\#, \pi_1^\#, \kappa^\#, \theta^\#) \} \text{ in} \\
&\quad \text{fix } \kappa_2^\# := \lambda(\sigma_2^\#, \rho_2^\#, \pi_2^\#). \$ \{ \mathbb{S}[\text{es}] (\text{memo})(\langle \{\sigma_2^\# \}_m \}, \rho_2^\#, \pi_2^\#, \kappa_1^\#, \kappa_2^\# :: \theta^\#) \} \text{ in} \\
&\quad \kappa_2^\# (\sigma^\#. \text{take}_m, \$\rho^\#, \$\pi^\#) \} \\
\mathbb{S}[\text{if } (t^m \rightarrow t^n) \text{ es}_1 \text{ es}_2 :: \text{rest}] &(\text{memo})(\sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\#) = \\
&\langle \{ \text{let } \langle v, s \rangle := \$\sigma^\#. \text{head} \text{ in} \\
&\quad \text{let } \sigma_{\text{arg}}^\# := \$\sigma^\#. \text{tail}.\text{take}_m \text{ in} \\
&\quad \text{let } \kappa_1^\# := \lambda(\sigma_1^\#, \rho_1^\#, \pi_1^\#). \$ \{ \mathbb{S}[\text{rest}] (\text{memo})(\langle \{\sigma_1^\# \}_n \uparrow \$\sigma^\#. \text{tail}.\text{pop}_m \}, \rho_1^\#, \pi_1^\#, \kappa^\#, \theta^\#) \} \text{ in} \\
&\quad \text{let } \pi_1^\# := \text{if } v \equiv 0 \text{ then } \neg s :: \$\pi^\# \text{ else } s :: \$\pi^\# \text{ in} \\
&\quad \text{if } v \equiv 0 \text{ then } \$ \{ \mathbb{S}[\text{es}_1] (\text{memo})(\langle \sigma_{\text{arg}}^\#, \rho^\#, \pi_1^\#, \kappa_1^\#, \kappa_1^\# :: \theta^\# \rangle) \} \\
&\quad \text{else } \$ \{ \mathbb{S}[\text{es}_2] (\text{memo})(\langle \sigma_{\text{arg}}^\#, \rho^\#, \pi_1^\#, \kappa_1^\#, \kappa_1^\# :: \theta^\# \rangle) \} \} \\
\mathbb{S}[\text{br } \ell :: \text{rest}] &(\text{memo})(\sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\#) = \langle \{ \$ \{ \theta^\#(\ell) \} (\sigma^\#, \$\rho^\#, \$\pi^\#) \} \\
\mathbb{S}[\text{call } x :: \text{rest}] &(\text{memo})(\sigma^\#, \rho^\#, \pi^\#, \kappa^\#, \theta^\#) = \\
&\text{let } \{ \text{type} : t^m \rightarrow t^n, \text{locals} : ts, \text{body} : es \} := \text{lookupFunc}(x) \text{ in} \\
&\langle \{ \text{let } f := \$\text{memo}(\$ \{ \text{lift}(x) \}) \text{ in} \\
&\quad \text{let } \rho_1^\# := \text{buildEnv}(\$ \sigma^\#. \text{take}_m, \$ \{ \text{lift}(ts) \}) \\
&\quad \text{let } \kappa_1^\# := \lambda(\sigma_1^\#, \rho_1^\#, \pi_1^\#). \$ \{ \mathbb{S}[\text{rest}] (\text{memo})(\langle \{\sigma_1^\# \}_n \uparrow \$\sigma^\#. \text{pop}_m \}, \rho_1^\#, \pi_1^\#, \kappa^\#, \theta^\#) \} \text{ in} \\
&\quad \$f(\rho_1^\#, \$\pi^\#, \kappa_1^\#, \$\text{memo}) \}
\end{aligned}$$

Fig. 7. Staged concolic semantics for selected cases (full definition can be found in extended version [62]).

functions in Memo access the Memo itself to call to other functions. When later interpreting a call instruction, we generate code looking up the corresponding compiled function in the memo. In practice, we implement staging with LMS [43], which internally memoizes staged function definitions, so recursive references reuse an existing definition rather than recompile its body. This prevents recursive calls from causing non-terminating specialization and eliminates the need to maintain Memo explicitly.

Adding Stage Annotations to Concolic Semantics. Now, we can refactor the concolic semantics into its staged version. The staging annotations indicate the evaluation time of expressions, staged expressions marked with ' $\cdot$ ' are delayed to runtime, while those unmarked are evaluated during specialization. Compared to the concolic semantics $\mathbb{C}[\cdot]$, we replace the pattern matching on the stack $\sigma^\#$ with explicit stack manipulation operations head and tail for clarity. $\mathbb{S}[\cdot]$ does not evaluate the final result of the program, instead, it constructs next-stage expressions that compute the final result when executed. In the following, we explain a few representative cases.

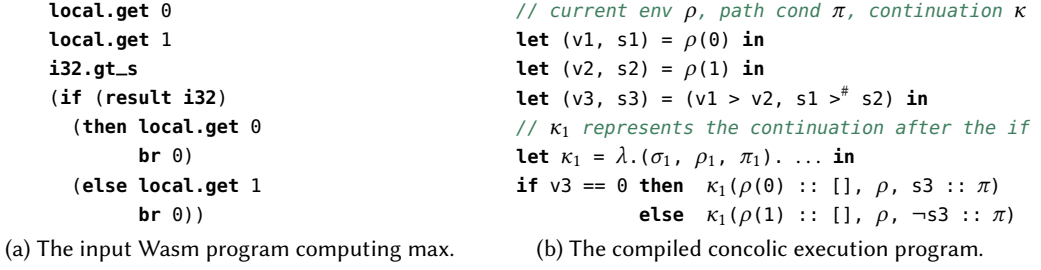


Fig. 8. An example of compilation via staging.

- In the case of $t.\text{const } c$, we splice the current stack $\sigma^\#$ into a staged expression that constructs the new stack. Then, we call $\mathbb{S}[\![\cdot]\!]$ recursively with the staged new stack to build the expression that computes the final result. Since c is a current stage value, we call meta-function lift to convert it to a next stage value before using it in the staged expression.
- For blocks, $\mathbb{S}[\![\text{block } \dots]\!]$ produces a staged let-expression that defines the continuation $\kappa_1^\#$ and splices the result of applying $\mathbb{S}[\![\text{es}]\!]$ to the block body.
We define continuation $\kappa_1^\#$ by splicing the call of $\mathbb{S}[\![\text{rest}]\!]$. The continuation $\kappa_1^\#$ maintains a valid stack matching the return type of this block when calling $\mathbb{S}[\![\text{rest}]\!]$: $\kappa_1^\#$ first truncates the top n elements from the incoming stack $\sigma_1^\#$ and appends the remaining stack $\sigma^\#$ after dropping its top m elements. When invoked, $\kappa_1^\#$ receives the latest environment $\rho_1^\#$ and path condition $\pi_1^\#$. Since $\rho_1^\#$ and $\pi_1^\#$ are variables bound by a staged λ -term, we quote them when referencing them in the spliced expression.
Finally, to generate code computing the block result, we call $\mathbb{S}[\![\text{es}]\!]$ on the block body using the top m elements of the stack $\sigma^\#$, the continuation ' $\kappa_1^\#$ ', and the trail ' $\kappa_1^\# :: \theta^\#$ '.
- The loop case is similar to block, except that we additionally construct a staged continuation $\kappa_2^\#$ representing the start of the loop body. Invoking $\kappa_2^\#$ with the current stack, environment, and path conditions produces the final result of this case.
- For the if instruction, we first pop the condition $\langle v, s \rangle$ from the stack and prepare the block arguments by taking the top m items from the remaining stack. The remaining computation after the if is residualized into continuation $\kappa_1^\#$. We build a new path condition $\pi_1^\#$ by prepending s for the true branch and $\neg s$ for the false branch to the current path condition $\pi^\#$.
Finally, we generate a staged conditional expression whose branches splice the current-stage calls $\mathbb{S}[\![\text{es}_1]\!]$ and $\mathbb{S}[\![\text{es}_2]\!]$, respectively. We quote ' $\sigma_{arg}^\#$ ', ' $\pi_1^\#$ ', and ' $\kappa_1^\#$ ' when passing them to $\mathbb{S}[\![\cdot]\!]$, since these variables are created inside the staged expression, while we pass $\rho^\#$ directly because it is a current-stage variable bound to a staged expression.
- In the case of call, we first retrieve the function identifier using the compile-time function lookupFunc. With the identifier x , we look up the run-time Memo-table *memo* to retrieve the compiled function f . We prepare the argument environment $\rho_1^\#$ and continuation $\kappa_1^\#$, similar to the unstaged counterpart. Finally, we generate a runtime call to f with the prepared arguments.

A Working Example. Figure 8a shows a Wasm block that computes the maximum of two integers stored in local variables using an if instruction. Its compiled concolic execution, obtained by applying the staged concolic semantics $\mathbb{S}[\![\cdot]\!]$, is shown in Figure 8b. Since the dispatching on the instructions has been performed during the staging, the specialized program contains only the computations necessary to compute the maximum of two integers and build the path conditions. The administrative instructions such as stack resizing and pushing/popping frames have been compiled away. In other words, the interpretation overhead has been eliminated.

4.3 Staged Concolic Scheduler

We adapt the exploration scheduler to work with the compiled concolic program and show it in the extended version of this paper [62]. The scheduler is a runtime function that invokes the staged module semantics $\mathbb{M}[\![\text{mod}]\!]$ to compile all Wasm functions. Then, using MSP's standard facility `run`, the scheduler evaluates the generated staged expression to obtain a map of compiled functions.

The compiled program executes only the essential concolic logic for the source program, eliminating interpretation overhead from instruction dispatching. In practice, the scheduler can be implemented in the same language as the generated program and invoke the compiled code directly. For example, in GENWASYM implementation (Section 7), we generate C++ code for concolic execution of Wasm programs, and the scheduler is also implemented in C++ as part of the runtime, which will be linked with the generated code.

4.4 Termination and Correctness of Compilation

Since we are building a compiler, we must ensure that our staging process terminates and produces staged programs that preserve the source semantics. The staged concolic interpreter $\mathbb{S}[\![\cdot]\!]$ avoids diverging for recursive Wasm function calls by using the Memo to store staged function definitions. We construct a staged function call by looking up its definition from Memo, rather than directly evaluating (i.e., specializing) the function's body.

Compared to building a concolic-compiler from scratch, our approach is not only simpler but also easier to reason about its correctness, since the unstaged interpretation serves as the specification:

$$\mathbb{V}input, fs, es.run(\mathbb{S}[\![es]\!])(\mathbb{M}[\![fs]\!]))(input, \rho_0^\#, \pi_0^\#, \kappa_{halt}, [\kappa_{halt}]) \equiv \mathbb{C}[\![es]\!](input, \rho_0^\#, \pi_0^\#, \kappa_{halt}, [\kappa_{halt}])$$

The equation states that the staging semantics $\mathbb{S}[\![\cdot]\!]$ and $\mathbb{M}[\![\cdot]\!]$ must agree with the original concolic semantics $\mathbb{C}[\![\cdot]\!]$. This property requires the source program and the specialized program always return the same result when given the same *input* and other initial arguments, including empty environment $\rho_0^\#$, empty path condition $\pi_0^\#$, halt continuation κ_{halt} , and trail with only κ_{halt} . The specialized program is obtained by applying `run` on the staged term produced by $\mathbb{S}[\![\cdot]\!]$ and $\mathbb{M}[\![\cdot]\!]$. On the right-hand side of the equation, $\mathbb{C}[\![\cdot]\!]$ does not take *fs* as input, because it implicitly accesses function definitions in the module via `lookupFunc`.

5 Snapshotting Compiled Concolic Execution with Continuations

The scheduler defined in Section 4.3 re-executes the program from the beginning for each new input, causing redundant work along shared path prefixes. This redundancy becomes costly for programs with deep execution paths. We now present the snapshot-reuse optimization, which uses continuations to eliminate redundant execution in the compiled concolic program and its scheduler.

Conceptually, concolic execution explores an implicit execution tree whose nodes represent branch points, leaves mark execution ends, and root-to-leaf paths correspond to complete traces. As the tree expands, both the number of leaves and the re-execution cost grow exponentially. The key idea is to save both program states *and* continuations at branch points, which together we call *snapshots*. Resuming from snapshots explores new paths instead of re-executing shared path segments.

From State-Passing to Global Mutable States. Resuming from a snapshot requires symbolic states sufficient to reconstruct the entire program state. However, the staging semantics in Figure 7 works on a local stack and environment that are passed as parameters. These parameters capture only a portion of the program state rather than the entire state. For example, in Figure 7, we always ensure the function's local stack is initialized to empty before entering the function body, hence the caller's stack is not visible to the callee. Therefore, we refactor the state-passing interpreter to use mutable global variables, i.e., $\sigma_g^\#$, $\rho_g^\#$, and $\pi_g^\#$, which hold references to the *entire* stack, environment,

Staged Snapshotting Concolic Continuation Semantics

Global stack $\sigma_g^\#$, global environment $\rho_g^\#$, global path condition $\pi_g^\#$

$$\begin{aligned} \mathbb{S}[\![\cdot]\!] : \text{List}[\text{Inst}] &\rightarrow (\text{Expr}[\text{Cont}^\#] \times \text{Trail}^\#) \rightarrow \text{Expr}[\text{Ans}] \\ \mathbb{S}[\![\text{if } (t^m \rightarrow t^n) \text{ } es_1 \text{ } es_2 :: \text{rest}]\!](\pi^\#, \kappa^\#, \theta^\#) = \\ &'\{ \text{let } \langle v, s \rangle := \$\sigma_g^\#.pop^! \text{ in} \\ &\quad \text{let } (_m \# \sigma_{old}^\#) := \$\sigma_g^\# \text{ in} \\ &\quad \text{let } \kappa_1^\# := \lambda(). \$\sigma_g^\# \leftarrow \lfloor \$\sigma_g^\# \rfloor_n \# \sigma_{old}^\#; \$\{ \mathbb{S}[\![\text{rest}]\!](\kappa^\#, \theta^\#) \} \text{ in} \\ &\quad \text{let } \kappa_{tru}^\# := \lambda(). \$\{ \mathbb{S}[\![es_1]\!](\kappa_1^\#, \kappa_1^\# :: \theta^\#) \} \text{ in} \\ &\quad \text{let } \kappa_{fls}^\# := \lambda(). \$\{ \mathbb{S}[\![es_2]\!](\kappa_1^\#, \kappa_1^\# :: \theta^\#) \} \text{ in} \\ &\quad \text{let } \Sigma_{tru} := \langle \kappa_{tru}^\#, \$\sigma_g^\#.sym, \$\rho_g^\#.sym \rangle \text{ in} \\ &\quad \text{let } \Sigma_{fls} := \langle \kappa_{fls}^\#, \$\sigma_g^\#.sym, \$\rho_g^\#.sym \rangle \text{ in} \\ &\quad \text{if } v \equiv 0 \text{ then } \$\pi^\#.push^!(\neg s, \Sigma_{tru}); \kappa_{fls}^\#() \text{ else } \$\pi^\#.push^!(s, \Sigma_{fls}); \kappa_{tru}^\#() \} \end{aligned}$$

Fig. 9. The staged snapshotting concolic semantics for the conditional instruction.

and path condition, respectively. Accordingly, the continuation function type does not take any argument and performs effects by updating these globals: $\kappa^\# \in \text{Cont}^\# = () \rightarrow \text{Ans}$.

Capturing Snapshots. A snapshot is a triple consisting of continuations, symbolic stacks, and symbolic environments. Together they capture necessary information to resume execution from a branch point. We also redefine $\text{PC}^\#$ to include snapshots of the *untaken* branch in addition to the constraints of the *taken* branch:

$$\begin{aligned} \Sigma \in \text{Snapshot} &= \text{Cont}^\# \times \text{List}[\text{Symbolic}] \times \text{List}[\text{Symbolic}] \\ \pi^\# \in \text{PC}^\# &= \text{List}[\text{Symbolic} \times \text{Snapshot}]. \end{aligned}$$

Figure 9 highlights the modification to the staged semantics in Figure 7 for handling if instructions. The key change is to represent the computations of the “then” and “else” branches as next-stage continuations, $\kappa_{tru}^\#$ and $\kappa_{fls}^\#$. Using these continuations and the current symbolic states, we create the corresponding snapshots Σ_{tru} and Σ_{fls} . Since the stack consists of concolic pairs, we extract the symbolic stack using $\sigma^\#.sym$, and apply the same to the environment. We then update the global path condition according to the branch taken and invoke the corresponding continuation.

Since now the symbolic stack/environment are global mutable states, operations on them also become imperative. At the beginning, we pop the concolic condition from the global stack $\sigma^\#$ by $\text{pop}^!$. We mark these effectful operations on global states with a superscript exclamation mark. Recall that in WebAssembly, the type of if regulates the input/output stack: the branches can access top m elements on the stack, and each branch leaves the stack with n new elements after removing the m inputs. Therefore, we save the stack below those m inputs into $\sigma_{old}^\#$. When resuming via $\kappa_1^\#$, we use the result of prepending the n outputs to $\sigma_{old}^\#$ to restore the global stack $\sigma_g^\#$.

Resuming from Snapshots. Algorithm 2 shows a modified scheduler that reconstructs concrete states from snapshots to resume the execution without re-executing from the beginning.

The worklist pairs path conditions and their corresponding snapshots. It is initialized with an empty path condition and a snapshot of the program entry. In each iteration, the scheduler pops a pair and checks the condition’s feasibility as in Section 3. If the condition is satisfiable, it uses the model to instantiate the snapshot’s symbolic states to update the global states. The scheduler then invokes the continuation stored in the snapshot to resume execution from the branch point. The

Algorithm 2 Concolic Snapshot Scheduler

```

1: function SCHEDULER( $mod \in \text{Module}, f \in \text{Identifier}, input \in \text{List}[\text{Symbolic}]$ )
2:    $memo \leftarrow \text{run } \mathbb{M} \llbracket mod \rrbracket$ 
3:    $entry \leftarrow memo(f)$ 
4:    $WL \leftarrow \{ \langle [], \langle \lambda().entry(\kappa_{halt}^\#, memo), input, [] \rangle \rangle \}$   $\triangleright WL \in \mathcal{P}(\text{List}[\text{Symbolic}] \times \text{Snapshot})$ 
5:    $Seen \leftarrow \emptyset$   $\triangleright Seen \in \mathcal{P}(\text{List}[\text{Symbolic}])$ 
6:   while  $WL \neq \emptyset$  do
7:      $\text{pop } \langle \pi^\#, \langle \kappa, \sigma^\#, \rho^\# \rangle \rangle$  from  $WL$ 
8:     if  $\pi^\# \in Seen$  then continue
9:      $m \leftarrow \text{solve}(\pi^\#)$ 
10:    if  $m = \text{UNSAT}$  then continue
11:     $\langle \sigma_g^\#, \rho_g^\#, \pi_g^\# \rangle \leftarrow \langle \text{instantiate}(\sigma^\#, m), \text{instantiate}(\rho^\#, m), \pi^\# \rangle$   $\triangleright$  instantiate global stack/environment
12:     $\triangleright$  update global path condition
13:     $\Pi^\# \leftarrow \kappa()$ 
14:     $Seen \leftarrow Seen \cup \{ \Pi^\#.pc \}$   $\triangleright \Pi^\#.pc$  extracts a list of symbolic conditions
15:    for all  $\langle \pi_1, \Sigma_1 \rangle \in \text{NEGATE}(\Pi^\#)$  do
16:       $WL \leftarrow WL \cup \{ \langle \pi_1, \Sigma_1 \rangle \}$ 
17: function NEGATE( $\Pi^\# \in \text{PC}^\#$ )  $\rightarrow \text{List}[\text{List}[\text{Symbolic}] \times \text{Snapshot}] \dots$   $\triangleright$  (see extended version [62])

```

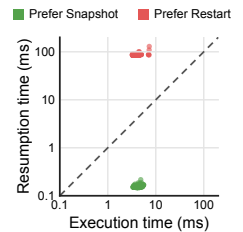
execution returns a new $\Pi^\#$ pairing taken-branch path conditions with snapshots of the untaken branches. Finally, the scheduler calls `NEGATE` to append the untaken branches' path conditions and snapshots to the worklist for future exploration. Thus, continuation-based resumption avoids re-executing shared path prefixes while retaining the performance of compiled execution.

Improving Asymptotic Complexity. By applying snapshots to avoid re-execution, we improve the asymptotic complexity of concolic execution. Suppose the implicit execution tree of a concolic execution contains n nodes. The tree's height ranges from $O(\log n)$ (balanced) to $O(n)$ (unbalanced). The complexity of exploring a path from root to leaf takes between $O(\log n)$ and $O(n)$, and exploring all paths from the root takes between $O(n \log n)$ to $O(n^2)$. With snapshots, however, each exploration starts from a fresh node that has never been visited before. As a result, every node is visited exactly once, reducing the overall complexity to $O(n)$ in the worst case.

6 Deciding When to Use Snapshots

Section 5 introduced snapshots to uniformly avoid re-executing previously explored program prefixes. Its practical effectiveness, however, depends on both the depth of the shared execution prefix and the size of the resumed symbolic state. For programs whose execution paths share only shallow prefixes, creating and instantiating snapshots may cost more than re-executing the program. Thus, applying snapshot reuse uniformly, as in Algorithm 2, may yield little benefit or even degrade performance for such programs.

The figure on the right illustrates this trade-off for the quicksort benchmark used in our evaluation (Section 8.2). The benchmark contains 862 complete paths and 861 branch points. Each of the 861 plotted points represents the choice at one branch point between resuming from its snapshot and re-executing from the program entry to reach it. The x-axis shows the re-execution cost and the y-axis the snapshot-resumption cost. Of these 861 choices, 283 lie above the dashed equal-cost line, indicating that snapshot resumption is more expensive than re-execution. In this example, reusing



snapshots uniformly at these unprofitable points would therefore make the overall execution slower than uniform re-execution, as the overall cost of these 283 snapshots is substantially higher than the cost of re-execution.

Additionally, the snapshot mechanism in Figure 9 captures only the states represented in the semantics. However, realistic programs interact with external environments (e.g., the OS), whose states are typically not preserved in a snapshot. When an execution path involves such interactions, the symbolic state recorded in the snapshot may no longer be reachable in a concrete execution. In that case, resuming from the snapshot may break the soundness of concolic execution.

To make snapshot reuse both effective and sound, we need a strategy for deciding when to resume execution from a snapshot and when re-execution from the beginning is preferable. This section develops a heuristic for making this choice based on cost estimation while enforcing soundness.

Cost of Re-execution vs. Instantiation. The main cost of instantiating a snapshot comes from reconstructing the concrete state from the snapshot, which involves traversing all symbolic expressions in the symbolic stack and environment and evaluating them with respect to the SMT model. We estimate this cost by the number of symbolic expressions and denote it as C_{Ins} :

$$C_{\text{Ins}} = \delta * (\text{sizeOfSymExp}(\sigma^\#) + \text{sizeOfSymExp}(p^\#))$$

where parameter δ is the average timing cost of evaluating symbolic expressions in the solver. In practice, we can obtain δ by measuring the evaluation time of a large set of symbolic expressions. During runtime, we also record the execution time from the beginning of the program to the branch point as the re-execution cost, denoted C_{Re} .

Soundly Handling External Calls. External calls invoke functions whose implementations are unavailable to the concolic or symbolic execution engine, such as system calls or library calls. One way to handle external calls is to execute them concretely and treat the returned concrete values as their symbolic counterparts in subsequent concolic execution.

However, this strategy can make snapshot reuse unsound: a snapshot based on an observed concrete return value assumes that the external call returns the same value across executions. In a fresh execution from the beginning, the call may return a different value because its arguments or the external environment have changed. As a result, the symbolic states captured by the snapshot may be invalid, and the resumed execution may fail to correspond to a feasible program path.

To avoid creating unsound snapshots, we can conservatively disable resumption of snapshots whose execution prefix contains an external call. For such snapshots, instead of computing C_{Ins} using sizeOfSymExp , we set it to infinity, ensuring that the scheduler re-executes the program from beginning. A more sophisticated concolic execution engine can handle external calls soundly by symbolically modeling the semantics of external functions. We leave the integration of sound external call handling and its interaction with snapshot reuse as future work.

Heuristic Decision. The decision of whether to resume from a snapshot at a branch therefore reduces to comparing the cost of instantiation with that of re-execution. At runtime, we compute the ratio $C_{\text{Re}}/C_{\text{Ins}}$. If this ratio exceeds 1, re-execution is deemed more expensive than instantiation, and we therefore prefer to resume from the snapshot. Otherwise, we re-execute the program from the beginning. To implement this heuristic, we extend Snapshot with an additional field $\text{Cost} = \text{Int}$, which estimates the cost of reaching the program point at which the snapshot was created:

$$s \in \text{Snapshot} = \text{Cont}^\# \times \text{List}[\text{Symbolic}] \times \text{List}[\text{Symbolic}] \times \text{Cost}.$$

The extended version of this paper [62] presents the full heuristic scheduler in the appendix. In each iteration, the scheduler compares the cost stored in the snapshot against the estimated cost of instantiation, represented by the size of the symbolic state.

7 Implementation

We scale our approach to a substantially larger subset of WebAssembly and implement it in GENWASYM, a compiler written in Scala using the Lightweight Modular Staging (LMS) framework [43].

7.1 Compilation via Staging

At the core of GENWASYM is a staged concolic interpreter for WebAssembly, derived from our staged concolic semantics with snapshots. Compilation is performed by specializing this interpreter to a given Wasm program. Unlike the quasi-quotation and splicing presentations in earlier sections, the implementation uses LMS [43], which offers convenient staging APIs through operator overloading. As a result, while the implementation follows the same staging semantics and type-level stage discipline, it does not use explicit quasi-quotation/splicing syntax. LMS also provides a graph-based IR with built-in optimizations [9], as well as multiple code-generation backends, including C++. The core staged interpreter of GENWASYM comprises 1.5K lines of Scala code.

GENWASYM generates efficient C++ code for concolic execution in CPS. Each WebAssembly function is compiled into a C++ function that takes an additional continuation argument. Following the refactoring described in Section 5, the stack, environment, and path condition are maintained as global mutable runtime states, and the generated code operates directly on them. Continuations therefore do not need to thread these states explicitly. Moreover, the continuation trail disappears entirely in the generated code, since all computation over the trail is resolved at the first stage (Section 4). WebAssembly stack and frame operations compile directly to operations on the corresponding global runtime structures. To support efficient state snapshotting, we represent the stack and environment using persistent data structures from Immer [40], which provide efficient copy and update operations while reducing the need to copy large amounts of memory.

7.2 Supporting More WebAssembly Features

GENWASYM supports all numeric types in Wasm 2.0 [1], including 32-/64-bit integers and floating-point numbers. Symbolically, these values are represented using the bit-vector and floating-point theories of Z3 [17]. Beyond the features covered in our core formalization, GENWASYM supports a large fragment of commonly used WebAssembly features, including additional arithmetic operations, function tables, indirect calls, global variables, and linear memory. Supporting these additional features ensures that GENWASYM can be applied to real-world Wasm programs. The implementation of these features still follows the semantics-based approach, and here we discuss the implementation of some of these features, including global variables, linear memory, and indirect control flow.

Global Variables. As described in Section 5, we have refactored the semantics by lifting the stack, environment, and path condition into mutable global states. To support WebAssembly global variables, we add a separate global component to the semantics. In GENWASYM, WebAssembly globals are compiled into a C++ global data structure, and global-variable instructions (e.g., `global.get` and `global.set`) are translated into direct operations on it. As with the stack and environment, we use persistent data structures from Immer [40] to represent global state, enabling efficient snapshotting.

Linear Memory. Support for linear memory is essential for symbolic reasoning about realistic data structures, which are typically allocated in WebAssembly memory. In the official WebAssembly semantics, linear memory is modeled as a mapping from addresses to bytes. For concolic execution, we refine this model by representing each address as a pair of concrete and symbolic byte values.

WebAssembly provides memory operations over multiple data types, including 32-/64-bit integers and floating-point numbers. To handle them uniformly, we follow WASP and implement memory operations as byte-level operations on both concrete and symbolic memory. For example, `i32.load`

reads four consecutive bytes and combines them into a 32-bit value, while `i32.store` decomposes a 32-bit value into four bytes and writes them back to consecutive addresses. In the implementation, GENWASYM represents memory using persistent data structures, enabling efficient snapshotting.

Indirect Control Flow. WebAssembly indirect control flow instructions such as `call_indirect` allow the call target to be resolved at runtime rather than fixed statically. This mechanism helps compile higher-level languages with dynamic dispatch to WebAssembly. The WebAssembly standard assigns a unique index to each function, and a runtime global table stores these indices. The `call_indirect` instruction uses its operand to look up this table and resolve the target. Following the specification, GENWASYM initializes a global table mapping WebAssembly function indices to C++ function pointers at program startup. This table is initialized when the compiled program starts and is used to resolve indirect calls at runtime. We compile each `call_indirect` instruction into a table lookup followed by a call to the resolved function.

Limitations. As a prototype implementation, GENWASYM already supports a substantial subset of WebAssembly 2.0 [1], but it does not yet cover all features defined in the WebAssembly standard. GENWASYM currently does not support several additional WebAssembly features, such as SIMD instructions, function references, and module import/export. Nevertheless, as long as the specification provides clear semantics for these features, extending GENWASYM to support these features would be straightforward and requires moderate engineering effort.

8 Empirical Evaluation

This section evaluates the performance of GENWASYM with respect to the following three questions:

RQ1: How much speedup does compilation provide over interpretation-based approaches?

RQ2: When snapshot and the heuristic are effective in GENWASYM?

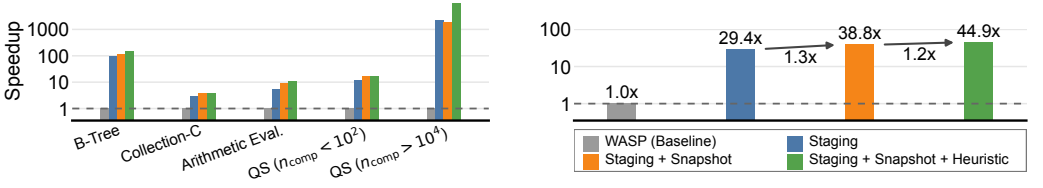
RQ3: Is GENWASYM effective in finding real bugs?

Environment and Setup. All experiments ran on Ubuntu 22.04 LTS with an AMD Ryzen 7 7735H CPU and 16 GB RAM. We used Z3 4.8.12 and compiled generated code with Clang 22.1.0 at `-O3`. For the baseline, we used WASP version 0.2.3, the latest release at submission time of this paper.

For performance measurement, we decompose the total wall-clock time into instruction-execution time and SMT-solving time. Our evaluation focuses on instruction execution, reporting the arithmetic mean over 5 runs for each benchmark. We also report end-to-end execution time (including time spent in SMT solvers) for completeness, although it depends on symbolic-expression encodings and query optimizations, which are beyond the scope of this paper. To validate the generated SMT queries and ensure the fairness of the comparison, we verify that the number of explored paths matches that of the baseline WASP [35].

Benchmarks. We evaluate against four groups of benchmarks, shown in Table 1. To assess performance on practical workloads, we adopt two benchmarks from prior work: (1) the B-Tree benchmark from WASP [35], and (2) the Collections-C test suites [47], which exercise various data structures from the Gillian project [44]. Compared with their original configurations in WASP, we use larger symbolic inputs and explore more paths for both the B-Tree benchmark [35] and the Collections-C benchmark [47] to better approximate practical workloads. The B-Tree benchmark contains 26 cases, while the Collections-C benchmark contains 143 across 8 modules, yielding 169 cases in total.

We include two micro-benchmarks to understand the performance characteristics of snapshot reuse: (3) an arithmetic evaluator and (4) a quicksort implementation. These two benchmarks are designed to isolate the performance effects of snapshot reuse (RQ2). The quicksort benchmark is written in C and compiled to Wasm, whereas the arithmetic evaluator is written directly in



(a) Per-benchmark speedup. Bars show WASP (1×) (b) Overall average speedup, computed as the geometric mean of all the benchmarks speedups across 4 reuse, uniform snapshot reuse, and heuristic snapshot reuse. Higher is better; QS denotes quicksort. benchmark groups. Arrows show the speedup factor from each GENWASYM configuration to the next.

Fig. 10. Speedup from staging, snapshot reuse and snapshot reuse heuristics.

Wasm. The arithmetic-evaluator benchmark contains 9 Wasm programs, each of which parses and evaluates a string-represented arithmetic expression. The programs vary in expression size, measured by the number of arithmetic operators, and the number of explored paths.

The quicksort benchmark varies both the input-array size and the number of symbolic elements. We initialize the symbolic inputs with either atomic symbolic variables or complex symbolic arithmetic expressions, allowing us to study how symbolic-state complexity affects snapshot reuse.

To answer RQ3, we use two Collections-C benchmarks from WASP [35] to evaluate bug-detection effectiveness. Each benchmark has two versions: a *buggy version* containing two known bugs and a *correct version* in which both bugs are fixed.

Baselines. There are several recent interpretation-based symbolic execution engines for WebAssembly, including MANTICORE [36], SEEWASM [25], WASP [35] and WASP’s successor OWI [3]. For RQ1, we choose to use WASP as the baseline for comparison, as it is the only tool among them implementing the same concolic execution semantics as GENWASYM. OWI and MANTICORE both implement a general parallel symbolic execution semantics, which is not driven by concrete execution. SEEWASM constructs a control-flow graph for Wasm program, then symbolically explores the CFG using BFS or DFS strategy, without concrete inputs.

However, as these tools realize similar testing capabilities for end-users, their relative performance is worth noting to better understand where GENWASYM would stand. On the same B-Tree benchmark, OWI with 1 worker reports similar performance comparable to WASP (0.8× slowdown), while OWI with 24 workers achieves an average 4.1× speedup over WASP [3]. In addition, WASP reports 5.4×–26.2× speedups over MANTICORE [35], whereas SEEWASM reports 1.2×–4.8× speedups over MANTICORE [25]. Taken together, these published results from the literature suggest that WASP is a competitive baseline for our evaluation.

To the best of our knowledge, there currently exists no instrumentation-based concolic engine for Wasm, so we could not include such a baseline in our evaluation.

8.1 RQ1: GENWASYM Performance Compared to Interpretation-Based Approaches

Method. To isolate the benefits of staging from snapshot-reuse, we disable snapshot-reuse in GENWASYM for RQ1, forcing each execution to restart from the program entry, as in WASP. We report the mean instruction-execution time and end-to-end execution time, with a two-hour wall-clock timeout per run. For WASP, the measurements exclude WebAssembly module parsing.

We configure the B-Tree benchmark using two parameters: the numbers of ordered and unordered symbolic inputs, denoted by n_o and n_u , respectively. The extended Collections-C benchmark contains n_i test cases for each data structure, listed in Table 1. We compile all test cases to WebAssembly

Table 1. Performance comparison between GENWASYM and WASP.

		n_{paths}	WASP		GENWASYM										
			$T_{WASP}^{exec} \pm \sigma_{WASP}^{exec}$	T_{WASP}^{total}	$T_{no reuse}^{exec} \pm \sigma_{no reuse}^{exec}$	$T_{no reuse}^{total}$	$S_{staging}$	$T_{snapshot}^{exec} \pm \sigma_{snapshot}^{exec}$	$T_{snapshot}^{total}$	$S_{snapshot}$	$T_{heuristic}^{exec} \pm \sigma_{heuristic}^{exec}$	$T_{heuristic}^{total}$	$S_{heuristic}$		
n_o	n_u	Extended B-Tree Benchmark													
2	2	12	0.2237 ± 0.0243	2.14	0.0092 ± 0.0051	1.22	24.3×	0.0089 ± 0.0051	1.22	1.0×	0.0087 ± 0.0045	1.24	1.1×		
2	3	60	2.19 ± 0.0724	15.93	0.0398 ± 0.0093	12.14	55.0×	0.0363 ± 0.0082	12.24	1.1×	0.0362 ± 0.0085	12.28	1.1×		
3	2	20	0.5026 ± 0.0180	4.98	0.0150 ± 0.0051	3.67	33.5×	0.0149 ± 0.0055	3.73	1.0×	0.0143 ± 0.0049	3.63	1.0×		
3	3	120	6.46 ± 0.2174	45.26	0.0880 ± 0.0123	36.20	73.4×	0.0783 ± 0.0097	36.03	1.1×	0.0797 ± 0.0101	35.85	1.1×		
4	1	5	0.1206 ± 0.0046	1.28	0.0063 ± 0.0045	0.9061	19.1×	0.0061 ± 0.0043	0.9069	1.0×	0.0053 ± 0.0038	0.9058	1.2×		
4	2	30	1.09 ± 0.0396	10.44	0.0219 ± 0.0029	8.01	49.8×	0.0203 ± 0.0024	8.11	1.1×	0.0206 ± 0.0036	8.08	1.1×		
4	3	210	16.07 ± 0.4950	109.21	0.1806 ± 0.0213	87.23	89.0×	0.1620 ± 0.0201	88.35	1.1×	0.1612 ± 0.0180	87.78	1.1×		
5	1	6	0.1172 ± 0.0045	2.15	0.0065 ± 0.0044	1.58	18.0×	0.0059 ± 0.0038	1.57	1.1×	0.0061 ± 0.0038	1.58	1.1×		
5	2	42	2.50 ± 0.0870	20.04	0.0439 ± 0.0166	16.49	56.9×	0.0386 ± 0.0161	16.15	1.1×	0.0339 ± 0.0059	15.97	1.3×		
5	3	336	41.04 ± 1.09	241.80	0.3590 ± 0.0316	202.25	114.3×	0.2940 ± 0.0348	202.04	1.2×	0.3016 ± 0.0400	204.70	1.2×		
6	1	7	0.3145 ± 0.0134	3.25	0.0065 ± 0.0012	2.49	48.4×	0.0059 ± 0.0016	2.49	1.1×	0.0057 ± 0.0010	2.48	1.1×		
6	2	56	3.60 ± 0.1181	33.49	0.0557 ± 0.0053	28.93	64.6×	0.0475 ± 0.0042	28.73	1.2×	0.0472 ± 0.0046	28.58	1.2×		
6	3	504	116.95 ± 1.98	493.44	0.6307 ± 0.0869	418.09	185.4×	0.5311 ± 0.0758	416.86	1.2×	0.5241 ± 0.0532	405.87	1.2×		
7	1	8	0.3716 ± 0.0026	4.65	0.0086 ± 0.0016	4.11	43.2×	0.0080 ± 0.0019	4.15	1.1×	0.0080 ± 0.0020	4.10	1.1×		
7	2	72	5.76 ± 0.0583	57.30	0.0903 ± 0.0081	54.97	63.8×	0.0767 ± 0.0087	54.82	1.2×	0.0757 ± 0.0065	53.99	1.2×		
7	3	720	409.15 ± 1.76	1124.46	1.18 ± 0.1798	922.24	346.7×	1.01 ± 0.1176	898.69	1.2×	1.01 ± 0.1000	891.70	1.2×		
8	1	9	0.6124 ± 0.0053	7.22	0.0141 ± 0.0054	6.54	43.4×	0.0121 ± 0.0042	6.59	1.2×	0.0115 ± 0.0025	6.57	1.2×		
8	2	90	10.38 ± 0.0143	92.83	0.1460 ± 0.0283	101.61	71.1×	0.1279 ± 0.0258	102.27	1.1×	0.1263 ± 0.0237	100.41	1.2×		
9	1	10	0.9422 ± 0.0103	10.62	0.0162 ± 0.0027	10.51	58.2×	0.0141 ± 0.0021	10.50	1.1×	0.0137 ± 0.0019	10.49	1.2×		
9	2	110	18.16 ± 0.1377	149.14	0.2056 ± 0.0414	165.65	88.3×	0.1798 ± 0.0318	166.51	1.1×	0.1828 ± 0.0403	168.10	1.1×		
9	3	1320	1970.68 ± 11.36	4087.60	3.02 ± 0.5759	2844.54	652.5×	2.36 ± 0.0586	2537.99	1.3×	2.64 ± 0.5663	2946.57	1.1×		
10	1	11	0.9066 ± 0.0203	15.52	0.0205 ± 0.0032	14.31	44.2×	0.0174 ± 0.0033	14.22	1.2×	0.0185 ± 0.0035	14.87	1.1×		
10	2	132	28.65 ± 0.6753	225.18	0.2829 ± 0.0389	243.39	101.3×	0.2441 ± 0.0338	244.59	1.2×	0.2392 ± 0.0331	236.70	1.2×		
10	3	1716	-	TO	4.44 ± 1.52	4333.33	-	3.37 ± 0.0906	3817.26	1.3×	3.65 ± 0.7807	4260.93	1.2×		
11	1	12	0.9233 ± 0.0031	19.44	0.0241 ± 0.0059	19.86	38.3×	0.0225 ± 0.0082	20.76	1.1×	0.0220 ± 0.0069	19.86	1.1×		
11	2	156	44.43 ± 0.7216	325.50	0.3617 ± 0.0828	348.49	122.8×	0.3285 ± 0.0769	369.19	1.1×	0.3202 ± 0.0831	355.48	1.1×		
Module		n_l	Geomean 66.7×												
			Geomean 1.1×												
Collections-C Benchmark															
Array	21	35	27.01 ± 0.1914	32.90	7.18 ± 0.0633	9.21	3.8×	7.21 ± 0.0448	9.23	1.0×	7.23 ± 0.0774	9.30	1.0×		
List	32	121	15.00 ± 0.1431	18.25	3.65 ± 0.0222	7.92	4.1×	3.59 ± 0.0218	7.92	1.0×	3.65 ± 0.0271	7.95	1.0×		
Slist	33	33	13.64 ± 0.2055	14.20	3.34 ± 0.0382	4.39	4.1×	3.45 ± 0.0393	4.51	1.0×	3.42 ± 0.0408	4.46	1.0×		
RingBuffer	3	3	3.78 ± 0.0866	3.91	0.9563 ± 0.0138	1.08	4.0×	0.9240 ± 0.0085	1.04	1.0×	1.01 ± 0.0254	1.13	0.9×		
Queue	4	19	75.87 ± 1.11	76.02	18.09 ± 0.6380	20.50	4.2×	19.88 ± 1.14	22.39	0.9×	17.92 ± 0.2644	20.27	1.0×		
Treemap	6	31	97.62 ± 1.29	99.13	27.88 ± 0.3734	33.53	3.5×	28.24 ± 0.5709	33.81	1.0×	27.79 ± 0.2857	33.22	1.0×		
Treetable	13	55	124.51 ± 2.51	126.51	38.14 ± 0.2135	45.34	3.3×	38.21 ± 0.1065	45.54	1.0×	37.60 ± 0.2809	44.74	1.0×		
Deque	31	46	104.38 ± 1.55	122.02	24.24 ± 0.0522	28.11	4.3×	24.26 ± 0.0543	28.11	1.0×	25.07 ± 0.0482	28.88	1.0×		
			Geomean			3.9×			Geomean			Geomean 1.0×			
Arithmetic Evaluator															
2000	8	8	4.15 ± 0.3104	4.39	0.3251 ± 0.0074	0.7376	12.8×	0.2222 ± 0.0030	0.6313	1.5×	0.1181 ± 0.0036	0.5260	2.8×		
2000	16	16	7.68 ± 0.0280	8.54	0.6474 ± 0.0075	2.34	11.9×	0.2391 ± 0.0022	1.92	2.7×	0.1278 ± 0.0009	1.80	5.1×		
2000	32	32	16.51 ± 0.6243	20.11	1.29 ± 0.0079	8.36	12.8×	0.2674 ± 0.0026	7.34	4.8×	0.1535 ± 0.0020	7.19	8.4×		
4000	8	8	10.12 ± 0.8170	10.41	0.7055 ± 0.0127	1.46	14.3×	0.4668 ± 0.0060	1.21	1.5×	0.2618 ± 0.0049	1.01	2.7×		
4000	16	16	19.58 ± 0.6480	20.72	1.39 ± 0.0109	4.50	14.1×	0.4953 ± 0.0045	3.58	2.8×	0.2787 ± 0.0052	3.31	5.0×		
4000	32	32	44.31 ± 3.13	49.23	2.76 ± 0.0275	15.52	16.1×	0.5433 ± 0.0048	13.47	5.1×	0.3252 ± 0.0058	13.14	8.5×		
8000	8	8	29.46 ± 0.7530	29.85	1.07 ± 0.0230	1.98	27.5×	0.6163 ± 0.0044	1.53	1.7×	0.4033 ± 0.0038	1.30	2.7×		
8000	16	16	65.37 ± 2.57	67.03	3.16 ± 0.0580	9.18	20.7×	1.05 ± 0.0116	7.08	3.0×	0.6012 ± 0.0047	6.33	5.3×		
8000	32	32	145.49 ± 9.18	152.22	5.82 ± 0.0148	29.51	25.0×	1.15 ± 0.0082	25.03	5.1×	0.7410 ± 0.0105	24.22	7.9×		
			Geomean			16.5×			Geomean			Geomean 4.8×			
Quicksort															
16	2	20	3.72 ± 0.0736	18.32	0.3131 ± 0.0012	9.80	11.9×	0.2472 ± 0.0016	9.74	1.3×	0.2448 ± 0.0023	9.88	1.3×		
16	2	30	16.69 ± 0.8284	73.97	1.39 ± 0.0038	35.00	12.0×	1.02 ± 0.0036	34.94	1.4×	1.03 ± 0.0047	35.03	1.3×		
16	2	40	51.44 ± 0.5677	208.43	4.17 ± 0.0533	88.36	12.3×	3.00 ± 0.0151	86.46	1.4×	3.03 ± 0.0429	86.77	1.4×		
16	2	50	227.0	483.66	9.90 ± 0.0606	177.44	13.3×	7.29 ± 0.0231	176.36	1.4×	7.40 ± 0.0225	176.48	1.3×		
25724	3	5	1085.52 ± 20.80	1088.07	0.4650 ± 0.2527	1.73	2334.5×	0.5711 ± 0.3127	2.06	0.8×	0.1072 ± 0.0532	1.37	4.3×		
25724	3	10	862	TO	4.34 ± 0.0556	21.38	-	9.79 ± 0.0333	27.98	0.4×	1.90 ± 0.0570	19.24	2.3×		
			Geomean			35.3×			Geomean			Geomean 1.8×			
			Overall Geomean			29.4×			Overall Geomean			Overall Geomean 1.5×			

Note: T_{exec} reports the mean instruction-execution time over 5 runs; σ_{exec} reports their standard deviations. Timings are measured in seconds. Time out (TO) is set to 2 hours for end-to-end execution.

RQ1 speedup $S_{staging} = T_{WASP}^{no reuse} / T_{WASP}^{exec}$ compares GENWASYM without snapshot-reuse with WASP;

RQ2 speedup $S_{snapshot} = T_{no reuse}^{no reuse} / T_{no reuse}^{snapshot}$ compares *uniform* snapshot-reuse with no-reuse;

RQ2 speedup $S_{heuristic} = T_{no reuse}^{no reuse} / T_{no reuse}^{heuristic}$ compares *heuristic* snapshot-reuse with no-reuse.

and perform concolic execution on the resulting modules. For every RQ1 benchmark, we ensure that GENWASYM and WASP explore the same number of paths.

The two benchmark suites play complementary roles in assessing performance across synthetic and realistic workloads. The extended B-Tree benchmark provides a controlled setting for systematically varying the state-space by imposing different ordering constraints on symbolic inputs. For example, $a > b$ excludes inputs where $a \leq b$, reducing the feasible input space. Collections-C provides more realistic workloads: its Wasm programs are compiled from a C implementation of various collection data structures that are widely used in real-world applications.

Results. Table 1 shows the results. We compute the speedup of staging $S_{staging}$ as $T_{exec}^{WASP} / T_{exec}^{noreuse}$. We observe that GENWASYM consistently and substantially outperforms WASP in instruction-execution time across all configurations of the extended B-Tree and Collections-C benchmarks (Table 1). On average, GENWASYM achieves a $66.7\times$ speedup over WASP on the B-Tree benchmark, with the highest $652.5\times$ speedup excluding the timed-out configuration $(n_o, n_u) = (10, 3)$. On the Collections-C benchmark, GENWASYM achieves a $3.9\times$ speedup. Overall, across 169 test programs spanning two benchmark groups, GENWASYM achieves an average speedup of $33.4\times$. On the arithmetic-evaluator benchmark and the quicksort benchmark, GENWASYM achieves an average speedup of $21.6\times$.

On the B-Tree benchmark, WASP exhibits increasingly severe slowdowns as execution time grows. We hypothesize that this degradation stems largely from its small-step interpretation and garbage collection (GC) in OCaml. The performance gains of GENWASYM come primarily from three sources: (1) Compilation eliminates the interpretation overhead of repeatedly dispatching instructions. (2) By leveraging the Wasm CPS semantics [53], the generated C++ code translates control-flow instructions directly into function calls, whereas WASP evaluates control flow through administrative instructions. (3) And lastly, the generated code avoids GC and uses efficient data structures to represent execution state, including the stack and environment.

Summary. Figure 10a summarizes the speedup results for each benchmark group. Overall, the geometric mean of GENWASYM’s speedups over WASP across all four benchmark groups is $29.4\times$.

8.2 RQ2: When Are Snapshot Reuse and Our Heuristic Effective?

Method. To evaluate snapshot reuse and our heuristic, we compare configurations of GENWASYM with these features enabled or disabled. In general, the net benefit of snapshot reuse depends on two factors: the execution it avoids and the cost of instantiating snapshots. Although the RQ1 benchmarks support a broad performance evaluation, they do not isolate this trade-off. We therefore use two controlled benchmarks (an arithmetic evaluator and quicksort) to vary these factors independently. For reference, we also report WASP’s performance.

The arithmetic-evaluator benchmark is parameterized by the input-expression AST size (n_{node} in Table 1) and the number of explored paths. Increasing the expression size increases the execution depth of each path and thus the number of executed instructions. Because path forking occurs late in execution in this benchmark, increasing the number of paths would increase sharing in the execution tree, allowing snapshot reuse to skip more common execution.

Quicksort may reorder array elements but does not construct new concrete or symbolic values. We exploit this property to control symbolic-state complexity through the AST sizes of the symbolic inputs. Each symbolic input can be initialized as either an atomic symbolic variable or a complex symbolic arithmetic expression. Larger expressions make snapshots more expensive to copy and instantiate. Table 1 shows the benchmark parameters: the syntactic complexity of symbolic inputs n_{comp} , the input array size n_{size} , and the number of symbolic elements n_{sym} . The remaining $n_{size} - n_{sym}$ elements are initialized with distinct concrete values. We define n_{comp} as the sum of the syntactic sizes of all symbolic values in the array. For example, storing a 32-bit symbolic value x as 4 bytes in the Wasm memory requires 4 SMT $extract_{hi,lo}(x)$ expressions which have size 2 each, so an atomic symbolic variable has size 8. We can deliberately construct symbolic states of arbitrary complexity by repeatedly applying arithmetic operations to the initial symbolic values.

Results. Table 1 shows the results. The columns $T_{exec}^{noreuse}$, $T_{exec}^{snapshot}$, and $T_{exec}^{heuristic}$ report instruction-execution times without snapshot reuse, with uniform snapshot reuse, and with heuristic snapshot reuse, respectively. We report three speedups to quantify the benefits of avoiding re-execution:

$S_{staging}$, the speedup of staging over the WASP baseline; $S_{snapshot}$, the speedup of *uniform* snapshot reuse over no-reuse; and $S_{heuristic}$, the speedup of *heuristic* snapshot reuse over no-reuse.

On the B-Tree and Collections-C benchmarks, both uniform and heuristic snapshot reuse yield modest average speedups of 1.1 $\times$, compared to GENWASYM with snapshot reuse disabled. Some configurations instead exhibit slight slowdowns, with $S_{snapshot}$ as low as 0.9 $\times$ on the Queue benchmark, and $S_{heuristic}$ as low as 0.9 $\times$ on the RingBuffer benchmark. Although snapshot reuse avoids re-executing shared prefixes, it does not always reduce the total execution time because the cost of creating and resuming snapshots is comparable to that of re-executing those prefixes.

On the arithmetic-evaluator benchmark, *heuristic* snapshot reuse achieves an average 4.8 $\times$ speedup over no-reuse. It also outperforms *uniform* snapshot reuse by an average of 1.7 $\times$. Although snapshot incur overhead, its better reuse decisions consistently reduce overall execution time.

The quicksort results exhibit a different pattern. With atomic symbolic expressions as inputs (e.g., $n_{comp} \leq 10^2$), uniform snapshot reuse remains beneficial because snapshots can bypass shared execution prefixes at negligible resumption cost. With more complex symbolic states (e.g., $n_{comp} > 10^4$), however, uniform reuse slows execution relative to no-reuse, yielding an average $S_{snapshot}$ of 0.6 $\times$. For short executions, the cost of copying and resuming snapshots containing large symbolic expressions outweighs the cost of re-executing the shared prefix. However, the heuristic effectively avoids these unfavorable cases, reusing snapshots only when the saved execution time justifies the overhead and yielding a 3.1 $\times$ speedup over no-reuse.

Summary. Uniform snapshot creation and restoration can outweigh the saved execution time, causing slowdowns in some cases. Our heuristic identifies these cases and makes more effective reuse decisions, consistently outperforming uniform snapshot reuse. Figure 10b shows the average speedups from snapshot reuse. Beyond staging’s 29.4 $\times$ speedup over WASP, uniform snapshot reuse adds 1.3 $\times$ speedup (i.e. 38.8 $\times$ vs WASP), and heuristics adds another 1.2 $\times$ speedup over uniform snapshot reuse (i.e., 44.9 $\times$ vs WASP).

8.3 RQ3: Is GENWASYM effective in finding real bugs?

Method. To evaluate the effectiveness of GENWASYM in detecting real bugs in practical programs, we conduct experiments on two *buggy programs* from Collections-C. These programs were previously used to evaluate the bug-finding effectiveness of WASP [35] and Gillian [44]. Both contain memory-safety violations that may cause memory corruption and thus pose potential security risks. For our evaluation, we compile the Collections-C test cases from C to WebAssembly and run concolic execution on the resulting Wasm modules. We also run WASP as a reference baseline for comparison.

Table 2. Bug detection results on the *buggy version* of Collections-C. Timings are measured in seconds.

Test	$n_{paths}^{GENWASYM}$	n_{paths}^{WASP}	Finds bug	$T_{exec}^{GENWASYM}$	$T_{total}^{GENWASYM}$	T_{exec}^{WASP}	T_{total}^{WASP}
array_test_remove	13	1	Yes	0.0153	0.2663	0.0042	0.0155
list_test_zipIterAdd	2028	1	Yes	3.71	12.2	0.0058	0.0058

Results. Table 2 shows that both GENWASYM and WASP detect the bugs. GENWASYM explores more paths and takes longer than WASP because they use different concolic execution strategies: WASP stops upon detecting a bug, whereas GENWASYM continues exploring more paths for higher code coverage. These results show that GENWASYM can effectively find real bugs in practical programs.

8.4 Threats to Validity

Generalizability of Snapshot-Reuse. In RQ2, we evaluated snapshot reuse and our heuristic using relatively small benchmarks with short execution times. These programs also do not contain

external calls. In the Section 6, we use the size of symbolic expressions to estimate the instantiation time of snapshot; it is an approximation of the actual instantiation time, which may be affected by other factors such as the structure of symbolic expressions and the underlying theory being used.

Unsupported Language Features. As a prototype, GENWASYM supports a substantial subset of Wasm, but not its full feature set. Applying GENWASYM to real-world programs may therefore require users to implement unsupported features, such as external calls and SIMD instructions.

Compilation Time. In the evaluation, we compared only the Wasm instruction-execution time for both GENWASYM and WASP. However, when a program's execution time is short, GENWASYM's compilation time may be comparable to or even longer than its execution time. When using GENWASYM in practice, one should weigh the compilation cost against the expected savings in execution time to ensure a positive overall benefit.

9 Discussion

Instrumentation vs. Staging. Instrumentation is another popular approach to implementing efficient concolic execution engines [38, 39, 60]. It is a source-to-source transformation that generates an instrumented program for performing concolic execution. However, instrumentation and staging differ in several respects when used to implement concolic execution engines.

To achieve efficiency without interpretation overhead, instrumentation requires the underlying target language to already have an efficient compiler-based implementation, such as C or LLVM IR. Existing commodity compiler infrastructure can then be leveraged to implement the instrumentation and generate instrumented code. In contrast, our approach is semantics-first: it starts from a concrete semantics and derives a concolic interpreter for the target language. The efficiency gain comes from eliminating interpretation overhead through staging. In this sense, we use staging to implement a compiler for a non-standard execution semantics.

One consequence of this difference is that instrumentation-based concolic execution preserves the original program's concrete execution, including its control flow and native state representation, whereas staging-based concolic execution must define these aspects explicitly in the semantics. The benefit is that staging-based concolic execution offers greater flexibility in customizing and manipulating the program's control and state, which is useful for implementing features such as snapshot reuse. Our approach takes advantage of this flexibility by representing the program's control as continuations at the meta-level, thereby enabling flexible snapshot reuse. In contrast, the reliance on the underlying language's concrete execution makes snapshot reuse challenging to implement by instrumentation, because it requires reifying the program's control and concrete states as manipulable data structures, which is not always straightforward or even possible.

In general, instrumentation can also be understood as partial evaluation [18]. Feigin and Mycroft [18] characterize the efficiency of instrumentation by drawing on ideas from Jones Optimality [29].

Adopt Our Approach in Other Languages/IRs. We instantiate our proposed architecture for Wasm, but the underlying methodology is language-independent. To adopt it for another language, the implementer can start from a concrete definitional interpreter in CPS for that language. Following the mechanical steps in Sections 3 and 4, the implementer can extend the concrete domains with their symbolic counterparts to obtain a concolic interpreter and then refactor it into a staged one.

Additionally, adopting the approach would entail building the necessary runtime support for state representation, symbolic operations, SMT solver interaction, and path scheduling. The generated code from specializing the concolic interpreter should work with the runtime support to perform concolic execution of the input program.

10 Related Work

Concolic Execution. Concolic execution [22, 45] is a white-box testing technique, which guides symbolic execution by concrete execution. You et al. [59] extend the idea of concolic execution to higher-order programs where functions are first-class values. Hybrid concolic execution [34] combines fuzzing and concolic testing, which has been resulted in various tools [48, 60]. Often conflated in the literature, KLEE [10] performs fork-based symbolic execution, in contrast to concolic execution tools such as CUTE [45], which would require concrete input seeds.

In the Wasm ecosystem, MANTICORE [36] and SEEWASM [25] are both Python-based symbolic execution engines, while OWI [3] supports parallel symbolic execution of Wasm programs. None of these tools supports concolic execution. To our knowledge, WASP Marques et al. [35] was the only concolic execution engine for Wasm, implemented by extending the official Wasm interpreter. All of these approaches rely on interpretation and therefore incur substantial interpretation overhead.

Reuse in Concolic Execution. QSYM [60] performs concolic execution via dynamic binary instrumentation, thus avoids the interpretation overhead over IRs compared to traditional interpretation-based engines. The authors of QSYM argue that its performant execution makes it unnecessary to use an inefficient snapshot mechanism to avoid re-execution, especially compared to previous tools such as Driller [48]. However, it is worth noting to explain the causes of Driller’s unsatisfying performance. Driller uses angr [46] as its underlying concolic execution engine. Angr first translates binary code into Valgrind’s VEX IR [37], which will be *interpreted* to perform concolic execution. The indirection layer of interpretation makes it easy to implement “snapshots”, but also contributes to the slowdown. This paper attempts to bring the best of two worlds, *i.e.*, fast execution via compilation and the ability to reuse symbolic states via continuations.

In hybrid concolic execution, a fuzzer produces new inputs via mutation. However, the correspondence between those inputs and the negation point is unknown to the concolic execution engine, thus preventing effective reuse.

Code Generation for Symbolic/Concolic Execution. Our approach achieves compilation via specializing a definitional concolic interpreter. A commonly used implementation strategy for concolic execution is referred as instrumentation, for example, QSYM [60] uses Intel’s Pin to perform instruction-level dynamic binary instrumentation, SYMCC [38] lifts the level of instrumentation to LLVM’s IR, and SYMQEMU [39] instruments over QEMU’s IR. Compared to our approach, all these instrumentation-based concolic engines lack the ability to control the execution and require re-execution. GENSYM [54, 56] uses staging to achieve efficient symbolic execution for LLVM IR and inspires our work, but does not support concolic execution.

11 Conclusion

In this paper, we presented a novel approach to efficient concolic execution by compiling a definitional concolic interpreter written in CPS using multi-stage programming. Our approach offers three key benefits. First, by expressing concolic execution as a definitional interpreter, we reduce the implementation effort compared to building a compiler from scratch. Second, leveraging multi-stage programming eliminates interpretation overhead, yielding highly efficient generated code. Third, representing control flow with continuations allows us to capture snapshots and avoid costly re-execution when exploring new paths. We further introduced a heuristic for deciding when to reuse continuations, which improves performance in practice. We implemented this approach in a prototype concolic-execution compiler GENWASYM for WebAssembly programs. Our evaluation showed that GENWASYM significantly outperforms prior systems.

Data-Availability Statement

The paper’s artifact is available at <https://doi.org/10.5281/zenodo.21517229> [61], which contains the implementation of GENWASYM, the benchmark suite used in the paper, and the evaluation scripts. The GENWASYM is publicly maintained at <https://github.com/Generative-Program-Analysis/GenSym/>.

Acknowledgments

We thank the reviewers of OOPSLA 2026 and PLDI 2026 for their meticulous reviews and constructive suggestions. We thank Léo Andrès for helpful discussions about Owi and WASP. We also would like to thank the members of the Tufts Programming Languages Group (TuPL) and the audience of the WebAssembly Workshop at OOPSLA/ICFP 2025 for their valuable feedback on this work.

References

- [1] [n. d.]. WebAssembly Core Specification. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.
- [2] Torben Amtoft. 1999. Partial evaluation for constraint-based program analyses. *BRICS Report Series* 6, 45 (1999). <https://doi.org/10.7146/brics.v6i45.20115>
- [3] Léo Andrès, Filipe Marques, Arthur Carcano, Pierre Chambart, José Fragoso Santos, and Jean-Christophe Filliâtre. 2024. Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly. *CoRR* abs/2412.06391 (2024). <https://doi.org/10.48550/ARXIV.2412.06391>
- [4] Andrew W Appel. 1992. *Compiling with Continuations*. Cambridge University Press. <https://doi.org/10.1017/CBO9780511609619>
- [5] Thomas Ball and Jakub Daniel. 2015. Deconstructing Dynamic Symbolic Execution. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 26–41. <https://doi.org/10.3233/978-1-61499-495-4-26>
- [6] Clark Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. 2021. Satisfiability modulo theories. In *Handbook of satisfiability*. IOS Press, 1267–1329. <https://doi.org/10.3233/FAIA201017>
- [7] Dominique Boucher and Marc Feeley. 1996. Abstract Compilation: A New Implementation Paradigm for Static Analysis. In *CC (Lecture Notes in Computer Science, Vol. 1060)*. Springer, 192–207. https://doi.org/10.1007/3-540-61053-7_62
- [8] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT - a formal system for testing and debugging programs by symbolic execution. In *Reliable Software*. ACM, 234–245. <https://doi.org/10.1145/800027.808445>
- [9] Oliver Bracevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. 2023. Graph IRs for Impure Higher-Order Languages: Making Aggressive Optimizations Affordable with Precise Effect Dependencies. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 400–430. <https://doi.org/10.1145/3622813>
- [10] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX Association, 209–224.
- [11] 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>
- [12] Lori A. Clarke. 1976. A program testing system. In *ACM Annual Conference*. ACM, 488–491. <https://doi.org/10.1145/80191.805647>
- [13] Charles Consel and Olivier Danvy. 1991. For a Better Support of Static Data Flow. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 523)*, John Hughes (Ed.). Springer, 496–519. https://doi.org/10.1007/3540543961_24
- [14] Daniel Damian. 1999. Partial evaluation for program analysis. *Progress report, BRICS PhD School, University of Aarhus* (1999).
- [15] Olivier Danvy and Kevin Millikin. 2009. Refunctionalization at work. *Sci. Comput. Program.* 74, 8 (2009), 534–549. <https://doi.org/10.1016/J.SCICO.2007.10.007>
- [16] Olivier Danvy and Lasse R. Nielsen. 2001. Defunctionalization at Work. In *PPDP*. ACM, 162–174. <https://doi.org/10.1145/773184.773202>
- [17] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
- [18] Boris Feigin and Alan Mycroft. 2010. Formally Efficient Program Instrumentation. In *RV (Lecture Notes in Computer Science, Vol. 6418)*. Springer, 245–252. https://doi.org/10.1007/978-3-642-16612-9_19
- [19] Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Computers, Controls* 25 (1971), 45–50.

- [20] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *High. Order Symb. Comput.* 12, 4 (1999), 381–391. <https://doi.org/10.1023/A:1010095604496>
- [21] Jeremy Gibbons. 2022. Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity. *Art Sci. Eng. Program.* 6, 2 (2022), 7. <https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2022/6/7>
- [22] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI*. ACM, 213–223. <https://doi.org/10.1145/1065010.1065036>
- [23] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*. The Internet Society.
- [24] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *PLDI*. ACM, 185–200. <https://doi.org/10.1145/3062341.3062363>
- [25] Ningyu He, Zhehao Zhao, Hanqin Guan, Jikai Wang, Shuo Peng, Ding Li, Haoyu Wang, Xiangqun Chen, and Yao Guo. 2024. SeeWasm: An Efficient and Fully-Functional Symbolic Execution Engine for WebAssembly Binaries. In *ISSTA*. ACM, 1816–1820. <https://doi.org/10.1145/3650212.3685300>
- [26] Carsten Kehler Holst and Carsten K. Gomard. 1991. Partial Evaluation is Fuller Laziness. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, Connecticut, USA, June 17-19, 1991*, Charles Consel and Olivier Danvy (Eds.). ACM, 223–233. <https://doi.org/10.1145/115865.115890>
- [27] William E. Howden. 1977. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Trans. Software Eng.* 3, 4 (1977), 266–278. <https://doi.org/10.1109/TSE.1977.231144>
- [28] Neil D. Jones. 1996. What Not to Do When Writing an Interpreter for Specialisation. In *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers (Lecture Notes in Computer Science, Vol. 1110)*, Olivier Danvy, Robert Glück, and Peter Thiemann (Eds.). Springer, 216–237. https://doi.org/10.1007/3-540-61580-6_11
- [29] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Prentice Hall.
- [30] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. <https://doi.org/10.1145/360248.360252>
- [31] 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. https://doi.org/10.1007/978-3-319-07151-0_6
- [32] Julia L. Lawall and Olivier Danvy. 1994. Continuation-Based Partial Evaluation. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27-29 June 1994*, Robert R. Kessler (Ed.). ACM, 227–238. <https://doi.org/10.1145/182409.182483>
- [33] David Lion, Adrian Chiu, Michael Stumm, and Ding Yuan. 2022. Investigating Managed Language Runtime Performance: Why JavaScript and Python are 8x and 29x slower than C++, yet Java and Go can be Faster?. In *USENIX ATC*. USENIX Association, 835–852.
- [34] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *ICSE*. IEEE Computer Society, 416–426. <https://doi.org/10.1109/ICSE.2007.41>
- [35] Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão. 2022. Concolic Execution for WebAssembly. In *ECOOP (LIPIcs, Vol. 222)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:29. <https://doi.org/10.4230/LIPICS.ECOOP.2022.11>
- [36] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *ASE*. IEEE, 1186–1189. <https://doi.org/10.1109/ASE.2019.00133>
- [37] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*. ACM, 89–100. <https://doi.org/10.1145/1250734.1250746>
- [38] Sebastian Poehlau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *USENIX Security Symposium*. USENIX Association, 181–198.
- [39] Sebastian Poehlau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *NDSS*. The Internet Society. <https://doi.org/10.14722/ndss.2021.24118>
- [40] Juan Pedro Bolívar Puente. 2017. Persistence for the masses: RRB-vectors in a systems language. *Proc. ACM Program. Lang.* 1, ICFP (2017), 16:1–16:28. <https://doi.org/10.1145/3110260>
- [41] John C. Reynolds. 1972. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2*. <https://doi.org/10.1145/800194.805852>
- [42] John C. Reynolds. 1993. The Discoveries of Continuations. *LISP Symb. Comput.* 6, 3-4 (1993), 233–248. <https://doi.org/10.1007/BF01019459>
- [43] Tiark Rumpf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*. ACM, 127–136. <https://doi.org/10.1145/1868294.1868314>

- [44] José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, part i: a multi-language platform for symbolic execution. In *PLDI*. ACM, 927–942. <https://doi.org/10.1145/3385412.3386014>
- [45] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*. ACM, 263–272. <https://doi.org/10.1145/1081706.1081750>
- [46] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Fimalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS*. The Internet Society. <https://doi.org/10.14722/ndss.2015.23294>
- [47] srdja. n.d.. Collections-C. <https://github.com/srdja/Collections-C>. GitHub repository.
- [48] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. <https://doi.org/10.14722/ndss.2016.23368>
- [49] Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. Multi-stage programming with generative and analytical macros. In *GPCE*. ACM, 110–122. <https://doi.org/10.1145/3486609.3487203>
- [50] Walid Taha. 1999. *Multi-stage programming: Its theory and applications*. Ph. D. Dissertation. Oregon Graduate Institute of Science and Technology. <https://doi.org/10.1145/258993.259019>
- [51] Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *PEPM*. ACM, 203–217. <https://doi.org/10.1145/258993.259019>
- [52] Conrad Watt, Maja Trela, Peter Lammich, and Florian Märkl. 2023. WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly. *Proc. ACM Program. Lang.* 7, PLDI (June 2023). <https://doi.org/10.1145/3591224>
- [53] Guannan Wei, Alexander Y. Bai, Dinghong Zhong, and Jiatai Zhang. 2025. Reconstructing Continuation-Passing Semantics for WebAssembly. In *TFP (Lecture Notes in Computer Science, Vol. 15652)*. Springer, 185–204. https://doi.org/10.1007/978-3-031-99751-8_9
- [54] Guannan Wei, Oliver Bračevac, Shangyin Tan, and Tiark Rompf. 2020. Compiling symbolic execution with staging and algebraic effects. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 164:1–164:33. <https://doi.org/10.1145/3428232>
- [55] 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 (2019), 126:1–126:32. <https://doi.org/10.1145/3360552>
- [56] Guannan Wei, Songlin Jia, Ruiqi Gao, Haotian Deng, Shangyin Tan, Oliver Bračevac, and Tiark Rompf. 2023. Compiling Parallel Symbolic Execution with Continuations. In *ICSE*. IEEE, 1316–1328. <https://doi.org/10.1109/ICSE48619.2023.00116>
- [57] Clark Wiedmann. 1983. A performance comparison between an APL interpreter and compiler. In *APL*. ACM, 211–217. <https://doi.org/10.1145/800062.801219>
- [58] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *PLDI*. ACM, 662–676. <https://doi.org/10.1145/3062341.3062381>
- [59] Shu-Hung You, Robert Bruce Findler, and Christos Dimoulas. 2021. Sound and Complete Concolic Testing for Higher-order Functions. In *ESOP (Lecture Notes in Computer Science, Vol. 12648)*. Springer, 635–663. https://doi.org/10.1007/978-3-030-72019-3_23
- [60] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*. USENIX Association, 745–761.
- [61] Dinghong Zhong, Alexander Bai, Mikail Khan, and Guannan Wei. 2026. *Compiling WebAssembly Concolic Execution with Staging, Continuations, and Snapshots (Artifact)*. <https://doi.org/10.5281/zenodo.21517229>
- [62] Dinghong Zhong, Alexander Bai, Mikail Khan, and Guannan Wei. 2026. Compiling WebAssembly Concolic Execution with Staging, Continuations, and Snapshots (Extended Version). <https://doi.org/10.48550/ARXIV.2608.18327> arXiv:2608.18327 [cs.PL]

Received 2026-03-18; accepted 2026-08-06