A Formal Foundation for Symbolic Evaluation with Merging

SORAWE PORNCHAROENWASE, University of Washington, USA
LUKE NELSON, University of Washington, USA
XI WANG, University of Washington, USA
EMINA TORLAK, University of Washington, USA

Reusable symbolic evaluators are a key building block of solver-aided verification and synthesis tools. A reusable evaluator reduces the semantics of all paths in a program to logical constraints, and a client tool uses these constraints to formulate a satisfiability query that is discharged with SAT or SMT solvers. The correctness of the evaluator is critical to the soundness of the tool and the domain properties it aims to guarantee. Yet so far, the trust in these evaluators has been based on an ad-hoc foundation of testing and manual reasoning.

This paper presents the first formal framework for reasoning about the behavior of reusable symbolic evaluators. We develop a new symbolic semantics for these evaluators that incorporates state merging. Symbolic evaluators use state merging to avoid path explosion and generate compact encodings. To accommodate a wide range of implementations, our semantics is parameterized by a symbolic factory, which abstracts away the details of merging and creation of symbolic values. The semantics targets a rich language that extends Core Scheme with assumptions and assertions, and thus supports branching, loops, and (first-class) procedures. The semantics is designed to support reusability, by guaranteeing two key properties: legality of the generated symbolic states, and the reducibility of symbolic evaluation to concrete evaluation. Legality makes it simpler for client tools to formulate queries, and reducibility enables testing of client tools on concrete inputs. We use the Lean theorem prover to mechanize our symbolic semantics, prove that it is sound and complete with respect to the concrete semantics, and prove that it guarantees legality and reducibility.

To demonstrate the generality of our semantics, we develop Leanette, a reference evaluator written in Lean, and Rosette 4, an optimized evaluator written in Racket. We prove Leanette correct with respect to the semantics, and validate Rosette 4 against Leanette via solver-aided differential testing. To demonstrate the practicality of our approach, we port 16 published verification and synthesis tools from Rosette 3 to Rosette 4. Rosette 4 is an existing reusable evaluator that implements the classic merging semantics, adopted from bounded model checking. Rosette 4 replaces the semantic core of Rosette 3 but keeps its optimized symbolic factory. Our results show that Rosette 4 matches the performance of Rosette 3 across a wide range of benchmarks, while providing a cleaner interface that simplifies the implementation of client tools.

CCS Concepts: • Software and its engineering → Correctness; Formal methods; Semantics.

Additional Key Words and Phrases: symbolic evaluation, state merging

ACM Reference Format:
1 INTRODUCTION

Symbolic evaluation is a core component of solver-aided tools, which automate program verification and synthesis tasks by reducing them to satisfiability solving. These tools work by embedding an interpreter for the tool’s source language into a host language that is equipped with a reusable symbolic evaluator [Sen et al. 2015; Torlak and Bodik 2014; Uhler and Dave 2014]. When applied to a source program and symbolic inputs, the embedded interpreter is executed by the host’s reusable evaluator to produce a logical encoding of the program’s semantics. The tool then uses the resulting encoding to express a verification or synthesis task as a logical satisfiability query, solved with a SAT or SMT solver. This pipeline produces correct results if the tool generates the right query, and both the evaluator and the solver are correct. In practice, tool developers focus on testing or verifying their use of the symbolic evaluator [Weitz et al. 2017], and trust the evaluator and the solver to be correct. The trust in solvers is based on decades of community investment in their testing [Winterer et al. 2020], validation [Cruz-Filipe et al. 2017], and verification [Blanchette et al. 2017]. But the trust in reusable evaluators rests on a weaker foundation of ad-hoc testing and manual inspection.

This paper presents the first formal framework for reasoning about the behavior of reusable evaluators. We develop a new symbolic semantics for reusable evaluators, which we call $\mathcal{S}_c$, and we prove that it is sound and complete with respect to the underlying concrete semantics. The framework targets a small but expressive language, $\lambda_c$, that extends Core Scheme [Flanagan et al. 1993] with assumptions and assertions. As such, $\lambda_c$ includes the core features supported by existing reusable evaluators: branching, loops, (first-class) procedures, and specification constructs. Our symbolic semantics for $\lambda_c$ incorporates state merging [Biere et al. 1999; Clarke et al. 2004, 2003], which is key to generating small (polynomially sized) encodings. Unlike the classic merging semantics [Biere et al. 1999; Clarke et al. 2004, 2003], which was developed for verification of loop-free code, $\mathcal{S}_c$ is designed to be reused by a wide range of tools, on a wide range of programs. It maintains strong invariants on the formulas that characterize the symbolic state, and on termination of halted paths. The former simplifies the formulation of queries, and the latter ensures that symbolic evaluation terminates as often as concrete execution on concrete inputs, which is vital for the development and testing of client tools. We prove these reusability properties and the correctness of $\mathcal{S}_c$ using the Lean theorem prover [de Moura et al. 2015].

Our framework aims to provide a general contract for implementing and validating reusable evaluators. To meet this goal, it must accommodate a wide range of implementations, which is uniquely challenging for reusable evaluators. To see why, consider the toy verification query in Figure 1a. The query verifies that the unsigned value of an $n$-bit integer $x$ exceeds the number of 1’s in its binary representation. Figure 1b shows the symbolic execution tree [Clarke 1976; King 1976] for this query when $n = 2$. The tree captures all feasible concrete executions of our program, with the feasibility of each branch decided by an SMT solver. Because of this feasibility check, all implementations of symbolic execution will (in principle) generate an isomorphic tree. But reusable evaluators do not necessarily call the solver during evaluation, so their behavior depends on their strategy for constructing symbolic values, as illustrated in Figure 1c. If the evaluator deduces that right-shifting $x$ two or more times produces 0, it will terminate and produce a finite DAG. Otherwise, it will diverge, producing an infinite DAG. A general semantics for reusable evaluators must account for all of these behaviors.

We address this challenge by defining $\mathcal{S}_c$ with respect to a symbolic factory, and proving that it is partially correct with respect to the concrete semantics of $\lambda_c$. A symbolic factory is a parameter to the semantics, consisting of functions that abstract away the details of creation, simplification, and merging of symbolic values. Our framework specifies only what it means for these functions to be sound; their implementation is otherwise opaque. For example, all outcomes shown in Figure 1c can be produced by a sound factory. To account for non-termination, our correctness criterion...
1 (define (addbits x s) ; If \( x = 0 \)
2  (if (bvzero? x) ; then \( s \)
3       s ; else addbits
4  (addbits (bvshl x (bv 1 n))) ; \((x \ll 1)\)
5  (bvadd (bvand x (bv 1 n)) s)))) ; \(((x \& 1) + s)\).
6
7 (define (popcount x) ; Returns the number of 1 bits
8  (addbits x (bv 0 n))) ; in the n-bit bitvector \( x \).
9
10 (define-symbolic x (bitvector n)) ; Symbolic n-bit bitvector.
11
12 (verify ; Verify that for every n-bit
13  (assert ; bitvector \( x \), \( x \geq_\text{u} \) (popcount x)),
14  (bvuge x (popcount x)))) ; where \( \geq_\text{u} \) is unsigned comparison.

(a) A toy program with a verification query.

(b) Symbolic execution tree for \( n = 2 \).

(c) Symbolic evaluation DAGs for \( n = 2 \).

Fig. 1. A toy program with a verification query (a), along with the symbolic execution tree (b) and symbolic evaluation DAGs (c) for this query. Nodes represent symbolic states. Edges represent guarded transitions between states. Dotted edges denote omitted parts of the graph. Circed numbers refer to lines in the toy program.

specifies what it means for a reusable evaluator to produce a sound and complete result if the evaluation terminates. All the DAGs shown in Figure 1c are correct according to our definition, with the infinite one satisfying the definition trivially. For programs that are free of loops, our semantics terminates and is totally correct with respect to all sound symbolic factories.

In addition to providing a general contract for reusable evaluators, our framework also aims to expose a practical interface to their client tools. We identify two key properties of practical symbolic evaluators, reducibility and legality, and we design our symbolic semantics so that every (correct) implementation of its rules satisfies these properties.

Reducibility states that symbolic evaluation terminates on a given program and a fully concrete input whenever the concrete semantics does so. This property is trivially satisfied by all symbolic evaluators on loop-free programs, but only symbolic execution is designed to reduce to concrete execution in the presence of loops. Our semantics integrates reducibility in its design as well, by including a mechanism for abandoning halted paths as soon as possible. If an assertion or assumption fails during symbolic evaluation, \( S_e \) requires the evaluator to abandon that path of execution and propagate the failure upstream, similarly to how an exception is propagated in concrete execution. This mechanism forces the evaluator to mirror the concrete semantics on fully concrete inputs, which enables the testing of client tools (e.g., [Nelson et al. 2019, 2020]). Beyond testing, this mechanism also enables the use of unwinding assertions [Clarke et al. 2004] to bound loops in the presence of symbolic inputs (see, e.g., Figure 11 of [Torlak and Bodik 2013]).
Legality simplifies the reuse of the evaluator’s output, which takes the form of symbolic states. In all forms of symbolic evaluation, the symbolic state \( \sigma \) is characterized by two formulas, which the client tools use to construct their queries. One encodes the assumptions, \( \text{assumes}(\sigma) \), and the other encodes the assertions, \( \text{asserts}(\sigma) \), that have been made to reach the state \( \sigma \). Intuitively, at least one of these formulas should always be true, because there is no concrete execution in which both an assumption and an assertion fail: the execution stops as soon as the first failure is reached. Yet in the classic merging semantics, both formulas can become false, and client tools must account for this when generating queries to avoid unsound results (see Section 4.3). To address this problem, our semantics computes \( \text{assumes}(\sigma) \) and \( \text{asserts}(\sigma) \) so that every state \( \sigma \) generated during symbolic evaluation is legal—i.e., every model makes at least one of the formulas true. With legality, client tools can adopt a simple interpretation of symbolic states, inherent in symbolic execution, where the validity of the formula \( \text{assumes}(\sigma) \rightarrow \text{asserts}(\sigma) \) ensures the absence of errors.

To demonstrate the suitability of our framework for implementing and validating reusable evaluators, we implement \( S_c \) in two different languages. The first implementation is a reference interpreter written in Lean, using a naïve symbolic factory. We use Lean to prove that the reference interpreter implements \( S_c \), and that it satisfies our correctness criterion. The second implementation is an optimized evaluator for Rosette [Torlak and Bodik 2014], an existing language with a reusable symbolic evaluator that hosts a variety of verification and synthesis tools. We refer to our new evaluator as Rosette 4. Rosette’s default evaluator is based on the classic merging semantics, and we refer to it as Rosette 3. Both evaluators use the same optimized symbolic factory. We validate Rosette 4 against the Lean reference interpreter using differential testing, aided by an SMT solver. We find that their behaviors match on 10,000 automatically generated test programs.

To evaluate the practical impact of using \( S_c \), we port 16 published verification and synthesis tools to Rosette 4, and compare the performance of the ported code to the original code built on top of Rosette 3. Our benchmarks include the 15 tools studied in prior work [Bornholt and Torlak 2018] on profiling the performance of reusable evaluators, and a recent system, Jitterbug [Nelson et al. 2020], for verifying and synthesizing just-in-time compilers in the Linux kernel. Our results show that Rosette 4 is up to 14× faster than Rosette 3 across all benchmarks. The largest slowdown we observe is 6×. On average, Rosette 4 is 10–20% faster than Rosette 3. Our evaluation also shows that the legality-preserving interface exposed by Rosette 4 simplifies the implementation of our most complex benchmark, Jitterbug. The original implementation of Jitterbug relied on custom code for tracking assumptions, which required careful manual reasoning from the developers to ensure its correctness. The ported implementation removes this custom code, leading to code that is simpler and easier to understand.

In summary, this paper makes the following contributions:

- The first formal framework for reasoning about symbolic evaluation with merging. We develop a new parametric merging semantics for an expressive core language; we prove that our semantics is sound and complete; and we prove that it preserves legality and reducibility.
- A mechanization of this framework in the Lean theorem prover.
- Two implementations of our semantics, one in Lean and one in Rosette. We prove the Lean implementation correct against our semantics, and we use solver-aided differential testing to show the Rosette implementation matches the Lean implementation.
- An evaluation of the Rosette implementation on 16 tools for program verification and synthesis developed in prior work. Our evaluation shows that this new implementation offers better performance and a cleaner interface than Rosette’s default symbolic evaluator.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces our target language and illustrates basic applications of a reusable evaluator for this language.
Section 4 presents the \( S_c \) merging semantics and compares it to the classic one. Section 5 states our correctness criterion and shows that \( S_c \) satisfies it. Section 6 describes our Lean and Rosette implementations of \( S_c \), proofs, and differential testing results. Section 7 evaluates the utility and performance of the Rosette implementation. Section 8 concludes the paper.

2 RELATED WORK

Reusable symbolic evaluators [Sen et al. 2015; Torlak and Bodik 2014; Uhler and Dave 2014] are designed to serve as platforms for building new tools. They reduce programs to constraints via a combination of symbolic execution [Clarke 1976; King 1976] and bounded model checking [Biere et al. 1999]. Symbolic execution encodes each path through a program separately, giving rise to a potentially infinite symbolic execution tree. Each node in this tree is a symbolic state, which represents a set of concrete program states, and each path represents a set of feasible concrete paths. Symbolic execution is well-understood and has been formalized for imperative languages by recognizing a correspondence between concrete and symbolic paths in the absence of state merging [Lucanu et al. 2017]. Prior work [Nguyên et al. 2017] has also formalized symbolic execution for a language that extends lambda calculus with numbers and contract constructs, similarly to our work.

Bounded model checking uses state merging to optimize symbolic execution of loop-free programs. It merges symbolic states from different paths at each control-flow join, giving rise to a DAG that is asymptotically smaller than the corresponding symbolic execution tree. Reusable evaluators extend state merging to programs with loops. Because of this extension, their behavior falls outside of the well-understood semantics for both bounded model checking, which assumes loop-free programs, and symbolic execution, which assumes path-based evaluation.

In contrast to the classic merging semantics from bounded model checking, \( S_c \) targets programs with loops, has a mechanized proof of soundness and completeness, and preserves legality and reducibility. These features are designed to facilitate reuse. Legality helps developers reuse the evaluator’s output to formulate custom queries, while reducibility helps developers test their tools. For example, if a tool targets a language with a concrete reference implementation (e.g., a CPU emulator for an ISA), reducibility makes it possible to test the tool’s modeling of the reference semantics on concrete programs and inputs [Nelson et al. 2019].

GL [Swords 2010] is a related effort that provides verified implementations of symbolic evaluators in the ACL2 theorem prover. GL is used to automate proofs within ACL2, and uses binary-decision diagrams (BDDs) to represent symbolic expressions. Given some symbolic input, the symbolic evaluator computes the truth value of a theorem as a BDD; if the BDD is the constant ‘true’, then the theorem can be shown to hold via the correctness of the symbolic evaluator. GL includes two methods of symbolic evaluation: one is a compiler that transforms concrete ACL2 functions into symbolic equivalents, and the other is an interpreter over the ACL2 syntax. Both can be parameterized with a set of primitive functions, and both merge values at control-flow joins. This merging algorithm is fixed in GL, as is the shape of the generated symbolic evaluation DAG. GL uses the classic merging semantics and forces termination through the use of fuel.

Compared to GL, our framework is more general. First, \( S_c \) is defined with respect to a symbolic factory interface, which is not restricted to a specific merging algorithm or underlying representation of symbolic values. Second, our target language is a strict superset of the GL language. To support reuse, it includes first-class procedures, assumptions, and assertions, which are missing from GL. Finally, \( S_c \) is formalized as a big-step operational semantics, via an inductive predicate in Lean, so \( S_c \) evaluators do not need to use fuel to bound executions, whereas GL evaluators do. The main disadvantage of a fuel-based semantics is that it forces implementations to track the fuel value. This tracking can be difficult to implement for evaluators that are realized as a shallow
embedding in a host language. Such an evaluator is based on the host’s unbounded interpreter, and it usually has no easy way to bound the host interpreter’s behavior.

3 λc: A CORE LANGUAGE FOR REUSABLE SYMBOLIC EVALUATORS

This section presents λc, a core language for reusable symbolic evaluators. We begin by describing the syntax, concrete semantics, and key features of λc. We then define what it means for a λc program to execute normally and to be correct. Finally, we use these definitions to formalize the angelic execution [Bodik et al. 2010] and verification queries for λc programs. Solving these queries is the core computational task performed by solver-aided tools. For example, many synthesis tools are based on the CEGIS algorithm [Solar-Lezama et al. 2006], which combines a demonic verifier and an angelic guesser to solve synthesis queries. Sections 4 and 5 will describe how to solve these queries using our symbolic semantics for λc.

3.1 The Syntax and Concrete Semantics of λc

Figure 2 shows the syntax of λc. The language extends Core Scheme [Flanagan et al. 1993] with the ability to express assumptions and assertions using the (error) and (abort) expressions. Intuitively, (error) is equivalent to asserting false, and (abort) is equivalent to assuming false. Combined with conditionals, these constructs can be used to encode arbitrary assertions and assumptions over λc expressions. Compared to Core Scheme, we place a light restriction on the syntax of λc by requiring the arguments to procedures and primitive operators to be variables. This restriction simplifies our formalization and proofs without sacrificing expressiveness. In particular, every Core Scheme program can be converted to an equivalent λc program by using let expressions; e.g., the application expression (e1 e2) becomes (let (x1 e1) (let (x2 e2) (x1 x2))). Like Core Scheme, the syntax of λc is parameterized by the set of primitive values (D) and primitive operators (O). Booleans and procedures are the only constants fixed by the language.

Figure 3 shows our big-step operational semantics for λc. The judgment \( \langle e, E \rangle \Downarrow r \) states that the expression \( e \) produces the result \( r \) when evaluated in the environment \( E \). The environment \( E \in E \) is a finite map from variables to values. A value \( v \in V \) is a boolean constant, a primitive constant from the set \( D \), or a closure that combines a procedure with an environment.\(^1\) The result \( r \in R \) is Ans\( (v) \) if the evaluation terminates normally and produces the value \( v \); Err if it errors; and Abt if it aborts.

The evaluation rules for λc are standard. Atomic expressions terminate and produce the expected results (LITERAL, CLOSURE, ERROR, and ABORT). Operator calls (CALLOP) also terminate, producing a value, aborting, or erroring. Their meaning is given by the function \( op : O \rightarrow V^* \rightarrow R \), which is a parameter to the semantics of λc. Unlike operator calls, procedure calls need not terminate. In particular, a procedure call \( (x_1 x_2) \) evaluates both of its arguments and checks if \( x_1 \) is bound to a closure \( \langle cl x, e, E_1 \rangle \). If not, the call errors (CALLBAD). Otherwise, the call evaluates \( e \) in the environment \( E_1[x \mapsto v_2] \), which binds \( x \) to the value of \( x_2 \) in \( E_1 \). The result of this evaluation, if any, is the result of the call (CALL). The rules for evaluating let expressions are similar (LET and LETHALT). Finally, a conditional expression \( (if \ x \ e_1 e_2) \) evaluates to the result of \( e_1 \) if \( x \) is not bound

\(^1\)Our formalization also includes lists but we omit them from the presentation for brevity.
The program, \( \lambda_c \), consists of the operators \{−, <, =\}, and \( op : O \to \mathbb{V}^* \to R \), which gives meaning to the operators \( o \in O \). The notation \( \mathbb{V}^* \) stands for a sequence of values.

to \#f\), and otherwise, it evaluates to \( e_2 \) (IfTrue and IfFalse). This semantics mirrors that of Core Scheme: conditionals treat every value except \#f as 'true'.

While the syntax and semantics of \( \lambda_c \) are largely standard, one difference from prior work is worth noting. The language places no restrictions on the use of variable names, so a program can include free variables. Prior work on verified compilation [Chlipala 2010] prevented this by using parametric higher-order abstract syntax. We intentionally allow programs to contain free variables, and treat them as inputs to the program, supplied by the environment. The semantics gets stuck if the environment does not provide a variable referenced during execution (VARIABLE). Treating free variables as inputs lets us introduce symbolic values into a program without having to include a dedicated syntactic construct for creating symbolic values—and having to specify some concrete semantics for it.

The semantics of \( \lambda_c \) is deterministic (Theorem 1), like that of Core Scheme. Given a program and an environment, it either diverges or produces a unique result. In the rest of this paper, when we write about the result of a program, we mean this unique result, if one exists.

**Theorem 1.** Evaluating an expression from the same environment produces the same result:

\[
\forall e, E, r_1, r_2. \ (\langle e, E \rangle \downarrow r_1 \land \langle e, E \rangle \downarrow r_2) \to r_1 = r_2.
\]

**Example 1.** To illustrate the syntax and semantics of \( \lambda_c \), consider an instantiation \( \lambda_2 \) where \( D \) is the set of all integers, \( O \) consists of the operators \{−, <, =\}, and \( op \) gives these operators their standard meaning over integers. Using this instantiation, we can write the following program:

```plaintext
1  (let (z 0) ; z is zero
2  (let (abs (λ x . (let (b0 (< x z)) (if b0 (- x) x))))
3  (let (b1 (= y z))
4  (let (_, (if b1 (abort) #t)) ; assume y = 0
5  (let (y0 (abs y))
6  (let (b2 (< z y0))
7  (let (_, (if b2 #t (error))) ; assert |y| > 0
8  _))))))
```

The program, \( \varepsilon_{abs} \), asserts that the absolute value of its free variable \( y \) is positive, assuming that \( y \) is not 0. The evaluation of this program gets stuck in all environments that have no binding for \( y \). We also have \( \langle \varepsilon_{abs}, \{y \mapsto -1\} \rangle \downarrow \mathrm{Ans}(\#t) \) and \( \langle \varepsilon_{abs}, \{y \mapsto 0\} \rangle \downarrow \mathrm{Abt} \).


3.2 The Angelic Execution and Verification Queries for $\lambda_c$

Given the syntax and semantics of $\lambda_c$, we define what it means for an execution to be normal and for a program to be correct (Definitions 1 and 2). A program executes normally in a given environment if it terminates and produces $\text{Ans}(v)$ for some value $v \in V$. The execution errors if it produces $\text{Err}$. If a program does not error in any environment $E \in \mathcal{E}$, we say that it is correct with respect to the input space defined by the set of environments $\mathcal{E}$.

**Definition 1 (Executions of $\lambda_c$ programs).** A program $e$ evaluates normally in an environment $E$ iff $\langle e, E \rangle \downarrow \text{Ans}(v)$ for some value $v \in V$; it errors iff $\langle e, E \rangle \downarrow \text{Err}$; and it aborts iff $\langle e, E \rangle \downarrow \text{Abt}$. We denote these outcomes with $\text{normal}(e, E)$, $\text{errors}(e, E)$, or $\text{aborts}(e, E)$, respectively.

**Definition 2 (Correctness of $\lambda_c$ programs).** A program $e$ is correct with respect to the set of environments $\mathcal{E}$ iff it does not error in any $E \in \mathcal{E}$, i.e., $\text{correct}(e, \mathcal{E}) := \forall E \in \mathcal{E}. \neg \text{errors}(e, E)$.

These two definitions underlie the core computational tasks performed by solver-aided tools: angelic execution and verification. Both tasks, which we call queries, can be understood as forms of search. Angelic execution searches for a normal execution of a given program, while verification searches for an execution that errors. We formalize both (Definitions 3 and 4) as partial functions from a program and a set of environments to an environment that satisfies the query or $\text{unsat}$ if the query is unsatisfiable. The functions are partial because the two queries may not terminate in general. But if they terminate, they must produce a correct result—i.e., we take them to be sound semi-decision procedures for the angelic execution and verification problems.

**Definition 3 (Angelic execution).** Given a program $e$ and set of environments $\mathcal{E}$, the angelic execution query $\text{guess}(e, \mathcal{E})$ diverges or produces one of two results: either an environment $E \in \mathcal{E}$ such that $\text{normal}(e, E)$, or $\text{unsat}$ if no such environment exists in $\mathcal{E}$.

**Definition 4 (Verification).** Given a program $e$ and set of environments $\mathcal{E}$, the verification query $\text{verify}(e, \mathcal{E})$ diverges or produces one of two results: either an environment $E \in \mathcal{E}$ such that $\text{errors}(e, E)$, or $\text{unsat}$ if no such environment exists in $\mathcal{E}$.

Example 2. To illustrate Definitions 1–4, consider the program $e_{\text{abs}}$ from Example 1. This program is correct with respect to $\mathcal{E}_Z$, the set of all environments that bind $y$ to an integer. It also has infinitely many normal executions with respect to this set. As a result, $\text{verify}(e_{\text{abs}}, \mathcal{E}_Z) = \text{unsat}$, and $\text{guess}(e_{\text{abs}}, \mathcal{E}_Z)$ may return any environment that binds $y$ to a non-zero value, e.g., $\{y \mapsto -1\}$. But $e_{\text{abs}}$ is not correct with respect to $\mathcal{E}_V$, the set of all environments that bind $y$ to any value. In particular, it will error at line 2 in every environment that binds $y$ to a non-integer value, since the equality operator $=$ expects its inputs to be integers. The verification query $\text{verify}(e_{\text{abs}}, \mathcal{E}_V)$ may return any of these environments, e.g., $\{y \mapsto \#t\}$. Since every environment that is in $\mathcal{E}_V$ but not in $\mathcal{E}_Z$ leads to an error, $\text{guess}(e_{\text{abs}}, \mathcal{E}_V)$ and $\text{guess}(e_{\text{abs}}, \mathcal{E}_Z)$ draw their outputs from the same non-empty subset of $\mathcal{E}_Z$. The next section shows how to automate these queries using our symbolic semantics for $\lambda_c$.

4 $S_c$: A SEMANTICS FOR SYMBOLIC EVALUATION WITH MERGING

To automate angelic execution and verification, solver-aided tools rely on symbolic evaluation to express the $\text{guess}(e, \mathcal{E})$ and $\text{verify}(e, \mathcal{E})$ queries as logical formulas. This section presents $S_c$, a new symbolic semantics for reducing $\lambda_c$ programs to formulas. To accommodate a wide range of practical implementations, $S_c$ is parameterized by a *symbolic factory*, which is a set of abstract functions for creating and manipulating *symbolic values*. We start by formalizing these notions and relating them to our concrete semantics. Next, we present the symbolic evaluation rules for $S_c$, and contrast them to the classic merging semantics [Biere et al. 1999]. We conclude this section.
Symbolic values \( \hat{v} \in \hat{V} \) \( \llbracket \cdot \rrbracket : M \rightarrow \hat{V} \rightarrow V \) truth: \( \hat{V} \rightarrow \hat{B} \) \( \forall m, \hat{v}. [\text{truth}(\hat{v})]_m^B \leftrightarrow (\llbracket \hat{v} \rrbracket_m^V \neq \#f) \) lift: \( (B \cup D \cup \hat{C}) \rightarrow \hat{V} \) \( \forall m, b. [\text{lift}(b)]_m^B = b \) \( \forall m, d. [\text{lift}(d)]_m^D = d \) \( \forall m, \hat{c}. [\text{lift}(\hat{c})]_m^\hat{C} = \llbracket \hat{c} \rrbracket_m^\hat{C} \) merge: \( \mathbb{G}_V \rightarrow \hat{V} \) \( \forall m, G. \text{one}_m(G) \rightarrow [\text{merge}(G)]_m^\hat{V} = [G]_m^\hat{V} \) cast: \( \hat{V} \rightarrow \mathbb{G}_C \) \( \forall m, \hat{v}. \llbracket \hat{v} \rrbracket_m^C \in C \rightarrow (\text{one}_m(\text{cast}(\hat{v}))) \land \llbracket \text{cast}(\hat{v}) \rrbracket_m^\mathbb{G}_C = \llbracket \hat{v} \rrbracket_m^\mathbb{G}_C \) \( \forall m, \hat{v}. \llbracket \hat{v} \rrbracket_m^C \notin C \rightarrow \text{none}_m(\text{cast}(\hat{v})) \) \( \hat{op} : O \rightarrow \hat{V}^* \rightarrow \hat{R} \) \( \forall m, \hat{v}_1, \ldots, \hat{v}_k. \llbracket \hat{op}(\hat{v}_1, \ldots, \hat{v}_k) \rrbracket_m^\hat{R} = \hat{op}(\llbracket \hat{v}_1 \rrbracket_m, \ldots, \llbracket \hat{v}_k \rrbracket_m) \land \text{legal}_m(\hat{op}(\hat{v}_1, \ldots, \hat{v}_k)) \) Symbolic boolean \( \hat{b} \in \hat{B} \) \( \llbracket \cdot \rrbracket : M \rightarrow \hat{B} \rightarrow B \) tt, ff: \( \hat{B} \) \( \forall m. \llbracket [\text{tt}] \rrbracket_m^\hat{B} = #t \) \( \forall m. \llbracket [\text{ff}] \rrbracket_m^\hat{B} = #f \) not: \( \hat{B} \rightarrow \hat{B} \) \( \forall m, \hat{b}. \llbracket \text{not}(\hat{b}) \rrbracket_m^\hat{B} = \neg \llbracket \hat{b} \rrbracket_m^\hat{B} \) and: \( \hat{B} \rightarrow \hat{B} \rightarrow \hat{B} \) \( \forall m, \hat{b}_1, \hat{b}_2. \llbracket \hat{b}_1 \land \hat{b}_2 \rrbracket_m^\hat{B} = (\llbracket \hat{b}_1 \rrbracket_m \land \llbracket \hat{b}_2 \rrbracket_m) \) or: \( \hat{B} \rightarrow \hat{B} \rightarrow \hat{B} \) \( \forall m, \hat{b}_1, \hat{b}_2. \llbracket \hat{b}_1 \lor \hat{b}_2 \rrbracket_m^\hat{B} = (\llbracket \hat{b}_1 \rrbracket_m \lor \llbracket \hat{b}_2 \rrbracket_m) \) imp: \( \hat{B} \rightarrow \hat{B} \rightarrow \hat{B} \) \( \forall m, \hat{b}_1, \hat{b}_2. \llbracket \text{imp}(\hat{b}_1, \hat{b}_2) \rrbracket_m^\hat{B} = (\llbracket \hat{b}_1 \rrbracket_m \rightarrow \llbracket \hat{b}_2 \rrbracket_m) \) Symbolic closure \( \hat{c} \in \hat{C} \) \( \equiv (\llbracket \text{cl} \times e, \hat{E} \rrbracket) \) Symbolic environment \( \hat{E} \in \hat{E} \) \( \equiv \text{Variable} \rightarrow \hat{V} \) Guarded choice \( g \in \hat{B} \times \alpha \) \( \equiv (\hat{b}, a) \) \( \text{where } a \in \alpha, \llbracket a \rrbracket_m^\alpha \in \beta \) \( \text{guard}(g) \equiv \hat{b} \) \( \text{choice}(g) \equiv a \) Guarded choices \( G \in \mathcal{G}_\alpha \equiv [g_1, \ldots, g_n] \) \( \llbracket \cdot \rrbracket^\mathcal{G}_\alpha : M \rightarrow \mathcal{G}_\alpha \rightarrow \beta \) \( \llbracket g : \mathcal{G}_\alpha \rrbracket_m^\beta := \text{if } \llbracket \text{guard}(g) \rrbracket_m^\alpha \text{ then } \llbracket \text{choice}(g) \rrbracket_m^\alpha \text{ else } (\mathcal{G}_\alpha)_m^\beta \) one: \( M \rightarrow \mathcal{G}_\alpha \rightarrow B \) \( \text{one}_m(G) \equiv \text{length}(\text{filter}(\lambda g. \llbracket \text{guard}(g) \rrbracket_m^\alpha, G)) = 1 \) none: \( M \rightarrow \mathcal{G}_\alpha \rightarrow B \) \( \text{none}_m(G) \equiv \text{length}(\text{filter}(\lambda g. \llbracket \text{guard}(g) \rrbracket_m^\alpha, G)) = 0 \) Symbolic state \( \sigma \in \Sigma \) \( \equiv (\hat{b}_1, \hat{b}_2) \) \( \text{assumes}(\sigma) \equiv \hat{b}_1 \) \( \text{asserts}(\sigma) \equiv \hat{b}_2 \) normal: \( M \rightarrow \Sigma \rightarrow B \) \( \text{normal}_m(\sigma) \equiv [\text{assumes}(\sigma)]_m^\Sigma \land [\text{asserts}(\sigma)]_m^\Sigma \) aborts: \( M \rightarrow \Sigma \rightarrow B \) \( \text{aborts}_m(\sigma) \equiv \neg [\text{assumes}(\sigma)]_m^\Sigma \land [\text{asserts}(\sigma)]_m^\Sigma \) errors: \( M \rightarrow \Sigma \rightarrow B \) \( \text{errors}_m(\sigma) \equiv [\text{assumes}(\sigma)]_m^\Sigma \land \neg [\text{asserts}(\sigma)]_m^\Sigma \) legal: \( M \rightarrow \Sigma \rightarrow B \) \( \text{legal}_m(\sigma) \equiv [\text{assumes}(\sigma)]_m^\Sigma \land [\text{asserts}(\sigma)]_m^\Sigma \) \( \equiv: M \rightarrow \Sigma \rightarrow \Sigma \rightarrow B \) \( \sigma_1 \equiv \sigma_2 \equiv [\text{assumes}(\sigma_1)]_m^\Sigma \land [\text{asserts}(\sigma_2)]_m^\Sigma \) Symbolic result \( \hat{r} \in \hat{R} \) \( \equiv \text{Out}(\sigma, \hat{v}) \land \text{Halt}(\sigma) \) state(\( \hat{r} \)) \( \equiv \sigma \) \( \text{value}(\text{Out}(\sigma, \hat{v})) \equiv \hat{v} \) \( \text{value}(\text{Halt}(\sigma)) \equiv \text{default}(\hat{V}) \) Result: \( \Sigma \rightarrow \hat{V} \rightarrow \hat{R} \) Result(\( \sigma, \hat{v} \)) \( \equiv \text{if } (\text{assumes}(\sigma) = \text{ff} \lor \text{asserts}(\sigma) = \text{ff}) \text{ then } \text{Halt}(\sigma) \text{ else } \text{Out}(\sigma, \hat{v}) \) one: \( \hat{R} : M \rightarrow \hat{R} \rightarrow R \) \( \text{one}_m(R) \equiv \text{if normal}_m(\sigma) \text{ then } \text{Ans}(\llbracket \hat{r} \rrbracket_m^\hat{R} \text{ else } \text{aborts}_m(\sigma) \text{ then } \text{Abt else Err} \) \( \text{Halt}(\sigma) \equiv \text{if aborts}_m(\sigma) \text{ then } \text{Abt else Err} \) one: \( \hat{R} : M \rightarrow \hat{R} \rightarrow B \) \( \text{legal}_m(\text{Out}(\sigma, \hat{v})) \equiv \text{legal}_m(\sigma) \) \( \text{legal}_m(\text{Halt}(\sigma)) \equiv \text{legal}_m(\sigma) \land \neg \text{normal}_m(\sigma) \) Fig. 4. Symbolic factory interface, parameterized by the set of all models \( M \), symbolic values \( \hat{V} \), and symbolic booleans \( \hat{B} \), as well as the parameters \( O, D \), and \( \hat{op} \) from the definition of \( \lambda_c \). We use \( \llbracket \cdot \rrbracket^\alpha \) to denote the interpreter for values of type \( \alpha \), omitting the superscript \( \alpha \) when it is clear from the context. The hat accent denotes the symbolic counterpart of a concrete entity; e.g., \( \hat{v} \) is a symbolic value, where \( v \) is a concrete value.

by defining guess(\( e, E \)) and verify(\( e, E \)) in terms of the symbolic states computed by \( S_c \). The next section establishes the correctness of \( S_c \) and the queries we define on top of it.

### 4.1 The Symbolic Factory Interface

At a high level, a symbolic semantics lifts the rules of a concrete semantics to operate on sets of concrete values, compactly represented as symbolic values. Practical evaluators have different ways
of representing and manipulating symbolic values, and as we saw in Section 1, these differences can lead to fundamentally different behaviors. To cover all reasonable behaviors in our formalization, we define $S_c$ against the abstract factory interface shown in Figure 4.

The factory interface is based on three core types: models $M$, symbolic values $\hat{V}$, and symbolic booleans $\hat{B}$. All three types are parameters to the factory, in addition to the parameters $D, O$, and $op$ inherited from $\lambda_c$. Conceptually, a symbolic boolean is a logical constraint on symbolic values; a symbolic value is an expression over symbolic variables; a model maps symbolic variables to concrete constants; and each symbolic boolean and value represents a unique concrete value under a given model. The factory interface captures these relationships abstractly through the interpretation functions $[\cdot]^\hat{B}$ and $[\cdot]^\hat{V}$, which use models to give concrete meaning to symbolic booleans and values, respectively. We specify the rest of the interface in terms of these core types and functions, dropping the superscript from the interpreter notation when it is clear from the context.

The factory types and functions provide a basic mechanism for lifting the semantics of $\lambda_c$. They serve as the symbolic counterparts of the concrete types, functions, and predicates that are used to define the concrete evaluation rules for $\lambda_c$ (Figure 3). We first explain how the factory components help lift these rules, and then illustrate a toy factory for the language $\lambda_Z$ from Example 1.

**Lifting constants.** The function $\text{lift} : (B \cup D \cup \hat{C}) \rightarrow \hat{V}$ provides a way to lift the meaning of constant expressions (LITERAL and Closure in Figure 3). Given an input $i$, $\text{lift}(i)$ returns a symbolic value that has the same interpretation as $i$ under all models. The input $i$ is either a concrete boolean, a concrete constant of type $D$, or a symbolic closure. A symbolic closure $\langle cl \ x \ e, \hat{E} \rangle \in \hat{C}$ is a straightforward generalization of a concrete closure: it combines a procedure $\langle \lambda x. e \rangle$ with a symbolic environment $\hat{E} \in \hat{E}$, which maps variable names to symbolic values. Symbolic closures and environments evaluate to their concrete counterparts via the interpretation functions $[\cdot]^\hat{C}$ and $[\cdot]^\hat{E}$. Conversely, concrete closures and environments can be made symbolic by recursively lifting their contents via $\text{lift}(i)$.

**Lifting conditionals.** The functions $\text{truth} : \hat{V} \rightarrow \hat{B}$ and $\text{merge} : \mathbb{G}_\hat{V} \rightarrow \hat{V}$ help lift the semantics of conditional expressions (IfTrue and IfFalse). The former lifts the conditional test for $\lambda_c$, and the latter lifts the output of conditional evaluation. In particular, $\text{truth}(\hat{v})$ encodes a logical predicate on $\hat{v}$ that is true unless $\hat{v}$ evaluates to $\#f$, while $\text{merge}(G)$ encodes the selection of a value from a list $G \in \mathbb{G}_\hat{V}$ of guarded choices. A guarded choice $g \in \mathbb{G}_\hat{V}$ pairs a symbolic boolean with a symbolic value (or, more generally, any value with an interpreter). When a list $G$ of such choices has one true guard under a given model, $\text{merge}(G)$ evaluates to the choice with the true guard. For example, $\text{merge}([\langle \hat{b} \ , \hat{v}_1 \rangle \ , \langle \text{not}(\hat{b}) \ , \hat{v}_2 \rangle])$ evaluates to $\langle \hat{v}_1 \rangle_m$ if $\langle \hat{b} \rangle_m$ is true, and to $\langle \hat{v}_2 \rangle_m$ otherwise. If $G$ has no or many true guards under a given model, the behavior of $\text{merge}(G)$ is unspecified for that model, and irrelevant in the context of the symbolic semantics $S_c$.

**Lifting procedure calls.** The function $\text{cast} : \hat{V} \rightarrow \mathbb{G}_\hat{C}$ lifts the dynamic cast from values to closures that is implicit in the semantics of procedure calls. The semantics uses two rules (CALL and CALLBAD) to handle the results of successful and failed casts on concrete values. We use $\text{cast}(\hat{v})$ to make this operation explicit on symbolic values. The result of a symbolic cast is a list of guarded symbolic closures, $\text{cast}(\hat{v}) \in \mathbb{G}_\hat{C}$. This list selects at most one closure under every model to match the behavior of $\hat{v}$: if $\langle \hat{v} \rangle_m$ is a closure, then $\text{cast}(\hat{v})$ evaluates to $\langle \hat{v} \rangle_m$, and if not, all guards in $\text{cast}(\hat{v})$ are false under $m$, indicating that the cast has failed. As an example, $\text{cast}(\text{merge}([\langle \hat{b} \ , \text{lift}(\hat{c}) \rangle \ , \langle \text{not}(\hat{b}) \ , \text{lift}(\#f) \rangle]))$ produces a list of guarded closures that is equivalent to $[\langle \hat{b} \ , \hat{c} \rangle]$ under every model. This list evaluates to $\langle \hat{c} \rangle_m$ when $\langle \hat{b} \rangle_m$ is true, just like the input to the cast, and has no true guards otherwise.
Symbolic booleans $\hat{b}_Z$, integers $\hat{d}_Z$, values $\hat{v}_Z$, and models $M_Z$ for $\lambda_Z$:

$$\hat{b} \in \hat{b}_Z \iff b \mid \{ \langle l \rangle \mid \{ b_1 & b_2 \} \} \mid \{ d_1 \sim \hat{d}_1 \} \quad b \in B \iff \#t \mid \#f \quad \sim \iff < \mid =$$

$$\hat{d} \in \hat{d}_Z \iff d \mid z \mid (-\hat{d}) \quad z \in Z \iff \text{symbolic integer variable} \quad d \in Z \iff \text{integer constant} \quad m \in M_Z \iff Z \rightarrow Z$$

Interpreters for symbolic booleans $\hat{b}_Z$, integers $\hat{d}_Z$, and values $\hat{v}_Z$, with respect to models $M_Z$:

$$\llbracket b \rrbracket_{M_Z} \equiv b \quad \llbracket l \rrbracket_{M_Z} \equiv \llbracket b \rrbracket_{M_Z} \quad \llbracket b_1 & b_2 \rrbracket_{M_Z} \equiv \llbracket b \rrbracket_{M_Z} \land \llbracket b_2 \rrbracket_{M_Z} \quad \llbracket d_1 \sim \hat{d}_1 \rrbracket_{M_Z} \equiv \llbracket d_1 \rrbracket_{M_Z} \sim \llbracket d_2 \rrbracket_{M_Z}$$

Factory operations on symbolic booleans $\hat{b}_Z$:

$$\llbracket \text{if} \rrbracket \equiv \#f \quad \text{not}(\hat{b}) \equiv \neg \hat{b} \quad \text{and}(\#f, \hat{b}) \equiv \#f \quad \text{and}(\hat{b}, \hat{b}) \equiv \hat{b} \quad \text{or}(\hat{b}_1, \hat{b}_2) \equiv \text{not}(\text{and}(\text{not}(\hat{b}_1), \text{not}(\hat{b}_2)))$$

Factory operations on symbolic values $\hat{v}_Z$:

$$\text{lift}(i) \equiv i \quad \text{truth}(\hat{b}) \equiv \hat{b} \quad \text{truth}(\hat{d}) \equiv \#t \quad \text{truth}(\hat{c}) \equiv \#t \quad \text{merge}(\{\}) \equiv \#f \quad \text{merge}(\{g\}) \equiv \text{choice}(g) \quad \text{merge}(g_1 \mid g_2 \mid G) \equiv \phi(\text{guard}(g_1), \text{choice}(g_1), \text{merge}(g_2 \mid G))$$

Next, we implement lift, truth, merge, cast, and $\hat{op}$. These are straightforward except for merge and $\hat{op}$. For example, $[\text{truth}(\phi(\hat{b}, \hat{b}_1, \hat{d}_2))]_{M_Z}$ is equivalent to $[(\hat{b})_{M_Z} \rightarrow [\hat{b}_1]_{M_Z} \land (\neg \hat{b})_{M_Z} \rightarrow [\#t]_{M_Z}]$, which says that truth($\phi(\hat{b}, \hat{b}_1, \hat{d}_2)$) is false only when $\hat{b}$ is true and $\hat{b}_1$ is false. Similarly, cast($\phi(\hat{b}, \hat{c}, \hat{d})$) produces $[(\hat{b}, \hat{c})]$, which evaluates to $\hat{c}$ when $\phi(\hat{b}, \hat{c}, \hat{d})$ does and has no true guards otherwise. To
see why merge\((G)\) works, recall that it must match the interpretation of \(G\) only under models that make exactly one guard in \(G\) true. In this case, merge\([\_]\) can return anything; merge\([\_\_]\) must return the sole chosen value; and the nested \(\phi\) value produced by merge\(\langle b, v_1, (\text{not}(b), v_2)\rangle\) is equivalent to \(g_1 :: g_2 :: G\). For example, merge\(\langle \langle b, v_1 \rangle, (\text{not}(b), v_2)\rangle\rangle\) returns \(\phi(b, v_1, v_2)\). We show the base cases for \(\phi(-, \ldots)\) and omit the rest for brevity. For example, \(\phi(-, \bar{d})\) errors because it is given the wrong the number of arguments, while \(\phi(-, \bar{d})\) terminates normally and produces the right value.

4.2 Evaluation Rules for \(S_c\)

Given a symbolic factory, we define the symbolic semantics \(S_c\) using the rules shown in Figure 6. The rules lift the concrete semantics of \(\lambda_c\) (Figure 3) to work on symbolic values. The judgment \(\langle e, \hat{E}, \sigma \rangle \Downarrow \hat{r}\) says that the expression \(e\) evaluates the symbolic result \(\hat{r}\) when evaluated in the symbolic environment \(\hat{E}\) and state \(\sigma\). Intuitively, this judgment encodes a set of concrete executions \(\langle e, [\hat{E}]_m \rangle \Downarrow [\hat{r}]_m\), one for each model \(m \in M\). The environment \(\hat{E} \in \hat{\Sigma}\) maps variables to symbolic values. The input state \(\sigma \in \Sigma\) consists of two formulas, \(\text{assumes}(\sigma)\) and \(\text{asserts}(\sigma)\), that jointly indicate if prior execution steps terminated normally, errored, or aborted under a given model (Figure 4). The result \(\hat{r} \in \hat{R}\) is either \(\text{Out}(\sigma', \hat{v})\) or \(\text{Halt}(\sigma')\). The former means that \(e\) may error, abort, or terminate normally to produce \([\hat{v}]_m\), with the outcome determined by the output state \(\sigma'\). The latter means that \(e\) cannot terminate normally under any model; i.e., \(\text{normal}_m(\sigma')\) is guaranteed to be false. We describe the rules of \(S_c\) below, and discuss the main features of our design in Section 4.3.

**Lifting constants and variables.** The rules for evaluating constants (\text{LITERAL}, \text{CLOSURE}) and variables (\text{VARIABLE}) are straightforward. Each lifts the concrete evaluation rule of the same name (Figure 3), by replacing all the concrete constructs with their symbolic counterparts. For example, the concrete VARIABLE rule looks up the variable \(x\) in the symbolic environment \(\hat{E}\), just as the concrete VARIABLE rule looks up \(x\) in the concrete environment \(E\). Similarly, the symbolic \text{LITERAL} and \text{CLOSURE} rules return the result of lifting a given literal and closure, respectively, via \(\text{lift}(i)\).

**Lifting error and abort expressions.** The rules \text{ERROR} and \text{ABORT} are more interesting. Each produces a result of the form \(\text{Halt}(\sigma')\), since evaluating (\text{ERROR}) or (\text{ABORT}) always leads to abnormal termination. The two rules differ in how they compute the output state \(\sigma'\): \text{ERROR} uses \text{assert}(\sigma, \text{ff}) and \text{ABORT} uses \text{assume}(\sigma, \text{ff})\). The resulting output states update one component of the input state—asserts or assumes, respectively—and leave the other component unchanged. These updates are symmetric, which is a key feature of \(S_c\) discussed in Section 4.3. For now, we note that \(\text{errors}_m(\text{assert}(\sigma, \text{ff}))\) and \(\text{aborts}_m(\text{assume}(\sigma, \text{ff}))\) if \(\sigma\) is normal under \(m\), and the two states are equivalent (\(\equiv_m\)) to \(\sigma\) otherwise. In other words, errors are treated as failed assertions; aborts are treated as failed assumptions; and both are no-ops when the input state is already abnormal.

**Example 4.** To illustrate the \text{ERROR} rule (and, by symmetry, the \text{ABORT} rule), suppose that we have \(\langle (\text{ERROR}), \hat{E}, \sigma \rangle \Downarrow \text{Halt}(\sigma')\), where \(\sigma'\) stands for \text{assert}(\sigma, \text{ff}). If the state \(\sigma\) is normal under a model \(m\), then we have:

\[
\text{errors}_m(\sigma') = [\text{assumes}(\sigma')]_m \land \neg[\text{asserts}(\sigma')]_m
\]

\[
= [\text{assumes}(\sigma)]_m \land \neg[\text{and}(\text{assumes}(\sigma), \text{imp}(\text{assumes}(\sigma), \text{ff}))]_m
\]

\[
= [\text{assumes}(\sigma)]_m \land \neg[\text{asserts}(\sigma)]_m \land ([\text{assumes}(\sigma)]_m \rightarrow [\text{ff}]_m)
\]

\[
= \#t \land \neg(#t \land (#t \rightarrow \#f))
\]

\[
= \#t
\]

In this case, \(\text{Halt}(\sigma')\) evaluates to \(\text{Err}\) under \(m\), as expected. Similar reasoning shows that if \(\sigma\) is abnormal under \(m\), then \(\sigma' \equiv_m \sigma\), so \(\text{Halt}(\sigma')\) evaluates to \(\text{Err}\) or \(\text{Abt}\), depending on \(\sigma'\), and the \text{ERROR} rule propagates the existing cause of abnormal termination.
A Formal Foundation for Symbolic Evaluation with Merging

\[
\begin{align*}
\text{LITERAL} & \quad \nu \in B \cup D \quad \Downarrow \begin{cases} 
\langle \nu, \hat{E}, \sigma \rangle & \Downarrow \text{Out}(\sigma, \text{lift}(\nu))
\end{cases} \\
\text{VARIABLE} & \quad x \in \text{dom}(\hat{E}) \quad \Downarrow \begin{cases} 
\langle x, \hat{E}, \sigma \rangle & \Downarrow \text{Out}(\sigma, \hat{E}[x/\hat{x}])
\end{cases} \\
\text{CALL} & \quad \langle x_1, \hat{E}, \sigma \rangle \Downarrow \text{Out}(\sigma, \hat{v}_1) \quad \Rightarrow \begin{cases} 
\langle x_2, \hat{E}, \sigma \rangle \Downarrow \text{Out}(\sigma, \hat{v}_2) \quad G_1 = \text{cast}(\hat{v}_1) \quad n = \text{length}(G_1) = \text{length}(G_2) \\
y = \text{some}(\text{guard}, G_1) \quad \sigma' = \text{assert}(\sigma, y) \quad y \neq \text{ff} \land \text{assumes}(\sigma') \neq \text{ff} \land \text{asserts}(\sigma') \neq \text{ff} \quad \forall i \in [0, n), \text{let } \langle \hat{b}_i, (\hat{c} l x, e, \hat{E}_i) \rangle := G_1[i], \langle \hat{b}_2, \hat{r} \rangle := G_2[i] \text{ in } \langle \hat{b}_1 = \hat{b}_2 \land (e, \hat{E}_1[x \mapsto \hat{v}_2], \text{assert}(\sigma', \hat{b}_1)) \Downarrow \hat{r} \rangle
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{CALLBAD} & \quad \langle x_1, \hat{E}, \sigma \rangle \Downarrow \text{Out}(\sigma, \hat{v}_1) \quad \Rightarrow \begin{cases} 
\langle x_2, \hat{E}, \sigma \rangle \Downarrow \text{Out}(\sigma, \hat{v}_2) \quad G_1 = \text{cast}(\hat{v}_1) \\
y = \text{some}(\text{guard}, G_1) \quad \sigma' = \text{assert}(\sigma, y) \quad y = \text{ff} \lor \text{assumes}(\sigma') = \text{ff} \lor \text{asserts}(\sigma') = \text{ff} \quad \langle \langle x_1, x_2, \hat{E}, \sigma \rangle \Downarrow \text{merge}(\sigma', G_2) \rangle
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{LET} & \quad \langle e_1, \hat{E}, \sigma \rangle \Downarrow \text{Out}(\sigma', \hat{v}_1) \quad \Rightarrow \begin{cases} 
\langle e_2, \hat{E}[x \mapsto \hat{v}_1], \sigma' \rangle \Downarrow \hat{r} \end{cases} \\
\text{LETHALT} & \quad \langle e_1, \hat{E}, \sigma \rangle \Downarrow \text{Halt}(\sigma') \quad \Rightarrow \begin{cases} 
\langle e_2, \hat{E}, \sigma \rangle \Downarrow \hat{r} \end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{ERROR} & \quad \langle \text{error}, \hat{E}, \sigma \rangle \Downarrow \text{Halt}(\text{assert}(\sigma, \text{ff})) \\
\text{ABORT} & \quad \langle \text{abort}, \hat{E}, \sigma \rangle \Downarrow \text{Halt}(\text{assert}(\sigma, \text{ff}))
\end{align*}
\]

\[
\begin{align*}
\text{assume} : \Sigma & \Rightarrow \hat{B} \Rightarrow \Sigma \\
\text{assert} : \Sigma & \Rightarrow \hat{B} \Rightarrow \Sigma \\
\text{merge}_R : \Sigma & \Rightarrow \hat{G}_R \Rightarrow \hat{R} \\
\text{merge}_G : \Sigma & \Rightarrow \hat{G}_R \Rightarrow \Sigma \\
\text{strengthen} : \Sigma & \Rightarrow \hat{R} \Rightarrow \hat{R} \\
\text{compose} : \Sigma & \Rightarrow \Sigma \Rightarrow \Sigma
\end{align*}
\]

\[
\begin{align*}
\text{some} : \alpha \Rightarrow \hat{B} & \Rightarrow \alpha^* \Rightarrow \hat{B} \quad \text{all} : \alpha \Rightarrow \hat{B} & \Rightarrow \alpha^* \Rightarrow \hat{B}
\end{align*}
\]

\[
\begin{align*}
\text{foldr} & \quad \text{foldr}(\lambda a. \hat{b} \cdot \text{or}(f(a), \hat{b}), \text{ff}, A) \\
\text{some} & \quad \text{some}(\lambda a. \hat{b} \cdot \text{or}(f(a), \hat{b}), \text{ff}, A)
\end{align*}
\]

Fig. 6. Symbolic semantics $S_c$ for $\lambda_c$, parameterized by the symbolic factory types and functions (Figure 4), as well as the parameters $O$, $D$, and $\text{op}$ from the syntax and concrete semantics of $\lambda_c$ (Figure 2, 3).

**Lifting procedure calls.** The rules CALL and CALLBAD lift the semantics of procedure calls $(x_1 \ z_2)$ as follows. Given $(x_1, \hat{E}, \sigma) \Downarrow \text{Out}(\sigma, \hat{v}_1)$ and $(x_2, \hat{E}, \sigma) \Downarrow \text{Out}(\sigma, \hat{v}_2)$, both rules compute cast$(\hat{v}_1)$ to extract all possible symbolic closures $G_1$ from $\hat{v}_1$. Then, they use $G_1$ to construct the formula $y = \text{some}(\text{guard}, G_1)$, which is true when some guard in $G_1$ is true, and the state $\sigma' = \text{assert}(\sigma, y)$, which
asserts that \( \gamma \) holds. If the factory reduces either \( \gamma \) or a component of \( \sigma' \) to \( \text{ff} \), we know that \( \hat{v}_1 \) is not a closure or \( \sigma' \) is not normal under any model. In this case, \textsc{CallBad} triggers and produces \text{Halt}(\sigma')

Otherwise, \textsc{Call} establishes that, for every choice \( \langle \hat{b}_1, \hat{c}_1 \rangle \in G_1 \), applying the closure \( \hat{c}_1 \) to the value \( \hat{v}_2 \) in the state assumes(\( \sigma', \hat{b}_1 \)) produces the result \( \hat{r}_i \). The guarded results \( \langle \hat{b}_1, \hat{r}_i \rangle \) form the list \( G_2 \) of all possible outcomes of applying \( \hat{v}_i \) to \( \hat{v}_2 \). \textsc{Call} merges these outcomes into the result merge_{\text{merge}}(\( \sigma', G_2 \)), which evaluates to \( \lceil G_2 \rceil_m \) if \( \sigma' \) is normal under \( m \), and to \( \lceil \text{Halt}(\sigma') \rceil_m \) otherwise. In a nutshell, \textsc{Call} applies the closures from \text{cast}(\( \hat{v}_1 \)) separately to \( \hat{v}_2 \) and merges the results, and \textsc{CallBad} short-circuits this process when the factory is able to determine that cast(\( \hat{v}_1 \)) can never succeed.

**Example 5.** Consider evaluating the procedure call (\( x_1, x_2 \)) in the environment \( \hat{E} = \{ x_1 \mapsto \hat{v}_1, x_2 \mapsto \hat{v}_2 \} \) and state \( \sigma = \langle \text{tt}, \text{tt} \rangle \), using the toy symbolic factory from Example 3. In this setting, we have \( \langle x_1, \hat{E}, \sigma \rangle \not\vdash \text{Out}(\sigma, \hat{v}_1) \) and \( \langle x_2, \hat{E}, \sigma \rangle \not\vdash \text{Out}(\sigma, \hat{v}_2) \), and we illustrate the two call rules as follows.

First, suppose that \( \hat{v}_1 = 42 \). Using the definitions from Figure 5, we see that \( G_1 = \text{cast}(\hat{v}_1) = [] \), \( \gamma = \text{some}(\text{guard}, G_1) = \text{ff} \), and \( \sigma' = \text{assert} (\sigma, \gamma) \). This trigger \text{CallBAD} to return \text{Halt}(\sigma'), which evaluates to \text{Err} under all models.

Second, suppose that \( \hat{v}_1 = \phi(b, \hat{c}_1, \hat{c}_2) \), where \( \hat{b} = (z < 0) \), \( \hat{c}_1 = (\hat{c} E x, 0, \hat{E} x) \), and \( \hat{c}_2 = (\hat{c} E x, 1, \hat{E} x) \).

We now have \( G_1 = \text{cast}(\hat{v}_1) = [\{ (\hat{b}, \hat{c}_1), (\hat{b}, \hat{c}_2) \}] \), \( \gamma = ((\text{ff} \hat{b}) \& \& (\text{ff} \hat{b})) \), and \( \sigma' = \text{assert} (\sigma, \gamma) \). This triggers the \text{Call} rule to establish that \( G_2 = [\{ (\hat{b}, \text{Out}(\sigma_1, 0)), (\hat{b}, \text{Out}(\sigma_2, 1)) \}] \), where \( \sigma_1 = \text{assume}(\sigma', \hat{b}) = (\text{ff} \hat{b}, \gamma) \) and \( \sigma_2 = \text{assume}(\sigma', \hat{b}) = (\text{ff} \hat{b}, \gamma) \). Finally, we have merge_{\text{merge}}(\( \sigma', G_2 \)) = \text{Out}(\text{merge}_{\text{merge}}(\( \sigma', G_2 \))) = \text{Out}(\{ (\hat{b}_1, \hat{b}_2), (\hat{b}_1, \hat{b}_2) \}) = \gamma \& \& (\text{ff} \hat{b} \& \& ! \gamma) \& \& !(\text{ff} \hat{b} \& \& ! \gamma) \). Applying basic logical simplifications to \( \gamma, \hat{b}_1, \) and \( \hat{b}_2 \), we see that they are all equivalent to \text{tt}. So, the result of the call is \text{Out}(\{ (\text{tt}, \text{tt}), (\phi(b, 0, 1)) \}), which matches the interpretation of \( G_2 \) under all models.

**Lifting let expressions and conditionals.** The rules \textsc{Let}, \textsc{Lethalt}, \textsc{Ifftrue}, \textsc{Iffalse}, and \textsc{Ifsym} are analogous to \textsc{call} and \textsc{CallBad}. In particular, \textsc{Let} and \textsc{Ifsym} provide a general mechanism for lifting let expressions and conditionals, and the remaining rules short-circuit this mechanism in important special cases. \textsc{Lethalt} ensures that the expression \( \langle \text{let} (x e_1) e_2 \rangle \) halts for the same reason as \( e_1 \) when \( e_1 \) is guaranteed to halt under all models. \textsc{Ifftrue} and \textsc{Iffalse} avoid executing infeasible branches of a conditional when the conditional test is a logical constant and therefore has the same value under all models. All three of these special-case rules mirror the corresponding concrete evaluation rules in Figure 3. The two general rules, \textsc{Let} and \textsc{Ifsym}, behave similarly to \textsc{Call}.

**Lifting operator calls.** The rule \textsc{Callop} lifts the semantics of operator calls using the factory function \( \hat{\circ} \) and the auxiliary function strength. The function strength(\( \sigma, \hat{r} \)) updates the result \( \hat{r} \) of \( \hat{\circ} \) so that it evaluates to \( \lceil \hat{r} \rceil_m \) when \( \sigma \) is normal under the model \( m \) and to \( \lceil \text{Halt}(\sigma) \rceil_m \) otherwise. In essence, \( \hat{\circ} \) calculates its result assuming an unconstrained start state, i.e., \( \langle \text{tt}, \text{tt} \rangle \), and \textsc{Callop} strengthens this result to reflect the constraints imposed by the input state \( \sigma \).

**Example 6.** Suppose that we want to evaluate \( \langle - x \rangle \) in the environment \( \hat{E} = \{ x \mapsto 0 \} \) and state \( \sigma \), using the toy factory from Example 3. In this case, we have \( \hat{r} = \hat{\circ}(\langle - 0 \rangle, 0) = \text{Out}(\langle \text{tt}, \text{tt} \rangle, 0) \). Assuming that neither component of \( \sigma \) is constant, we have strength(\( \sigma, \hat{r} \)) = \text{Result}(\text{compose}(\sigma, \langle \text{tt}, \text{tt} \rangle), 0) = \text{Result}(\langle \text{and}(\text{assumes}(\sigma)), \text{imp}(\text{asserts}(\sigma), \text{tt}) \rangle, \text{and}(\text{asserts}(\sigma), \text{imp}(\text{assumes}(\sigma), \text{tt}))) \), simplified to \text{Out}(\text{assumes}(\sigma), \text{asserts}(\sigma), 0) = \text{Out}(\sigma, 0). This result matches \( \lceil \hat{r} \rceil_m \) when \( \sigma \) is normal under \( m \), and it evaluates to \text{Err} or \text{Abt} otherwise, depending on \( \sigma \).

### 4.3 Properties of \( S_c \)

The evaluation rules for \( S_c \) are designed to preserve three key properties: legality, reducibility, and determinism. This last property is general and shared by other approaches. The first two are
inherent in symbolic execution but missing from prior merging semantics [Biere et al. 1999]. All three are important guarantees for tools built on top of reusable symbolic evaluators.

**Legality.** Legality is a property of symbolic states that gives client tools a simple interpretation of what these states mean. In particular, the two bits of state distinguish normal termination, where both bits are true, from errors and aborts, where one bit is true and the other is false. A state with two false bits has no natural meaning, so a state is legal under a model \( m \) if at least one of its components, \( \text{assumes}(\sigma) \) or \( \text{asserts}(\sigma) \), is true under \( m \) (Figure 4). Our semantics supports this intuitive interpretation by ensuring that every legal state leads to a legal result.

**Theorem 2.** Evaluating an expression from a legal symbolic state leads to a legal symbolic result:

\[
\forall e, \hat{E}, \sigma, \hat{r}, m. (\text{legal}_m(\sigma) \land \langle e, \hat{E}, \sigma \rangle \Downarrow \hat{r}) \rightarrow \text{legal}_m(\hat{r}).
\]

Preserving legality amounts to guaranteeing that each state \( \sigma \) is free of errors when \( \text{assumes}(\sigma) \rightarrow \text{asserts}(\sigma) \) holds under a given model. Symbolic execution maintains this relation by calling the solver to check the symbolic state after each update, and proceeding only if the state remains feasible and free of errors. Both symbolic execution and the classic merging semantics update the symbolic state in the same way: assert \( (\sigma, \hat{b}) \) from Figure 6 updates the assertions, and assume \( (\sigma, \hat{b}) \) := \langle and(\text{assumes}(\sigma), \hat{b}), \text{asserts}(\sigma) \rangle \) updates the assumptions. But the classic semantics does not check the updated states for errors and feasibility, and without these checks, the updates can produce illegal results. Our semantics uses a different updating function, assume \( (\sigma, \hat{b}) \), which preserves legality by construction, as illustrated in Example 7.

**Example 7.** To illustrate the difference between \( S_c \) and the classic merging semantics, consider evaluating the following program in the environment \( \hat{E} = \{ x \mapsto \hat{v}_1, y \mapsto \hat{v}_2 \} \) and state \( \sigma = \langle \text{tt}, \text{tt} \rangle \):

```
1  (let (_ (if x #t (error))) ; assert truth(x)
2  (let (_ (if y #t (abort))) ; assert truth(y)
```

Given these inputs, \( S_c \) produces a result that is equivalent to \( \hat{r} = \text{Out}(\text{assume}(\text{assert}(\sigma, \hat{b}_1), \hat{b}_2), \cdot) \), and the classic semantics produces one equivalent to \( \hat{r}' = \text{Out}(\text{assume}'(\text{assert}(\sigma, \hat{b}_1), \hat{b}_2), \cdot) \), where \( \hat{b}_1 = \text{truth}(\hat{v}_1) \) and \( \hat{b}_2 = \text{truth}(\hat{v}_2) \). After simplifying the resulting states, we get state(\( \hat{r} \)) = \langle \text{imp}(\hat{b}_1, \hat{b}_2), \hat{b}_1 \rangle \) and state(\( \hat{r}' \)) = \langle \hat{b}_2, \hat{b}_1 \rangle. The former is legal under all models, and it is free of errors when \( [\text{assumes}(\text{state}(\hat{r})) \rightarrow \text{asserts}(\text{state}(\hat{r}))]_m = [\hat{b}_1]_m \) holds. The latter is illegal when \( \hat{b}_1 \) and \( \hat{b}_2 \) are both false, and tools must account for this when formulating verification queries.

**Reducibility.** Reducibility is a property of symbolic evaluation that lets client tools treat the symbolic evaluator as a generalized concrete interpreter. Informally, a reducible symbolic evaluator behaves like the underlying concrete interpreter when applied to a (lifted) concrete environment. Reducibility is crucial for testing of client code (see, e.g., [Nelson et al. 2019]), and for ensuring that symbolic evaluation abandons infeasible paths as soon as possible. It is baked into symbolic execution, which reduces to evaluating the same program path as concrete execution in a concrete environment. In contrast, the classic merging semantics mirrors the concrete semantics only on loop-free programs—loops can cause it to diverge even when the corresponding concrete execution terminates. Our symbolic semantics reduces to the concrete semantics on all programs (Theorem 3), when coupled with a symbolic factory that takes (lifted) concrete inputs to (lifted) concrete outputs (Definition 5). This optimization is common to practical factories, as well as the toy one from Example 3. Unlike the classic merging semantics, \( S_c \) takes advantage of this optimization to abandon halted paths and avoid infeasible infinite loops whenever possible. The key is to introduce the notion of halted results, \( \text{Halt}(\sigma) \), and the rules for propagating them, as shown in Example 8.
**Definition 5 (Reducing Factory).** A symbolic factory is a reducing factory if and only if it satisfies the factory specification (Figure 4) and takes lifted concrete inputs to lifted concrete outputs:

\[
\begin{aligned}
&\text{truth}(\text{lift}_V(\#f)) = \text{ff} & \forall b. \text{not}(b) = \text{lift}_B(\neg b) \\
&\forall v. v \neq \#f \rightarrow \text{truth}(\text{lift}_V(v)) = \text{tt} & \forall b_1, b_2. \text{and}(b_1, b_2) = \text{lift}_B(b_1 \wedge b_2) \\
&\forall v. \text{merge}(\langle tt, \text{lift}_V(v) \rangle) = \text{lift}_V(v) & \forall b_1, b_2. \text{or}(b_1, b_2) = \text{lift}_B(b_1 \vee b_2) \\
&\forall x, e. \text{cast}(\text{lift}_V((\text{cl} x, e, E))) = \langle \langle \text{cl} x, e, \text{lift}_E(E) \rangle \rangle & \forall b_1, b_2. \text{imp}(b_1, b_2) = \text{lift}_B(b_1 \rightarrow b_2) \\
&\forall v. v \notin C \rightarrow \text{cast}((\text{lift}_V(v)) = [] & \forall v_1, \ldots, v_n. \text{op}(\alpha, \text{lift}_V(v_1), \ldots, \text{lift}_V(v_n)) = \text{lift}_R(\text{op}(\alpha, v_1, \ldots, v_n))
\end{aligned}
\]

Here, the lifting functions generalize \(\text{lift}\) as follows:

\[
\begin{aligned}
\text{lift}_V(b) &:= \text{lift}(b) \\
\text{lift}_B(\#t) &:= \text{tt} \\
\text{lift}_R(\text{Err}) &:= \text{Halt}(\langle tt, ff \rangle) \\
\text{lift}_R(\text{Abt}) &:= \text{Halt}(\langle tt, ff \rangle)
\end{aligned}
\]

\[
\begin{aligned}
\text{lift}_V(d) &:= \text{lift}(d) \\
\text{lift}_B(\#f) &:= \text{ff} \\
\text{lift}_R(\text{Ans}(v)) &:= \text{Out}(\langle tt, tt \rangle, \text{lift}_V(v))
\end{aligned}
\]

\[
\begin{aligned}
\text{lift}_V((\text{cl} x, e, E)) &:= \text{lift}(\langle \text{cl} x, e, \text{lift}_E(E) \rangle) \\
\text{lift}_E(E) &:= \langle x \mapsto \text{lift}_V(E[x]) \mid x \in \text{dom}(E) \rangle \\
\text{lift}_R(\langle E[x], \langle \text{cl} x, e, E \rangle \rangle) &:= \text{Out}(\langle tt, tt \rangle, \text{lift}_V(v))
\end{aligned}
\]

**Theorem 3.** If \(S_c\) is parameterized with a reducing symbolic factory, then it reduces to the concrete semantics of \(\lambda_c\) in all lifted concrete environments: \(\forall e, E, r. \langle e, \text{lift}_E(E), \langle tt, tt \rangle \rangle \not\parallel \text{lift}_R(r) \leftrightarrow \langle e, E \rangle \downarrow r.\)

**Example 8.** Consider evaluating the following program in the environment \(\tilde{E} = \{x_1 \mapsto \text{lift}_V(\#f), x_2 \mapsto \text{lift}_V(\#f)\}\) and state \(\sigma = \langle tt, tt \rangle\), using a reducing factory such as the toy factory from Example 3:

\[
\begin{aligned}
1 & \langle \text{let} (x_2 \ (x_1 \ x_2)) \rangle \\
2 & \langle \text{let} (y \ (y \ x) \ (y \ y)) \rangle \\
3 & \langle y \ y \rangle
\end{aligned}
\]

From Definition 5, we see that \(\text{cast}(\text{lift}_V(\#f)) = []\) so some(\text{guard}, []) = \text{ff}, triggering \text{CALLBAD} to return \text{Halt}(\sigma') = \text{Halt}(\text{assert}(\sigma, \#f)) = \text{Halt}(\langle tt, ff \rangle). This result then triggers \text{LETHALT} to return \text{Halt}(\langle tt, ff \rangle) = \text{lift}_R(\text{Err}) as the output of the program, matching its concrete execution in the environment \(E = \{x_1 \mapsto \#f, x_2 \mapsto \#f\}\). Now consider evaluating the same program with the classic merging semantics, which has no notion of halted results or rules for handling them. Without this mechanism, \text{CALL} would return \text{Out}(\text{assert}(\sigma, \#f), -1) and trigger \text{LET}, leading to an infinite loop.

**Determinism.** In addition to preserving legality and reducibility, \(S_c\) is also deterministic (Theorem 4): it always produces the same result when applied to the same environment and state. This is important for the development and debugging of client tools, as well as for their usability. Assuming that the underlying solver is deterministic too, a client query is guaranteed to behave consistently across runs, by consuming the same amount of resources to produce the same output.

**Theorem 4.** Evaluating an expression from the same symbolic environment and state produces the same symbolic result: \(\forall e, \tilde{E}, \sigma, \tilde{r}_1, \tilde{r}_2. \langle e, \tilde{E}, \sigma \rangle \not\parallel \tilde{r}_1 \wedge \langle e, \tilde{E}, \sigma \rangle \not\parallel \tilde{r}_2 \rightarrow \tilde{r}_1 = \tilde{r}_2.\)

### 4.4 Angelic Execution and Verification with \(S_c\)

Given an implementation of \(S_c\) and a solver for formulas \(\hat{b} \in \hat{B}\) (Definition 6), we can implement the angelic execution and verification queries as shown in Definition 7. Both queries use \(S_c\) to evaluate the input expression \(e\) in the symbolic environment \(\tilde{E}\) that represents a set of concrete environments \(\mathcal{E} = \{E \mid \exists m. \tilde{E}_m = E\}\). Angelic execution then uses the solver to search for a model in which the resulting symbolic state is normal, and verification searches for a model in which the resulting state errors. The next section shows that this correctly implements Definitions 3 and 4, respectively.

**Definition 6 (Solver).** Given a symbolic boolean \(\hat{b} \in \hat{B}\), a solver \(\text{solve}(\hat{b})\) diverges or produces one of two results: either a model \(m \in M\) such that \([\hat{b}]_m\) is true, or \text{unsat} if no such model exists.
What does it mean for a symbolic semantics to be correct, i.e., sound and complete? In the case of work [Jourdan 2016; Jourdan et al. 2015], this phrasing avoids the need to prove that abstract This formulation works for all programs and all symbolic execution trees. But because it assumes

This section establishes the correctness of 5 CORRECTNESS OF $S_c$

5.1 Soundness and Completeness of $S_c$

What does it mean for a symbolic semantics to be correct, i.e., sound and complete? In the case of symbolic execution, soundness and completeness can be formulated by relating paths in the symbolic execution tree to concrete execution traces (see, e.g., [Fragoso Santos et al. 2020; Lucanu et al. 2017]): soundness means that every concrete trace of length $n$ is covered by some feasible symbolic path of length $n$, and completeness means that every feasible symbolic path corresponds to a concrete trace. This formulation works for all programs and all symbolic execution trees. But because it assumes path-based evaluation, it does not apply to merging semantics such as $S_c$. To reason about symbolic evaluation with merging, we take inspiration from prior work on formalizing static analyzers [Jourdan 2016; Jourdan et al. 2015] based on abstract interpretation [Cousot and Cousot 1977, 1979].

We adapt two ideas from this work [Cousot and Cousot 1977, 1979; Jourdan 2016; Jourdan et al. 2015] to our setting. First, we define what it means for a merging semantics to be correct by relating the final result of symbolic evaluation to the final results of concrete evaluation, instead of relating paths in the symbolic evaluation graph to concrete traces. Second, we phrase our notion of correctness so that all diverging runs of the symbolic evaluator are trivially correct. In prior work [Jourdan 2016; Jourdan et al. 2015], this phrasing avoids the need to prove that abstract interpretation terminates, which is always possible but may be tedious. In our setting, this phrasing is necessary because symbolic evaluation may not terminate in the presence of loops.

Analogously to prior work, we formulate the soundness and completeness theorem for $S_c$ (Theorem 5) by viewing symbolic environments and results as sets of concrete environments and results. Recall from Section 4 that the factory interpretation functions $\lceil \cdot \rceil^d$ use models $m \in M$ to relate symbolic objects $\hat{a} \in \hat{a}$ to their concrete counterparts $a = [\hat{a}]_m^d \in \alpha$. So, every symbolic object $\hat{a}$ represents the set of all concrete objects $a \in \alpha$ to which $\hat{a}$ can evaluate via some model. We formalize this relation by writing $a \in \hat{a}$ to denote that $a = [\hat{a}]_m^d$ for some model $m \in M$ (Definition 8). Given the relation $\in$, our correctness theorem for $S_c$ states the following. Let $\hat{E}$ and $\hat{r}$ be a symbolic environment and result such that $\langle e, \hat{E}, (tt, tt) \rangle \triangleright \hat{r}$. Then, $\hat{r}$ both overapproximates and underapproximates the set of concrete results that can be reached from $\hat{E}$: if $r$ is the result of a concrete run from an environment $E \in \hat{E}$, then $r \in \hat{r}$; and every $r \in \hat{r}$ can be produced by a concrete run from some environment $E \in \hat{E}$. In other words, $\hat{r}$ precisely captures the set of all concrete results that are reachable from $\hat{E}$. 
We write and validate two different implementations of \( \lambda \) with a sound and complete result (Theorem 5).

The reference evaluator, which we call \( \tilde{\lambda} \), is implemented as a generic symbolic interpreter in Lean, and the other is an optimized evaluator written in Racket. We use Lean to prove that the reference evaluator correctly implements the optimized evaluator against the reference one. This section presents our implementations, building on the correctness of \( \tilde{\lambda} \) (Theorem 7).

### 5.2 Correctness of Queries Based on \( S_c \)

Building on the correctness of \( S_c \), we use Theorems 7 and 8 to show that our implementations of angelic execution and verification (Definition 7) satisfy Definitions 3 and 4, respectively. Theorem 7 establishes that guess(\( e, \hat{E} \)) produces a correct output whenever it terminates. If the output of guess(\( e, \hat{E} \)) is an environment \( E \), then \( e \) executes normally in \( E \in \hat{E} \). But if the output is unsat, then there is no environment \( E \in \hat{E} \) in which \( e \) executes normally. Theorem 8 is symmetric.

**Theorem 7.** If guess(\( e, \hat{E} \)) terminates, then its output is correct according to Definition 3.

\[
\begin{align*}
\text{SAT: } & \forall e, \hat{E}, E. \\text{guess}(e, \hat{E}) = E \rightarrow (E \in \hat{E} \land \text{normal}(e, E)) \\
\text{UNSAT: } & \forall e, \hat{E}, \text{guess}(e, \hat{E}) = \text{unsat} \rightarrow \forall E. (E \in \hat{E} \rightarrow \neg \text{normal}(e, E))
\end{align*}
\]

**Theorem 8.** If verify(\( e, \hat{E} \)) terminates, then its output is correct according to Definition 4.

\[
\begin{align*}
\text{SAT: } & \forall e, \hat{E}, E. \\text{verify}(e, \hat{E}) = E \rightarrow (E \in \hat{E} \land \text{errors}(e, E)) \\
\text{UNSAT: } & \forall e, \hat{E}, \text{verify}(e, \hat{E}) = \text{unsat} \rightarrow \forall E. (E \in \hat{E} \rightarrow \neg \text{errors}(e, E))
\end{align*}
\]

### 6 IMPLEMENTING \( S_c \): A CASE STUDY OF TWO EVALUATORS

To demonstrate the suitability of our framework for developing and validating reusable evaluators, we write and validate two different implementations of \( S_c \). One is a reference evaluator written in Lean, and the other is an optimized evaluator written in Racket. We use Lean to prove that the reference evaluator correctly implements \( S_c \), and we use solver-aided differential testing to validate the optimized evaluator against the reference one. This section presents our implementations, correctness theorems, testing setup, and test results.

#### 6.1 Leanette: A Verified Implementation of \( S_c \) in Lean

The reference evaluator, which we call LEANETTE, is implemented as a generic symbolic interpreter for \( \lambda \). Like \( S_c \), it is parameterized by a symbolic factory and relies on the factory interface to construct and deconstruct symbolic values. Because Lean requires all functions to terminate, LEANETTE ensures termination in the standard way, by using a fuel parameter \( n \in \mathbb{N} \) to bound the depth of the
recursive call stack. With the addition of fuel, \textsc{Leanette}(n, e, \hat{E}, \sigma) produces either a symbolic result \hat{r} such that \langle e, \hat{E}, \sigma \rangle \uplus \hat{r}, or \texttt{none}, to indicate that the evaluation is stuck. This implementation is both sound and complete with respect to our symbolic semantics (Theorem 9): \( S_c \) admits every result produced by \textsc{Leanette}, and, given enough fuel, \textsc{Leanette} can produce every result admitted by \( S_c \).

**Theorem 9 (Leanette soundness and completeness).** The \textsc{Leanette} symbolic evaluator is sound and complete with respect to the semantics \( S_c \).

**Soundness:** \forall n, e, \hat{E}, \sigma, \hat{r}. \textsc{Leanette}(n, e, \hat{E}, \sigma) = \hat{r} \rightarrow \langle e, \hat{E}, \sigma \rangle \uplus \hat{r}

**Completeness:** \exists e, \hat{E}, \sigma. \langle e, \hat{E}, \sigma \rangle \uplus \hat{r} \rightarrow \exists n. \textsc{Leanette}(n, e, \hat{E}, \sigma) = \hat{r}

We instantiate \textsc{Leanette} with a naïve symbolic factory \( F_L \), which supports integers, lists, and basic operations on these datatypes (+, *, =, <, cons, car, cdr, null?). The factory supports two types of symbolic variables, booleans and integers, and models \( m \in M \) map these variables to concrete values of the right type. Symbolic booleans \( \hat{B} \), integers \( \hat{D} \), and values \( \hat{V} \) are defined similarly to the toy factory in Figure 5, with one main exception: we use symbolic unions instead of \( \phi \) values to represent the output of conditional evaluation. In our implementation, a symbolic union is a list of guarded values, \( \{ (\hat{b}_1, \hat{v}_1), \ldots, (\hat{b}_k, \hat{v}_k) \} \in \hat{G}_V \), that are themselves not unions (i.e., \( \hat{v}_i \notin \hat{G}_V \)). This representation is a simplified version of the symbolic unions used in Rosette [Torlak and Bodik 2014]. Unlike Rosette’s factory, \( F_L \) offers no guarantees on the size of the unions or formulas generated during evaluation. In fact, both are worst-case exponential in the size of the (unrolled) input program. But \( F_L \) is easier to reason about, and we prove that it satisfies the symbolic factory interface (Theorem 10). We also prove that \( F_L \) is a reducing factory, so our instantiation of \textsc{Leanette} is a sound, complete, and reducible symbolic evaluator for \( \lambda_c \) with integers and lists.

**Theorem 10 (Leanette factory correctness and reducibility).** The \textsc{Leanette} factory \( F_L \) satisfies the symbolic factory interface (Figure 4) and the definition of a reducing factory (Definition 5).

### 6.2 Rosette 4: An Optimized Implementation of \( S_c \) in Racket

The optimized evaluator, \textsc{Rosette 4}, implements \( S_c \) for the entire Rosette language [Torlak and Bodik 2014], which is a superset of \( \lambda_c \). The Rosette language extends Racket [Felleisen et al. 2018; Platt and PLT 2010] with constructs for creating symbolic values, emitting assertions, and formulating solver-aided queries. Rosette’s existing reusable evaluator, which we call \textsc{Rosette 3}, is based on the classic merging semantics, and it has been used to develop over 30 solver-aided tools for a wide variety of applications. \textsc{Rosette 4} replaces the core evaluation rules of \textsc{Rosette 3} with those of \( S_c \).

\textsc{Rosette 4} also makes two changes to the \textsc{Rosette 3} interface, to reflect the switch to \( S_c \). First, it extends the Rosette language to include assumptions, \( \texttt{assume} \ e \), which act as syntactic sugar for the \( \texttt{abort} \) expression in \( \lambda_c \). This change makes assumptions a first-class construct in \textsc{Rosette 4}; they can appear anywhere in a program. In contrast, \textsc{Rosette 3} has limited support for assumptions via \#; \texttt{assume} \ e clauses in verification and synthesis queries. These clauses require client tools to emit all required assumptions upfront, which is not always feasible, leading to complex custom code for precondition tracking (see Section 7.1). Second, \textsc{Rosette 4} exposes a different interface for symbolic reflection [Torlak and Bodik 2014]. Symbolic reflection is a mechanism for allowing client tools to observe the symbolic state during evaluation, and to control the performance of the evaluator using high-level language constructs. In \textsc{Rosette 3}, this interface exposes concepts such as the assertion stack and the path condition, which are part of the classic merging semantics. In \textsc{Rosette 4}, this interface exposes concepts such as the symbolic state with assumptions and assertions, as defined by \( S_c \).

The changes to the evaluator and the interface account for the major differences between the two implementations. At the code level, this amounts to roughly 2,000 line insertions and deletions.
The rest of the code base is shared (roughly 5,000 lines), including the datatypes and procedures for operating on symbolic values. So both implementations share the same symbolic factory.

6.3 Validating Rosette 4 against Leanette with Solver-Aided Differential Testing

To gain confidence that Rosette 4 is correct, we perform differential testing [McKeeman 1998] against Leanette. Our testing setup consists of a generator for constructing input programs and environments, and an oracle for checking if the two evaluators produce equivalent results on the same input. We use this setup to check that Rosette 4 behaves like Leanette on a total of 10,000 test programs and environments. The rest of this section describes our test oracle, generator, and results.

Test oracle. Given a program $e$ and symbolic environment $\hat{E}$, the oracle compares the outputs produced by Rosette 4$(e, \hat{E}, \langle tt, tt \rangle)$ and Leanette$(n, e, \hat{E}, \langle tt, tt \rangle)$, within a timeout of 40 seconds and a fuel limit of $n = 100$. This comparison succeeds only when both outputs are symbolic results, and these results are equivalent under every model, i.e., Rosette 4$(e, \hat{E}, \langle tt, tt \rangle) = \hat{r}_R$, Leanette$(n, e, \hat{E}, \langle tt, tt \rangle) = \hat{r}_L$, and $\forall m. [\hat{r}_R]_m^K = [\hat{r}_L]_m^K$. To implement the equivalence check, the oracle imports $\hat{f}_L$ into Rosette 4, and uses Z3 [De Moura and Bjørner 2008] to solve the query $\langle\text{verify}\ (\text{equal}\ (\text{equal}\ [\hat{r}_R]_m^K\ [\hat{r}_L]_m^K))\rangle$. This query searches for a model $m$, if any, under which the two symbolic results evaluate to different concrete results according to $\text{equal}\$. The function $\text{equal}\$ considers its inputs to be equivalent if they have the same representation; e.g., two lists are $\text{equal}\$ if they have the same length and $\text{equal}\$ elements, and two closures are $\text{equal}\$ if they have $\text{equal}\$ lambda terms and environments. This equivalence relation behaves like the equality relation $=$ in Lean, so the oracle reflects the notion of equality used in our Lean formalization.

Test generator. Our test generator produces closed programs that couple $\lambda_c$ expressions with their symbolic environments. Each generated test is a valid Rosette program that consists of a sequence of $\langle\text{define-symbolic}\ x\ T\rangle$ expressions, followed by an expression $e$ from the $\lambda_c$ grammar. The definition sequence populates the symbolic environment by binding each free variable in $e$ to a fresh symbolic variable of type boolean or integer. More complex values can then be constructed by the body $e$ from these primitives. The tests are generated in two steps. First, the generator uses fair enumeration combinators [New et al. 2017] to create a bijection $\eta$ between the set of natural numbers and the set of all possible test programs. The bijection $\eta$ is set up so that larger numbers tend to correspond to larger programs. Next, the generator uses $\eta$ to convert random natural numbers into test programs of varying sizes. In particular, the generator takes two inputs: the number $N$ of tests to generate, and an indirect bound $k$ on the size of the generated tests. Given these inputs, it produces $N$ distinct tests by repeatedly calling $\eta(\text{random}(2^i))$, where $i$ increases linearly from 0 to $k$, and $\text{random}(2^i)$ generates a natural number from 0 to $2^i$ uniformly at random. This process can generate any test program given a suitable random seed, $N$, and $k$.

Example 9. To illustrate our differential testing setup, suppose that the generator produces the two tests shown in Figure 7. Both tests are conditional expressions that branch on the value of the boolean expression $0 * n = 0$, where $n$ is a symbolic integer. Test 7a returns $\#t$ if this expression is true and diverges otherwise. Test 7b errors if the expression is true and returns $\#t$ otherwise. Applying the oracle to these tests, we find that Rosette 4 and Leanette behave differently on 7a and equivalently on 7b.

When applied to 7a, Rosette 4 produces Out$(\langle tt, tt \rangle, tt)$, and Leanette runs out of fuel. In fact, Leanette will always run out of fuel on this program because its symbolic factory is unable to reduce $0 * n$ to 0. Rosette 4 is able to perform this reduction, so it terminates with the expected value. Both of these outcomes are correct according to formalization (Section 5).
state” columns show the number of non-trivial symbolic states produced by the evaluator, as well "time" columns display the maximum and average evaluation time for all programs in a bucket. The "Avg" columns are the maximum and average resulting symbolic state size

\[ \sigma = \langle v_1, \ldots, v_k \rangle \]

1 (define-symbolic n integer?); Symbolic integer
2 #t; then true
3 (if (zero? (* 0 n)); If 0 * n = 0
4 (assert #f); then error
5 (let loop () ; else loop infinitely
6 (loop))

(a) A program that terminates under Rosette 4 but does not terminate under Leanette.

(b) A program that produces Halt(·) under Rosette 4 but produces Out(·, ·) under Leanette.

Fig. 7. Two test programs that combine a \( \lambda_c \) expression and a symbolic environment. The programs are shown in the Rosette syntax for brevity. The test oracle determines that Rosette 4 and Leanette differ on (a) due to non-termination, and that they agree on (b) according to our equivalence relation.

Table 1. Summary of the testing results, which are bucketed by AST size. Under “Leanette state”, the “Sym” column refers to the number of non-trivial resulting symbolic states in Leanette evaluation. The “Max” and “Avg” columns are the maximum and average resulting symbolic state size among the non-trivial ones. Columns under “Rosette 4 state” have the same meanings. “Validated?” column is the oracle output, where “✓” means the evaluation results from both evaluators agree on all programs in a bucket.

<table>
<thead>
<tr>
<th>Bucket (by AST size)</th>
<th>Leanette state</th>
<th>Rosette 4 state</th>
<th>Leanette time</th>
<th>Rosette 4 time</th>
</tr>
</thead>
</table>
|                      | Sym | Max | Avg | Max | Avg | Avg | Max (s) | Avg (s) | Max (s) | Avg (s) | Validated?
| 1 - 12               | 1,006 | 6 | 220 | 635 | 41 | 88 | 20 | 4 | 2.4 | 2.2 | 0.3 | 0.1 | ✓
| 13 - 24              | 1,021 | 19 | 349 | 1,142 | 77 | 174 | 36 | 6 | 2.3 | 2.2 | 0.2 | 0.1 | ✓
| 25 - 38              | 1,042 | 32 | 430 | 2,855 | 102 | 242 | 34 | 6 | 2.3 | 2.2 | 0.8 | 0.1 | ✓
| 39 - 50              | 1,036 | 44 | 433 | 16,979 | 242 | 187 | 34 | 7 | 2.8 | 2.2 | 0.2 | 0.1 | ✓
| 51 - 65              | 1,040 | 58 | 450 | 7,532 | 186 | 227 | 30 | 8 | 2.4 | 2.2 | 0.2 | 0.1 | ✓
| 66 - 77              | 1,040 | 72 | 459 | 30,386 | 474 | 239 | 45 | 8 | 2.8 | 2.2 | 0.2 | 0.1 | ✓
| 78 - 88              | 1,053 | 83 | 454 | 19,835 | 426 | 231 | 36 | 8 | 2.5 | 2.2 | 0.2 | 0.1 | ✓
| 89 - 102             | 1,042 | 95 | 427 | 45,893 | 482 | 190 | 41 | 9 | 3.5 | 2.2 | 0.2 | 0.1 | ✓
| 103 - 119            | 1,009 | 111 | 425 | 61,439 | 686 | 218 | 45 | 9 | 4.1 | 2.2 | 1.0 | 0.1 | ✓
| 120 - 158            | 711 | 129 | 324 | 293,171 | 2,001 | 159 | 36 | 9 | 36.6 | 2.2 | 0.3 | 0.1 | ✓

When applied to 7b, Rosette 4 produces Halt(⟨tt, ff⟩), and Leanette produces Out(σ, tt), where \( σ = \langle tt, !(0 = (0 + n)) \rangle \). These symbolic results have different representations because, once again, Rosette 4 reduces \( 0 * n \) to 0 and Leanette does not. But they evaluate to the same concrete result under every model \( m \), i.e., \( [\text{Halt}(⟨tt, ff⟩)]^R_m = [\text{Out}(σ, tt)]^R_m \), so the oracle considers them equivalent.

**Test results.** We validate Rosette 4 against Leanette on a set \( T \) of 10,000 randomly generated test programs. For each program in \( T \), we collect the oracle output and, for each evaluator, the final symbolic state and the running time. Table 1 shows the test results, where \( T \) is partitioned into buckets by AST size. The largest program in \( T \) consists of 158 expressions, and the average program size is 63 expressions. Both Leanette and Rosette 4 terminate and produce a symbolic result on every program in \( T \), within the oracle timeout and fuel limit.

Using \( T \), our testing setup quickly discovers an intentional semantics discrepancy between Leanette and Rosette 4. In Leanette, the cons operator creates only lists. It takes as input a value \( v_0 \) and a list \( [v_1, \ldots, v_k] \), and returns the list \( [v_0, v_1, \ldots, v_k] \). In Rosette 4, cons takes as input any two values and returns a pair, which represents a list when the second argument is list. After we adjust the semantics of cons in Rosette 4 to match that of Leanette, we find that the two evaluators behave equivalently on all programs in \( T \), as shown in the last column of Table 1.

Table 1 also shows two sets of metrics that characterize the performance of these evaluators. The “time” columns display the maximum and average evaluation time for all programs in a bucket. The “state” columns show the number of non-trivial symbolic states produced by the evaluator, as well
We evaluate the utility of SymPro work on SymPro [Bornholt and Torlak 2018] and Jitterbug [Nelson et al. 2020]. SymPro is a tool for profiling the performance of symbolic evaluators, and its benchmarks are drawn from the evaluation suites of 15 published verification and synthesis tools. Each SymPro benchmark applies one of these tools to its slowest available input(s). The JITTERBUG benchmarks are drawn from the evaluation suite of JITTERBUG, a framework for developing and verifying just-in-time

<table>
<thead>
<tr>
<th>Injected mistake ($\beta$)</th>
<th>Location</th>
<th>Caught?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make and($b_a$, $b_b$) = $b_a$ where $b_a$ and $b_b$ are non-literal symbolic booleans</td>
<td>symbolic factory</td>
<td>✓</td>
</tr>
<tr>
<td>Make or($b$, $b$) = $tt$ where $b$ is a non-literal symbolic boolean</td>
<td>symbolic factory</td>
<td>✓</td>
</tr>
<tr>
<td>Make $d$p($z_a$, $z_b$) = ($z_b &lt; z_a$) where either $z_a$ or $z_b$ is a non-literal symbol</td>
<td>symbolic factory</td>
<td>✓</td>
</tr>
<tr>
<td>In CALL, change the guards for evaluation of the body to be $tt$</td>
<td>symbolic evaluator</td>
<td>✓</td>
</tr>
<tr>
<td>In CALLBAD, use the input symbolic state without asserting $y$</td>
<td>symbolic evaluator</td>
<td>✓</td>
</tr>
<tr>
<td>In LET, do not bind any variable</td>
<td>symbolic evaluator</td>
<td>✓</td>
</tr>
<tr>
<td>In IFSYM, swap not(truth($\hat{v}$)) and truth($\hat{v}$) when calling merge$_R$($\cdot$, $\cdot$)</td>
<td>symbolic evaluator</td>
<td>✓</td>
</tr>
<tr>
<td>In IFTRUE and IFFALSE, swap the conditions</td>
<td>symbolic evaluator</td>
<td>✓</td>
</tr>
<tr>
<td>In ERR, do ABORT</td>
<td>symbolic evaluator</td>
<td>✓</td>
</tr>
<tr>
<td>In ABORT, do Err</td>
<td>symbolic evaluator</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2. Coverage results of the randomly generated programs. Each of the injected faults $\beta$ violates soundness. “✓” indicates that the injected fault is caught by the generated test set $T$ (i.e., ROSETTE 4 and BADLEANETE($\beta$) disagree on at least one program in $T$).

as the maximum and average size of the non-trivial states in each bucket. A state is trivial if both of its components are constant ($ff$ or $tt$). The size of a state is the number of nodes in the graph representation of its components. In particular, both LEANETE and ROSETTE 4 represent symbolic booleans as abstract syntax graphs. These graphs are trees in LEANETE and DAGs in ROSETTE 4.

The data in Table 1 exhibits two notable trends. First, a large fraction of the computed states are trivial: 60% for LEANETE and 80% for ROSETTE 4. This trend is expected for randomly generated programs, which have a high probability of containing trivial errors (e.g., type errors). Second, ROSETTE 4 is orders-of-magnitude more efficient than LEANETE in terms of both running time and state size. This trend is also expected because ROSETTE 4 uses a heavily optimized symbolic factory, while LEANETE uses a na"ive one. In the worst case, LEANETE generates a symbolic state that is exponential in program size. ROSETTE 4, in contrast, generates polynomially sized states.

Coverage. To gain further confidence in our test results, we inject 11 faults into LEANETE, one at a time, and check that they are uncovered by at least one test in $T$. For each fault $\beta$, we create a new evaluator BADLEANETE($\beta$) that applies $\beta$ to LEANETE, and we test ROSETTE 4 against BADLEANETE($\beta$) on $T$. Table 2 describes the faults and the results of the differential testing. Three faults are in the symbolic factory, and the rest are in the implementation of the $S_c$ rules. Our test suite discovers all of them, giving us confidence that ROSETTE 4 correctly implements $S_c$.

7 UTILIT Y AND PERFORMA NCE OF AN $S_c$ EVALUATOR: EXPERIMENTS

This section evaluates the utility and performance of ROSETTE 4 by comparing them to those of ROSETTE 3. Our evaluation aims to answer the following research questions:

(1) Can the interface provided by an $S_c$ evaluator simplify the implementation of client tools?
(2) Can an $S_c$ evaluator match the performance of a classic evaluator when used within state-of-the-art solver-aided tools?

We conduct two sets of experiments and find a positive answer to both questions.

7.1 Comparing the Interface of ROSETTE 4 and ROSETTE 3

We evaluate the utility of $S_c$ by porting two sets of benchmarks to ROSETTE 4, developed in prior work on SymPro [Bornholt and Torlak 2018] and Jitterbug [Nelson et al. 2020]. SymPro is a tool for profiling the performance of symbolic evaluators, and its benchmarks are drawn from the evaluation suites of 15 published verification and synthesis tools. Each SymPro benchmark applies one of these tools to its slowest available input(s). The JITTERBUG benchmarks are drawn from the evaluation suite of JITTERBUG, a framework for developing and verifying just-in-time
Table 3. Differences between the implementations of the ported and original benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Rosette 3 LoC</th>
<th>Rosette 4 LoC</th>
<th>LoC diff</th>
<th>Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagpipe [Weitz et al. 2016]</td>
<td>2,643</td>
<td>2,643</td>
<td>+2 -2</td>
<td>Z3</td>
</tr>
<tr>
<td>Bonsai [Chandra and Bodik 2018]</td>
<td>437</td>
<td>437</td>
<td>+1 -1</td>
<td>Boolector</td>
</tr>
<tr>
<td>Cosette [Chu et al. 2017]</td>
<td>774</td>
<td>774</td>
<td>+0 -0</td>
<td>Z3</td>
</tr>
<tr>
<td>Ferrite [Bornholt et al. 2016]</td>
<td>348</td>
<td>348</td>
<td>+2 -2</td>
<td>Z3</td>
</tr>
<tr>
<td>Fluidics [Willeys et al. 2019]</td>
<td>98</td>
<td>98</td>
<td>+0 -0</td>
<td>Z3</td>
</tr>
<tr>
<td>IFCL [Tarlak and Bodik 2014]</td>
<td>483</td>
<td>483</td>
<td>+13 -13</td>
<td>Boolector</td>
</tr>
<tr>
<td>Neutrons [Pernsteiner et al. 2016]</td>
<td>37,174</td>
<td>37,174</td>
<td>+2 -2</td>
<td>Z3</td>
</tr>
<tr>
<td>Nonograms [Butler et al. 2017]</td>
<td>3,368</td>
<td>3,374</td>
<td>+11 -5</td>
<td>Z3</td>
</tr>
<tr>
<td>Quivela [Amazon Web Services 2018]</td>
<td>1,010</td>
<td>1,009</td>
<td>+10 -11</td>
<td>Z3</td>
</tr>
<tr>
<td>RTR [Kazeroonian et al. 2018]</td>
<td>1,649</td>
<td>1,640</td>
<td>+12 -21</td>
<td>CVC4</td>
</tr>
<tr>
<td>SynthCL [Tarlak and Bodik 2014]</td>
<td>2,257</td>
<td>2,256</td>
<td>+42 -43</td>
<td>Boolector</td>
</tr>
<tr>
<td>Wallingford [Borning 2016]</td>
<td>2,532</td>
<td>2,533</td>
<td>+12 -11</td>
<td>Z3</td>
</tr>
<tr>
<td>WebSynth [Tarlak and Bodik 2014]</td>
<td>1,181</td>
<td>1,189</td>
<td>+14 -6</td>
<td>Z3</td>
</tr>
</tbody>
</table>

(JIT) compilers for the Berkeley Packet Filter (BPF) [Fleming 2017] language in the Linux kernel. JITTERBUG performs verification on each BPF opcode individually, which amounts to 668 verification queries across the six architectures supported by JITTERBUG (Arm32, Arm64, RV32, RV64, x86-32, and x86-64). All tools in our benchmark sets were originally developed in Rosette 3.

As we saw in Section 6.2, Rosette 4 extends the Rosette 3 language to include assume expressions, and it exposes different constructs for symbolic reflection. To port our benchmarks to Rosette 4, we adapted their code to use these new constructs and the extended language. We detail the porting effort for the SymPro benchmarks next, and then describe how Rosette 4 enabled us to simplify the implementation of JITTERBUG.

7.1.1 Porting SymPro Benchmarks. Table 3 summarizes the differences between the ported and the original code for each benchmark. The first four columns report the line count for both implementations, and the number of insertions and deletions by which the ported code differs from the original code. The last column shows which SMT solver is used for a given benchmark: Z3 v4.8.8 [De Moura and Bjørner 2008], Boolector v3.2.1 [Niemetz et al. 2014], or CVC4 v1.8 [Barrett et al. 2011].

As the data in Table 3 indicates, porting the SymPro benchmarks to Rosette 4 required relatively few changes to their code. Most changes were mechanical: we either adapted the code to use the new symbolic reflection constructs, or replaced #:assume clauses with equivalent assume expressions. One exception is the RTR tool [Kazeroonian et al. 2018], which we simplified and made faster (see Table 4a) using assume expressions. RTR is a type checker for a refinement type system for Ruby, and it works on an intermediate verification language that includes assume statements. Since first-class assumptions are not available in Rosette 3, RTR implements assume statements as early exits that return a special value. We removed this custom code and implemented the assume statement directly using the assume expression. It took one author 4 days to port all 15 tools to Rosette 4.

7.1.2 Porting JITTERBUG. JITTERBUG is the most complex code base we ported to Rosette 4 (see Table 3). At the core of JITTERBUG’s proof strategy is its per-instruction correctness specification: given a source instruction and a JIT context (e.g., compiler configurations), JITTERBUG proves that the execution of the target instructions produced by the JIT exhibits the same behavior as that of the source instruction. To do so, JITTERBUG first symbolically evaluates the JIT implementation
with a BPF instruction and a JIT context to produce a symbolic representation of all possible sequences of target instructions. Next, it symbolically evaluates both the source instruction and target instructions on source and target states, respectively, to encode their semantics. Finally, it checks that the resulting source and target states are equivalent, assuming that the original source and target states are equivalent. This requires Jitterbug to model the assumptions made by the JIT, such as the well-formedness of BPF instructions and the memory layout of the Linux kernel.

Since Rosette 3 does not support assumptions, Jitterbug implements its own system for tracking assumptions. The system works by escaping to Racket to capture and store the assumptions outside of symbolic evaluation. It then reintroduces them as preconditions in later assertions. This process is subtle and does not preserve legality, requiring careful manual reasoning to ensure that the final verification query, which takes the form $pre \land \neg post$, is sound (c.f., Example 7).

When porting Jitterbug to Rosette 4, we replaced this workaround with `assume` expressions. This change simplified both the Jitterbug implementation and the formulation of the top-level correctness specification, which no longer has to recover and incorporate the stored assumptions. The legality guarantee provided by Rosette 4 also increased the confidence in the soundness of the verification queries emitted by Jitterbug. The porting process took one author 2 weeks. Our experience shows that the interface exposed by Rosette 4 can simplify the implementation of a complex client tool, and that the developer burden to utilize these new features is low.

### 7.2 Comparing the Performance of Rosette 4 and Rosette 3

To evaluate the performance of $S_c$, we compare the running time and encoding size of Rosette 4 and Rosette 3 on the ported and original version of each benchmark, respectively. We collected all performance data using Racket v8.1 on an Intel Core i5-6600K at 3.50 GHz with 16 GB of RAM. Table 4a shows the results for each SymPro benchmark, and Table 4b summarizes the results across all Jitterbug benchmarks. Both figures report the total time, symbolic evaluation time, solving time, and encoding size under Rosette 4 and Rosette 3. The encoding size is given as the number of symbolic terms generated during symbolic evaluation. This number overapproximates the size of the encoding sent to the solver and offers a rough measure of the total amount of work performed by the evaluator. The symbolic evaluation time includes both the time spent to generate the encoding and to pipe it to the solver. The solving time is computed as the difference between the wall clock time and the CPU time consumed by Racket. Finally, Table 4c reports the best, worst, and average ratio of the Rosette 4 and Rosette 3 performance across all metrics and benchmarks. For example, comparing the symbolic evaluation time of the ported and original SymPro benchmarks, we find that the ported ones are up to $2.1 \times$ slower and up to $3.8 \times$ faster than the original ones, with an average speedup of 13%.

The analysis in Table 4c shows that, on average, Rosette 4 generates 6–9% more terms than Rosette 3, taking roughly 10–20% less time to do so, and producing an encoding that is about 5–9% faster to solve. To understand the difference in the number of generated terms, recall from Section 4.3 that $S_c$ updates the assumption component of the symbolic state using $\text{assume}(\sigma, \hat{b})$, while the classic merging semantics uses $\text{assume}^\prime(\sigma, \hat{b})$. In the worst case, the symbolic factory for Rosette 4 and Rosette 3 creates 3 terms to represent assumes(assume($\sigma, \hat{b}$)) and only 1 term to represent assumes(assume($\sigma, \hat{b}$)). Additionally, Rosette 3 uses a simpler function for merging two symbolic states, which generates no new terms, while the $S_c$ function $\text{merge}_2(\sigma, G)$ can generate up to $4|G| + 2$ terms. These two differences are the main reason why Rosette 4 emits up to $8 \times$ more terms than Rosette 3 in our benchmarks. We avoid this blowup in the average case by specializing Rosette 4 with rewriting rules tailored for the terms generated by $\text{assume}(\sigma, \hat{b})$ and $\text{merge}_2(\sigma, G)$.

Given that Rosette 4 emits more terms than Rosette 3, it may seem surprising that it is, on average, slightly faster than Rosette 3, and that the resulting formula is slightly easier to solve. In
Table 4. Performance results for the ported and original SymPro (4a) and Jitterbug (4b) benchmarks, along with statistics (4c) summarizing the performance ratios between the ported and original code. In 4a and 4b, the columns “Total (s)”, “Eval. (s)”, and “Solving (s)” are the elapsed wall clock time, symbolic evaluation time, and solving time respectively. The column “Terms ($\times 10^3$)” indicates the encoding size. The average ratio in 4c is the geometric mean of the per-benchmark ratios for a given performance metric.

(a) Performance results for the 15 ported and original SymPro benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Rosette 3</th>
<th></th>
<th></th>
<th></th>
<th>Rosette 4</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total (s)</td>
<td>Eval. (s)</td>
<td>Solving (s)</td>
<td>Terms ($\times 10^3$)</td>
<td>Total (s)</td>
<td>Eval. (s)</td>
<td>Solving (s)</td>
<td>Terms ($\times 10^3$)</td>
</tr>
<tr>
<td>Bagpipe</td>
<td>23</td>
<td>22</td>
<td>&lt; 1</td>
<td>47</td>
<td>23</td>
<td>23</td>
<td>&lt; 1</td>
<td>48</td>
</tr>
<tr>
<td>Bonsai</td>
<td>20</td>
<td>18</td>
<td>2</td>
<td>664</td>
<td>40</td>
<td>37</td>
<td>3</td>
<td>1,222</td>
</tr>
<tr>
<td>Cosette</td>
<td>15</td>
<td>7</td>
<td>7</td>
<td>131</td>
<td>15</td>
<td>8</td>
<td>8</td>
<td>154</td>
</tr>
<tr>
<td>Ferrite</td>
<td>19</td>
<td>12</td>
<td>7</td>
<td>34</td>
<td>26</td>
<td>12</td>
<td>14</td>
<td>40</td>
</tr>
<tr>
<td>Fluidics</td>
<td>19</td>
<td>6</td>
<td>13</td>
<td>284</td>
<td>23</td>
<td>7</td>
<td>17</td>
<td>308</td>
</tr>
<tr>
<td>GreenThumb</td>
<td>53</td>
<td>7</td>
<td>46</td>
<td>239</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>74</td>
</tr>
<tr>
<td>IFCL</td>
<td>157</td>
<td>10</td>
<td>147</td>
<td>383</td>
<td>119</td>
<td>10</td>
<td>109</td>
<td>438</td>
</tr>
<tr>
<td>MemSynth</td>
<td>20</td>
<td>18</td>
<td>2</td>
<td>61</td>
<td>22</td>
<td>20</td>
<td>2</td>
<td>163</td>
</tr>
<tr>
<td>Neutrons</td>
<td>36</td>
<td>36</td>
<td>&lt; 1</td>
<td>444</td>
<td>10</td>
<td>10</td>
<td>&lt; 1</td>
<td>172</td>
</tr>
<tr>
<td>Nonograms</td>
<td>9</td>
<td>3</td>
<td>5</td>
<td>51</td>
<td>10</td>
<td>3</td>
<td>7</td>
<td>73</td>
</tr>
<tr>
<td>Quivela</td>
<td>31</td>
<td>29</td>
<td>2</td>
<td>1,113</td>
<td>34</td>
<td>31</td>
<td>4</td>
<td>1,340</td>
</tr>
<tr>
<td>RTR</td>
<td>329</td>
<td>312</td>
<td>18</td>
<td>741</td>
<td>113</td>
<td>82</td>
<td>32</td>
<td>1,106</td>
</tr>
<tr>
<td>SynthCL</td>
<td>258</td>
<td>13</td>
<td>246</td>
<td>726</td>
<td>253</td>
<td>15</td>
<td>238</td>
<td>732</td>
</tr>
<tr>
<td>Wallingford</td>
<td>5</td>
<td>&lt; 1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>&lt; 1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>WebSynth</td>
<td>10</td>
<td>10</td>
<td>&lt; 1</td>
<td>1,012</td>
<td>16</td>
<td>16</td>
<td>&lt; 1</td>
<td>778</td>
</tr>
</tbody>
</table>

(b) Performance results for the 668 ported and original Jitterbug benchmarks.

| Evaluator | Total (s) |       |       |       | Solving (s) |       |       |       | Terms ($\times 10^3$) |       |       |       |
|           | mean     | med.  | max   | mean   | med.  | max   | mean   | med.  | max   | mean     | med.  | max   | mean     |
| Rosette 3 | 50       | 22    | 9,963 | 2      | 1     | 69    | 48     | 20    | 9,894 | 119      | 15    | 1,678 |
| Rosette 4 | 38       | 20    | 4,100 | 1      | 1     | 73    | 36     | 19    | 4,027 | 120      | 23    | 1,837 |

(c) Performance ratios between the ported and original code for SymPro and Jitterbug benchmarks.

| Benchmark | Total |       |       |       | Solving |       |       |       | Terms |       |       |       |
| SymPro    | 0.07  | 1.99  | 0.81  | 0.26  | 2.10  | 0.87  | 0.04  | 2.21  | 0.95  | 0.31  | 2.65  | 1.06  |
| Jitterbug | 0.23  | 6.03  | 0.91  | 0.33  | 2.18  | 0.87  | 0.22  | 6.48  | 0.91  | 0.79  | 8.12  | 1.09  |

In general, these differences are difficult to fully explain since they depend on many factors, and the behavior of both systems is sensitive to small initial differences, e.g., a factory simplification that triggers in one system but not the other. With this caveat in mind, our experience suggests that the observed difference is due to two factors. First, at every point during and after the evaluation, Rosette 4 operates on a more constrained symbolic state than Rosette 3. Both of its state components carry terms that are not available in Rosette 3 and that may trigger additional simplifications. Second, thanks to legality, Rosette 4 emits the query $\text{assumes}(\sigma) \land \neg\text{asserts}(\sigma)$ (Definition 7), while Rosette 3 emits $\neg\text{asserts}(\sigma)$. It is sound for Rosette 4 to emit just $\neg\text{asserts}(\sigma)$, but we find that the redundant formula tends to elicit better performance in practice; for example, the Jitterbug benchmarks run on average 25% slower without the redundant formula. Overall, even
though Rosette 4 produces a larger encoding than Rosette 3, our results show that it matches the runtime performance of Rosette 3 in a wide range of tools.

8 CONCLUSION
This paper presented the first formal framework for reasoning about reusable symbolic evaluators. The framework is based on $S_c$, a new symbolic semantics with merging. This semantics is defined with respect to the symbolic factory interface, which abstracts away the details of how symbolic values are represented, created, and manipulated. As such, $S_c$ admits a wide range of implementations. We use Lean to prove that $S_c$ is sound and complete with respect to the concrete semantics of its target language, $\lambda_c$, which extends Core Scheme with assumptions and assertions. We also prove that $S_c$ preserves two properties, legality and reducibility, that are key to reusing a symbolic evaluator in a client tool. Leanette and Rosette 4, two implementations of $S_c$ in Lean and Racket, respectively, show that $S_c$ provides a general contract for building and validating reusable evaluators. By porting 16 published verification and synthesis tools from Rosette 3 to Rosette 4, we demonstrate that $S_c$ provides a practical interface for client tools: Rosette 4 both simplifies the implementation of two of the benchmarks and matches the performance of Rosette 3 across these tools. All source code accompanying this paper is publicly available at https://github.com/emina/rosette.

ACKNOWLEDGMENTS
We thank the anonymous reviewers for their helpful feedback. This work is supported by the National Science Foundation under Grant Nos. CCF-1651225, CCF-1836724, CNS-1844807, and by a gift from the VMware University Research Fund.

REFERENCES
A Formal Foundation for Symbolic Evaluation with Merging


Proc. ACM Program. Lang., Vol. 6, No. POPL, Article 47. Publication date: January 2022.