A formal foundation for symbolic evaluation with merging

ANONYMOUS AUTHOR(S)

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*, an optimized evaluator written in Racket. We prove LEANETTE correct with respect to the semantics, and validate ROSETTE* against LEANETTE via solver-aided differential testing. To demonstrate the practicality of our approach, we port 16 published verification and synthesis tools from ROSETTE to ROSETTE*. ROSETTE is an existing reusable evaluator that implements the classic merging semantics, adopted from bounded model checking. ROSETTE* replaces the semantic core of ROSETTE but keeps its optimized symbolic factory. Our results show that ROSETTE* matches the performance of ROSETTE 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

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 employ *reusable symbolic evaluators* [Sen et al. 2015; Torlak and Bodik 2014; Uhler and Dave 2014] to encode the semantics of a program into a logical formula. The resulting formula is then used 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

2022. 2475-1421/2022/1-ART1 $15.00
https://doi.org/

targets a small but expressive language, λ_c, that extends Core Scheme [Flanagan et al. 1993] with assumptions and assertions. As such, λ_c includes the core features supported by existing reusable evaluators: branching, loops, (first-class) procedures, and specification constructs. Our symbolic semantics for λ_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 the correctness and reusability 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.

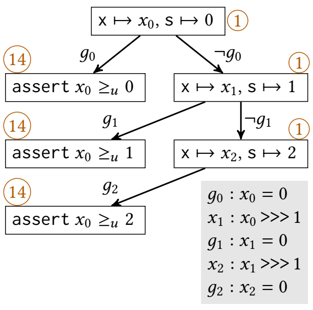
We address this challenge by defining $\mathcal{S}_c$ with respect to a *symbolic factory*, and proving that it is *partially correct* with respect to the concrete semantics of λ_c. A symbolic factory is a parameter to the semantics, consisting of functions that abstract away the details of creation, 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 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.

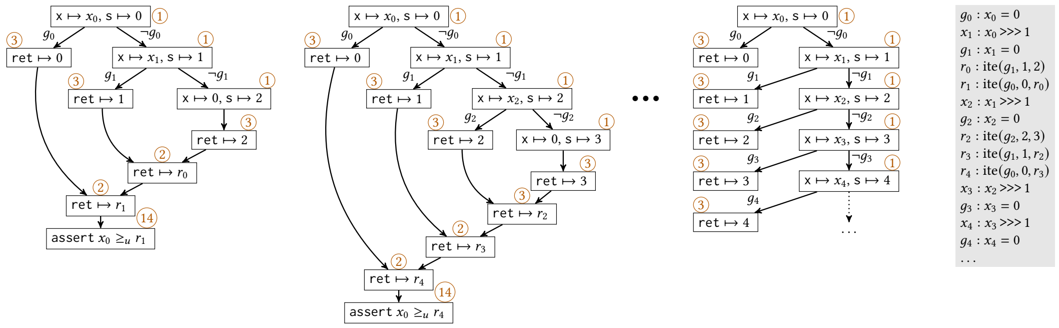
Reducibility states that symbolic evaluation terminates on a given program and a fully concrete input whenever the concrete semantics does so. This property is trivially satisfied by all symbolic evaluators on loop-free programs, but only symbolic execution is designed to reduce to concrete execution 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, $\mathcal{S}_c$ 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]).

```
 1 (define (addbits x s)
 2   (if (bvzero? x)                          ; If x = 0
 3       s                                    ; then s
 4       (addbits                             ; else addbits
 5        (bvlshr x (bv 1 n))                 ;  (x >>> 1)
 6        (bvadd (bvand x (bv 1 n)) s))))     ;  ((x & 1) + s).
 7
 8 (define (popcount x)      ; Returns the number of 1 bits
 9   (addbits x (bv 0 n)))   ; in the n-bit bitvector x.
10
11 (define-symbolic x (bitvector n)) ; Symbolic n-bit bitvector.
12
13 (verify                      ; Verify that for every n-bit
14  (assert                     ; bitvector x, x ≥u (popcount x),
15   (bvuge x (popcount x)))) ; where ≥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. Circled numbers refer to lines in the toy program.

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 σ is characterized by two formulas, which the client tools use to construct their queries. One encodes the assumptions, assumes(σ), and the other encodes the assertions, asserts(σ), that have been made to reach the state σ. 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 assumes(σ) and asserts(σ) so that every state σ 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 assumes(σ) $\rightarrow$ asserts(σ) ensures the absence of errors.

To demonstrate the suitability of our framework for implementing and validating reusable evaluators, we implement $\mathcal{S}_c$ 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 $\mathcal{S}_c$, 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*. Rosette's default evaluator is based on the classic merging semantics, and we refer to it as ROSETTE. Both evaluators use the same optimized symbolic factory. We validate

Rosette* 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 $\mathcal{S}_c$, we port 16 published verification and synthesis tools to Rosette*, and compare the performance of the ported code to the original code built on top of Rosette. 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* is up to 14× faster than Rosette across all benchmarks. The largest slowdown we observe is 6×. On average, Rosette* is 10–20% faster than Rosette. Our evaluation also shows that the legality-preserving interface exposed by Rosette* 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 $\mathcal{S}_c$ merging semantics and compares it to the classic one. Section 5 states our correctness criterion and shows that $\mathcal{S}_c$ satisfies it. Section 6 describes our Lean and Rosette implementations of $\mathcal{S}_c$, 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 in theorem provers by recognizing a correspondence between concrete and symbolic paths in the absence of state merging [Lucanu et al. 2017].

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, $\mathcal{S}_c$ 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, $\mathcal{S}_c$ 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 functions, assumptions, and assertions, which are missing from GL. Finally, $\mathcal{S}_c$ is formalized as a big-step operational semantics, via an inductive predicate in Lean, so $\mathcal{S}_c$ evaluators do not need to use fuel to bound executions, as GL evaluators do. This gives practical implementations of $\mathcal{S}_c$ the freedom to let their client tools ensure termination (through self-finitization [Torlak and Bodik 2014]), instead of imposing a priori finitization bounds.

3 λ_c: A CORE LANGUAGE FOR REUSABLE SYMBOLIC EVALUATORS

This section presents λ_c, a core language for reusable symbolic evaluators. We begin by describing the syntax, concrete semantics, and key features of λ_c. We then define what it means for a λ_c 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 λ_c 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 λ_c.

3.1 The syntax and concrete semantics of λ_c

Figure 2 shows the syntax of λ_c. 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 λ_c expressions. Compared to Core Scheme, we place a light restriction on the syntax of λ_c 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 λ_c program by using **let** expressions; e.g., the application expression $(e_1\ e_2)$ becomes $(\textbf{let}\ (x_1\ e_1)\ (\textbf{let}\ (x_2\ e_2)\ (x_1\ x_2)))$. Like Core Scheme,

$$
\begin{array}{lrl}
\text{Expression} & e ::= & \texttt{\#t} \mid \texttt{\#f} \mid d \mid (\lambda x.e) \mid x \mid \\
 & & (o\ x_1 \ldots x_n) \mid (x_1\ x_2) \mid (\textbf{let}\ (x\ e_1)\ e_2) \mid \\
 & & (\textbf{if}\ x\ e_1\ e_2) \mid (\textbf{error}) \mid (\textbf{abort})
\end{array}
\qquad
\begin{array}{ll}
\text{Variable} & x, y, \ldots \\
\text{Constant} & d \in D \\
\text{Operator} & o \in O
\end{array}
$$

Fig. 2. Syntax for λ_c, parameterized by the set of primitive values D and operators O.

$$
\textsc{Literal}\ \frac{v \in B \cup D}{\langle v, E\rangle \Downarrow \mathsf{Ans}(v)}
\qquad
\textsc{Closure}\ \frac{}{\langle (\lambda x.e), E\rangle \Downarrow \mathsf{Ans}(\langle \textbf{cl}\ x, e, E\rangle)}
\qquad
\textsc{Variable}\ \frac{x \in \mathsf{dom}(E)}{\langle x, E\rangle \Downarrow \mathsf{Ans}(E[x])}
$$

$$
\textsc{CallOp}\ \frac{\langle x_1, E\rangle \Downarrow \mathsf{Ans}(v_1) \ldots \langle x_n, E\rangle \Downarrow \mathsf{Ans}(v_n)}{\langle (o\ x_1 \ldots x_n), E\rangle \Downarrow op(o, v_1, \ldots, v_n)}
\qquad
\textsc{CallBad}\ \frac{\langle x_1, E\rangle \Downarrow \mathsf{Ans}(v_1) \quad \langle x_2, E\rangle \Downarrow \mathsf{Ans}(v_2) \quad v_1 \notin C}{\langle (x_1\ x_2), E\rangle \Downarrow \mathsf{Err}}
$$

$$
\textsc{Call}\ \frac{\langle x_1, E\rangle \Downarrow \mathsf{Ans}(\langle \textbf{cl}\ x, e, E_1\rangle) \quad \langle x_2, E\rangle \Downarrow \mathsf{Ans}(v_2) \quad \langle e, E_1[x \mapsto v_2]\rangle \Downarrow r}{\langle (x_1\ x_2), E\rangle \Downarrow r}
$$

$$
\textsc{Let}\ \frac{\langle e_1, E\rangle \Downarrow \mathsf{Ans}(v_1) \quad \langle e_2, E[x \mapsto v_1]\rangle \Downarrow r}{\langle (\textbf{let}\ (x\ e_1)\ e_2), E\rangle \Downarrow r}
\qquad
\textsc{LetHalt}\ \frac{\langle e_1, E\rangle \Downarrow r \quad r = \mathsf{Err} \lor r = \mathsf{Abt}}{\langle (\textbf{let}\ (x\ e_1)\ e_2), E\rangle \Downarrow r}
$$

$$
\textsc{IfTrue}\ \frac{\langle x, E\rangle \Downarrow \mathsf{Ans}(v) \quad v \neq \texttt{\#f} \quad \langle e_1, E\rangle \Downarrow r}{\langle (\textbf{if}\ x\ e_1\ e_2), E\rangle \Downarrow r}
\qquad
\textsc{IfFalse}\ \frac{\langle x, E\rangle \Downarrow \mathsf{Ans}(v) \quad v = \texttt{\#f} \quad \langle e_2, E\rangle \Downarrow r}{\langle (\textbf{if}\ x\ e_1\ e_2), E\rangle \Downarrow r}
$$

$$
\textsc{Error}\ \frac{}{\langle (\textbf{error}), E\rangle \Downarrow \mathsf{Err}}
\qquad
\textsc{Abort}\ \frac{}{\langle (\textbf{abort}), E\rangle \Downarrow \mathsf{Abt}}
$$

$$
E \in \mathbb{E} ::= \text{Variable} \rightharpoonup V \quad v \in V ::= b \mid d \mid c \quad b \in B ::= \texttt{\#t} \mid \texttt{\#f} \quad c \in C ::= \langle \textbf{cl}\ x, e, E\rangle \quad r \in R ::= \mathsf{Ans}(v) \mid \mathsf{Err} \mid \mathsf{Abt}
$$

Fig. 3. Concrete semantics for λ_c, parameterized by the set of primitive values D, operators O, and function $op : O \to V^* \to R$, which gives meaning to the operators $o \in O$. The notation V^* stands for a sequence of values.

the syntax of λ_c 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 λ_c. 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 $E \in \mathbb{E}$ 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 $\mathsf{Ans}(v)$ if the evaluation terminates normally and produces the value v; Err if it errors; and Abt if it aborts.

The evaluation rules for λ_c 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 $op : O \to V^* \to R$, which is a parameter to the semantics of λ_c. 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 $\langle \textbf{cl}\ x, e, E_1\rangle$. 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 $(\textbf{if}\ x\ e_1\ e_2)$ evaluates to the result of e_1 if x is not bound 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 λ_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

[1]Our formalization also includes lists but we omit them from the presentation for brevity.

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 bind 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 λ_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:* $\forall e, E, r_1, r_2.\, (\langle e, E\rangle \Downarrow \hat{r}_1 \land \langle e, E\rangle \Downarrow \hat{r}_2) \longrightarrow \hat{r}_1 = \hat{r}_2$.

Example 1. To illustrate the syntax and semantics of λ_c, consider an instantiation $\lambda_{\mathbb{Z}}$ 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:

```
(let (z 0)
(let (abs (λx . (let (b0 (< x z)) (if b0 (- x) x))))
(let (b1 (= y z))
(let (_  (if b1 (abort) #t))    ; assume y != 0
(let (y0 (abs y))
(let (b2 (< z y0))
(let (_  (if b2 #t (error)))    ; assert |y| > 0
  _)))))))
```

The program, e_{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 $\langle e_{\text{abs}}, \{\texttt{y} \mapsto -1\}\rangle \Downarrow \text{Ans}(\texttt{\#t})$ and $\langle e_{\text{abs}}, \{\texttt{y} \mapsto 0\}\rangle \Downarrow \text{Abt}$.

3.2 The angelic execution and verification queries for λ_c

Given the syntax and semantics of λ_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 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 λ_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 λ_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 **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* **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* $\mathrm{verify}(e, \mathcal{E})$ *diverges or produces one of two results: either an environment $E \in \mathcal{E}$ such that* $\mathrm{errors}(e, E)$*, or* **unsat** *if no such environment exists in $\mathcal{E}$.*

Example 2. To illustrate Definitions 1–4, consider the program e_{abs} from Example 1. This program is correct with respect to $\mathcal{E}_{\mathbb{Z}}$, 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, $\mathrm{verify}(e_{\mathrm{abs}}, \mathcal{E}_{\mathbb{Z}}) =$ **unsat**, and $\mathrm{guess}(e_{\mathrm{abs}}, \mathcal{E}_{\mathbb{Z}})$ may return any environment that binds y to a non-zero value, e.g., $\{\mathtt{y} \mapsto -1\}$. But e_{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 $\mathrm{verify}(e_{\mathrm{abs}}, \mathcal{E}_V)$ may return any of these environments, e.g., $\{\mathtt{y} \mapsto \mathtt{\#t}\}$. Since every environment that is in $\mathcal{E}_V$ but not in $\mathcal{E}_{\mathbb{Z}}$ leads to an error, $\mathrm{guess}(e_{\mathrm{abs}}, \mathcal{E}_V)$ and $\mathrm{guess}(e_{\mathrm{abs}}, \mathcal{E}_{\mathbb{Z}})$ draw their outputs from the same non-empty subset of $\mathcal{E}_{\mathbb{Z}}$. The next section shows how to automate these queries using our symbolic semantics for λ_c.

4 $\mathcal{S}_c$: A SEMANTICS FOR SYMBOLIC EVALUATION WITH MERGING

To automate angelic execution and verification, solver-aided tools rely on symbolic evaluation to express the $\mathrm{guess}(e, \mathcal{E})$ and $\mathrm{verify}(e, \mathcal{E})$ queries as logical formulas. This section presents $\mathcal{S}_c$, a new symbolic semantics for reducing λ_c programs to formulas. To accommodate a wide range of practical implementations, $\mathcal{S}_c$ 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 $\mathcal{S}_c$, and contrast them to the classic merging semantics [Biere et al. 1999]. We conclude this section by defining $\mathrm{guess}(e, \mathcal{E})$ and $\mathrm{verify}(e, \mathcal{E})$ in terms of the symbolic states computed by $\mathcal{S}_c$. The next section establishes the correctness of $\mathcal{S}_c$ 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 $\mathcal{S}_c$ 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 λ_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 λ_c. They serve as the symbolic counterparts of the concrete types, functions, and predicates that are used to define the concrete evaluation rules for λ_c (Figure 3). We first explain how the factory components help lift these rules, and then illustrate a toy factory for the language $\lambda_{\mathbb{Z}}$ from Example 1.

Lifting constants. The function $\mathrm{lift} : (B \cup D \cup \hat{C}) \to \hat{V}$ provides a way to lift the meaning of constant expressions (Literal and Closure in Figure 3). Given an input i, $\mathrm{lift}(i)$ returns a symbolic value that has the same interpretation as i under all models. The input i is either a concrete boolean, a concrete

Symbolic values	$\hat{v} \in \hat{V} \qquad [\![\cdot]\!]^{\hat{V}} : M \to \hat{V} \to V$
$\mathsf{truth} : \hat{V} \to \hat{B}$	$\forall m, \hat{v}. [\![\mathsf{truth}(\hat{v})]\!]^{\hat{B}}_m \leftrightarrow ([\![\hat{v}]\!]^{\hat{V}}_m \neq \#\mathsf{f})$
$\mathsf{lift} : (B \cup D \cup \hat{C}) \to \hat{V}$	$\forall m, b. [\![\mathsf{lift}(b)]\!]^{\hat{V}}_m = b \quad \forall m, d. [\![\mathsf{lift}(d)]\!]^{\hat{V}}_m = d \quad \forall m, \hat{c}. [\![\mathsf{lift}(\hat{c})]\!]^{\hat{V}}_m = [\![\hat{c}]\!]^{\hat{C}}_m$
$\mathsf{merge} : \mathbb{G}_{\hat{V}} \to \hat{V}$	$\forall m, G.\, \mathsf{one}_m(G) \to [\![\mathsf{merge}(G)]\!]_m = [\![G]\!]^{\mathbb{G}_{\hat{V}}}_m$
$\mathsf{cast} : \hat{V} \to \mathbb{G}_{\hat{C}}$	$\forall m, \hat{v}. [\![\hat{v}]\!]_m \in C \to (\mathsf{one}_m(\mathsf{cast}(\hat{v})) \wedge [\![\mathsf{cast}(\hat{v})]\!]^{\mathbb{G}_{\hat{C}}}_m = [\![\hat{v}]\!]_m)$
	$\forall m, \hat{v}. [\![\hat{v}]\!]_m \notin C \to \mathsf{none}_m(\mathsf{cast}(\hat{v}))$
$\hat{op} : O \to \hat{V}^* \to \hat{R}$	$\forall m, \hat{v}_1, \ldots, \hat{v}_k. [\![\hat{op}(\hat{v}_1, \ldots, \hat{v}_k)]\!]^{\hat{R}}_m = op([\![\hat{v}_1]\!]_m, \ldots, [\![\hat{v}_k]\!]_m) \wedge \mathsf{legal}_m(\hat{op}(\hat{v}_1, \ldots, \hat{v}_k))$
Symbolic booleans	$\hat{b} \in \hat{B} \qquad [\![\cdot]\!]^{\hat{B}} : M \to \hat{B} \to B$
$\mathsf{tt}, \mathsf{ff} : \hat{B}$	$\forall m. [\![\mathsf{tt}]\!]_m = \#\mathsf{t} \qquad \forall m. [\![\mathsf{ff}]\!]_m = \#\mathsf{f}$
$\mathsf{not} : \hat{B} \to \hat{B}$	$\forall m, \hat{b}. [\![\mathsf{not}(\hat{b})]\!]_m = \neg [\![\hat{b}]\!]_m$
$\mathsf{and} : \hat{B} \to \hat{B} \to \hat{B}$	$\forall m, \hat{b}_1, \hat{b}_2. [\![\mathsf{and}(\hat{b}_1, \hat{b}_2)]\!]_m = ([\![\hat{b}_1]\!]_m \wedge [\![\hat{b}_2]\!]_m)$
$\mathsf{or} : \hat{B} \to \hat{B} \to \hat{B}$	$\forall m, \hat{b}_1, \hat{b}_2. [\![\mathsf{or}(\hat{b}_1, \hat{b}_2)]\!]_m = ([\![\hat{b}_1]\!]_m \vee [\![\hat{b}_2]\!]_m)$
$\mathsf{imp} : \hat{B} \to \hat{B} \to \hat{B}$	$\forall m, \hat{b}_1, \hat{b}_2. [\![\mathsf{imp}(\hat{b}_1, \hat{b}_2)]\!]_m = ([\![\hat{b}_1]\!]_m \to [\![\hat{b}_2]\!]_m)$
Symbolic closure	$\hat{c} \in \hat{C} ::= \langle \hat{\mathbf{cl}}\ x, e, \hat{E} \rangle$
$[\![\cdot]\!]^{\hat{C}} : M \to \hat{C} \to V$	$[\![\langle \hat{\mathbf{cl}}\ x, e, \hat{E} \rangle]\!]^{\hat{C}}_m ::= \langle \mathbf{cl}\ x, e, [\![\hat{E}]\!]^{\hat{\mathbb{E}}}_m \rangle$
Symbolic environment	$\hat{E} \in \hat{\mathbb{E}} ::= \mathsf{Variable} \rightharpoonup \hat{V}$
$[\![\cdot]\!]^{\hat{\mathbb{E}}} : M \to \hat{\mathbb{E}} \to \mathbb{E}$	$[\![\hat{E}]\!]^{\hat{\mathbb{E}}}_m ::= \{x \mapsto v \mid x \in \mathsf{dom}(\hat{E}), \hat{v} = \hat{E}[x], [\![\hat{v}]\!]^{\hat{V}}_m = v\}$
Guarded choice	$g \in \hat{B} \times \alpha ::= \langle \hat{b}, a \rangle \qquad \text{where } a \in \alpha, [\![a]\!]^{\alpha}_m \in \beta \qquad \mathsf{guard}(g) ::= \hat{b} \qquad \mathsf{choice}(g) ::= a$
Guarded choices	$G \in \mathbb{G}_\alpha ::= [g_1, \ldots, g_n]$
$[\![\cdot]\!]^{\mathbb{G}_\alpha} : M \to \mathbb{G}_\alpha \to \beta$	$[\![g :: G]\!]^{\mathbb{G}_\alpha}_m ::= \mathbf{if}\ [\![\mathsf{guard}(g)]\!]_m\ \mathbf{then}\ [\![\mathsf{choice}(g)]\!]^{\alpha}_m\ \mathbf{else}\ [\![G]\!]^{\mathbb{G}_\alpha}_m$
	$[\![\ [\,]\ ]\!]^{\mathbb{G}_\alpha}_m ::= \mathsf{default}(\beta)$
$\mathsf{one} : M \to \mathbb{G}_\alpha \to B$	$\mathsf{one}_m(G) ::= \mathbf{length}(\mathbf{filter}(\lambda g. [\![\mathsf{guard}(g)]\!]_m, G)) = 1$
$\mathsf{none} : M \to \mathbb{G}_\alpha \to B$	$\mathsf{none}_m(G) ::= \mathbf{length}(\mathbf{filter}(\lambda g. [\![\mathsf{guard}(g)]\!]_m, G)) = 0$
Symbolic state	$\sigma \in \Sigma ::= \langle \hat{b}_1, \hat{b}_2 \rangle \qquad \mathsf{assumes}(\sigma) ::= \hat{b}_1 \qquad \mathsf{asserts}(\sigma) ::= \hat{b}_2$
$\mathsf{normal} : M \to \Sigma \to B$	$\mathsf{normal}_m(\sigma) ::= [\![\mathsf{assumes}(\sigma)]\!]_m \wedge [\![\mathsf{asserts}(\sigma)]\!]_m$
$\mathsf{aborts} : M \to \Sigma \to B$	$\mathsf{aborts}_m(\sigma) ::= \neg [\![\mathsf{assumes}(\sigma)]\!]_m \wedge [\![\mathsf{asserts}(\sigma)]\!]_m$
$\mathsf{errors} : M \to \Sigma \to B$	$\mathsf{errors}_m(\sigma) ::= [\![\mathsf{assumes}(\sigma)]\!]_m \wedge \neg [\![\mathsf{asserts}(\sigma)]\!]_m$
$\mathsf{legal} : M \to \Sigma \to B$	$\mathsf{legal}_m(\sigma) ::= [\![\mathsf{assumes}(\sigma)]\!]_m \vee [\![\mathsf{asserts}(\sigma)]\!]_m$
$\equiv : M \to \Sigma \to \Sigma \to B$	$\sigma_1 \equiv_m \sigma_2 ::= [\![\mathsf{assumes}(\sigma_1)]\!]_m = [\![\mathsf{assumes}(\sigma_2)]\!]_m \wedge [\![\mathsf{asserts}(\sigma_1)]\!]_m = [\![\mathsf{asserts}(\sigma_2)]\!]_m$
Symbolic result	$\hat{r} \in \hat{R} ::= \mathsf{Out}(\sigma, \hat{v}) \mid \mathsf{Halt}(\sigma)$
	$\mathsf{state}(\hat{r}) ::= \sigma \qquad \mathsf{value}(\mathsf{Out}(\sigma, \hat{v})) ::= \hat{v} \qquad \mathsf{value}(\mathsf{Halt}(\sigma)) ::= \mathsf{default}(\hat{V})$
$\mathsf{Result} : \Sigma \to \hat{V} \to \hat{R}$	$\mathsf{Result}(\sigma, \hat{v}) ::= \mathbf{if}\ (\mathsf{assumes}(\sigma) = \mathsf{ff} \vee \mathsf{asserts}(\sigma) = \mathsf{ff})\ \mathbf{then}\ \mathsf{Halt}(\sigma)\ \mathbf{else}\ \mathsf{Out}(\sigma, \hat{v})$
$[\![\cdot]\!]^{\hat{R}} : M \to \hat{R} \to R$	$[\![\mathsf{Out}(\sigma, \hat{v})]\!]^{\hat{R}}_m ::= \mathbf{if}\ \mathsf{normal}_m(\sigma)\ \mathbf{then}\ \mathsf{Ans}([\![\hat{v}]\!]_m)\ \mathbf{else\ if}\ \mathsf{aborts}_m(\sigma)\ \mathbf{then}\ \mathsf{Abt}\ \mathbf{else}\ \mathsf{Err}$
	$[\![\mathsf{Halt}(\sigma)]\!]^{\hat{R}}_m ::= \mathbf{if}\ \mathsf{aborts}_m(\sigma)\ \mathbf{then}\ \mathsf{Abt}\ \mathbf{else}\ \mathsf{Err}$
$\mathsf{legal} : M \to \hat{R} \to B$	$\mathsf{legal}_m(\mathsf{Out}(\sigma, \hat{v})) ::= \mathsf{legal}_m(\sigma)$
	$\mathsf{legal}_m(\mathsf{Halt}(\sigma)) ::= \mathsf{legal}_m(\sigma) \wedge \neg \mathsf{normal}_m(\sigma)$

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 op from the definition of λ_c. We use $[\![\cdot]\!]^{\alpha}$ to denote the interpreter for values of type α, omitting the superscript α 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.

constant of type D, or a symbolic closure. A symbolic closure $\langle \hat{\mathbf{cl}}\ x, e, \hat{E} \rangle \in \hat{C}$ is a straightforward generalization of a concrete closure: it combines a procedure $(\lambda x.e)$ with a symbolic environment $\hat{E} \in \hat{\mathbb{E}}$, which maps variable names to symbolic values. Symbolic closures and environments evaluate to their concrete counterparts via the interpretation functions $[\![\cdot]\!]^{\hat{C}}$ and $[\![\cdot]\!]^{\hat{\mathbb{E}}}$. Conversely, concrete closures and environments can be made symbolic by recursively lifting their contents via $\mathsf{lift}(i)$.

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

Lifting procedure calls. The function cast : $\hat{V} \to \mathbb{G}_{\hat{C}}$ 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 $\mathsf{cast}(\hat{v})$ to make this operation explicit on symbolic values. The result of a symbolic cast is a list of guarded symbolic closures, $\mathsf{cast}(\hat{v}) \in \mathbb{G}_{\hat{C}}$. This list selects at most one closure under every model to match the behavior of $\hat{v}$: if $[\![\hat{v}]\!]_m$ is a closure, then $\mathsf{cast}(\hat{v})$ evaluates to $[\![\hat{v}]\!]_m$, and if not, all guards in $\mathsf{cast}(\hat{v})$ are false under m, indicating that the cast has failed. As an example, $\mathsf{cast}(\mathsf{merge}([\langle\hat{b}, \mathsf{lift}(\hat{c})\rangle, \langle\mathsf{not}(\hat{b}), \mathsf{lift}(\#\mathsf{f})\rangle]))$ produces a list of guarded closures that is equivalent to $[\langle\hat{b}, \hat{c}\rangle]$ under every model. This list evaluates to $[\![\hat{c}]\!]_m$ when $[\![\hat{b}]\!]_m$ is true, just like the input to the cast, and has no true guards otherwise.

Lifting operator calls. The function $\hat{op} : O \to \hat{V}^* \to \hat{R}$ lifts the function $op : O \to V^* \to R$, which defines the semantics of operator calls (CallOp). Given a sequence $\hat{v}^* = [\hat{v}_1, \ldots, \hat{v}_k]$ of symbolic values, $\hat{op}(\hat{v}^*)$ produces a *symbolic result* that evaluates to the concrete result of $op([\![\hat{v}_1]\!]_m, \ldots, [\![\hat{v}_k]\!]_m)$ under every model m. Section 4.2 describes symbolic results and states in detail. For now, it suffices to note that the symbolic result of $\hat{op}$ evaluates to the concrete result of op as expected.

Example 3. To illustrate the factory interface, consider the toy factory in Figure 5. This factory implements the interface for the language $\lambda_{\mathbb{Z}}$ from Example 1 as follows.

First, we define the symbolic boolean, integer, value, and model types for $\lambda_{\mathbb{Z}}$. Our implementation supports only one kind of symbolic variables, symbolic integers $z \in Z$, so models $m \in M_{\mathbb{Z}}$ map symbolic integer variables to integer constants. Symbolic values include symbolic booleans, integers, closures (Figure 4), and ϕ expressions, which select between two symbolic values based on a symbolic boolean guard. We use ϕ expressions to represent conditionals for simplicity; practical factories rely on more efficient representations (see, e.g., [Sen et al. 2015; Torlak and Bodik 2014]).

Next, we implement standard bottom-up interpreters for our base types, followed by a basic implementation of the logical operations provided by the factory. Our logical operators simplify their outputs when given concrete inputs and perform no other optimizations. In contrast, practical implementations use dozens of logical equivalences to simplify their outputs as much as possible.

Finally, we implement lift, truth, merge, cast, and $\hat{op}$. These are straightforward except for merge and $\hat{op}$. For example, $[\![\mathsf{truth}(\phi(\hat{b}, \hat{b}_1, \hat{d}_2))]\!]_m$ is equivalent to $([\![\hat{b}]\!]_m \to [\![\hat{b}_1]\!]_m) \wedge (\neg[\![\hat{b}]\!]_m \to [\![\#\mathsf{t}]\!]_m)$, which says that $\mathsf{truth}(\phi(\hat{b}, \hat{b}_1, \hat{d}_2))$ is false only when $\hat{b}$ is true and $\hat{b}_1$ is false. Similarly, $\mathsf{cast}(\phi(\hat{b}, \hat{c}, \hat{d}))$ produces $[\langle\hat{b}, \hat{c}\rangle]$, which evaluates to $\hat{c}$ when $\phi(\hat{b}, \hat{c}, \hat{d})$ does and has no true guards otherwise. To see why $\mathsf{merge}(G)$ works, recall that it must match the interpretation of G only under models that make exactly one guard in G true. In this case, $\mathsf{merge}([])$ can return anything; $\mathsf{merge}([g])$ must return the sole chosen value; and the nested ϕ value produced by $\mathsf{merge}(g_1 :: g_2 :: G)$ is equivalent to $g_1 :: g_2 :: G$. For example, $\mathsf{merge}([\langle\hat{b}, \hat{v}_1\rangle, \langle\mathsf{not}(\hat{b}), \hat{v}_2\rangle])$ returns $\phi(\hat{b}, \hat{v}_1, \hat{v}_2)$. We show the base cases for $\hat{op}(-, \ldots)$ and omit the rest for brevity. For example, $\hat{op}(-)$

Symbolic booleans $\hat{B}_{\mathbb{Z}}$, integers $\hat{D}_{\mathbb{Z}}$, values $\hat{V}_{\mathbb{Z}}$, and models $M_{\mathbb{Z}}$ for $\lambda_{\mathbb{Z}}$:

$$\begin{array}{llll}
\hat{b} \in \hat{B}_{\mathbb{Z}} ::= b \mid (!\,\hat{b}) \mid (\hat{b}_1 \,\&\&\, \hat{b}_2) \mid (\hat{d}_1 \sim \hat{d}_2) & b \in B \;::= \texttt{\#t} \mid \texttt{\#f} & \sim \;::= \; < \mid = \\
\hat{d} \in \hat{D}_{\mathbb{Z}} ::= d \mid z \mid (-\hat{d}) & z \in Z \;::= \text{symbolic integer variable} & d \in \mathbb{Z} ::= \text{integer constant} \\
\hat{v} \in \hat{V}_{\mathbb{Z}} ::= \hat{b} \mid \hat{d} \mid \hat{c} \mid \phi(\hat{b}, \hat{v}_1, \hat{v}_2) & m \in M_{\mathbb{Z}} ::= Z \to \mathbb{Z} &
\end{array}$$

Interpreters for symbolic booleans $\hat{B}_{\mathbb{Z}}$, integers $\hat{D}_{\mathbb{Z}}$, and values $\hat{V}_{\mathbb{Z}}$, with respect to models $M_{\mathbb{Z}}$:

$$\begin{array}{llll}
\llbracket b \rrbracket_m^{\hat{B}_{\mathbb{Z}}} ::= b & \llbracket (!\,\hat{b}) \rrbracket_m^{\hat{B}_{\mathbb{Z}}} ::= \neg \llbracket \hat{b} \rrbracket_m^{\hat{B}_{\mathbb{Z}}} & \llbracket (\hat{b}_1 \,\&\&\, \hat{b}_2) \rrbracket_m^{\hat{B}_{\mathbb{Z}}} ::= \llbracket \hat{b}_1 \rrbracket_m^{\hat{B}_{\mathbb{Z}}} \wedge \llbracket \hat{b}_2 \rrbracket_m^{\hat{B}_{\mathbb{Z}}} & \llbracket (\hat{d}_1 \sim \hat{d}_2) \rrbracket_m^{\hat{B}_{\mathbb{Z}}} ::= \llbracket \hat{d}_1 \rrbracket_m^{\hat{D}_{\mathbb{Z}}} \sim \llbracket \hat{d}_2 \rrbracket_m^{\hat{D}_{\mathbb{Z}}} \\
\llbracket d \rrbracket_m^{\hat{D}_{\mathbb{Z}}} ::= d & \llbracket z \rrbracket_m^{\hat{D}_{\mathbb{Z}}} ::= m[z] & \llbracket (-\hat{d}) \rrbracket_m^{\hat{D}_{\mathbb{Z}}} ::= -\llbracket d \rrbracket_m^{\hat{D}_{\mathbb{Z}}} & \\
\llbracket \hat{b} \rrbracket_m^{\hat{V}_{\mathbb{Z}}} ::= \llbracket \hat{b} \rrbracket_m^{\hat{B}_{\mathbb{Z}}} & \llbracket \hat{d} \rrbracket_m^{\hat{V}_{\mathbb{Z}}} ::= \llbracket \hat{d} \rrbracket_m^{\hat{D}_{\mathbb{Z}}} & \llbracket \hat{c} \rrbracket_m^{\hat{V}_{\mathbb{Z}}} ::= \llbracket \hat{c} \rrbracket_m^{\hat{C}} & \llbracket \phi(\hat{b}, \hat{v}_1, \hat{v}_2) \rrbracket_m^{\hat{V}_{\mathbb{Z}}} ::= \textbf{if } \llbracket \hat{b} \rrbracket_m^{\hat{B}} \textbf{ then } \llbracket \hat{v}_1 \rrbracket_m^{\hat{V}} \textbf{ else } \llbracket \hat{v}_2 \rrbracket_m^{\hat{V}}
\end{array}$$

Factory operations on symbolic booleans $\hat{B}_{\mathbb{Z}}$:

$$\begin{array}{lllll}
\mathsf{ff} ::= \texttt{\#f} & \mathsf{not}(b) ::= \neg b & \mathsf{and}(\texttt{\#f}, \hat{b}) ::= \texttt{\#f} & \mathsf{and}(\hat{b}, b) ::= \mathsf{and}(b, \hat{b}) & \mathsf{or}(\hat{b}_1, \hat{b}_2) ::= \mathsf{not}(\mathsf{and}(\mathsf{not}(\hat{b}_1), \mathsf{not}(\hat{b}_2))) \\
\mathsf{tt} ::= \texttt{\#t} & \mathsf{not}(\hat{b}) ::= (!\,\hat{b}) & \mathsf{and}(\texttt{\#t}, \hat{b}) ::= \hat{b} & \mathsf{and}(\hat{b}_1, \hat{b}_2) ::= (\hat{b}_1 \,\&\&\, \hat{b}_2) & \mathsf{imp}(\hat{b}_1, \hat{b}_2) ::= \mathsf{not}(\mathsf{and}(\hat{b}_1, \mathsf{not}(\hat{b}_2)))
\end{array}$$

Factory operations on symbolic values $\hat{V}_{\mathbb{Z}}$:

$$\begin{array}{llll}
\mathsf{lift}(i) ::= i & & & \\
\mathsf{truth}(\hat{b}) ::= \hat{b} & \mathsf{truth}(\hat{d}) ::= \texttt{\#t} & \mathsf{truth}(\hat{c}) ::= \texttt{\#t} & \mathsf{truth}(\phi(\hat{b}, \hat{v}_1, \hat{v}_2)) ::= \mathsf{and}(\mathsf{imp}(\hat{b}, \mathsf{truth}(\hat{v}_1)), \mathsf{imp}(\mathsf{not}(\hat{b}), \mathsf{truth}(\hat{v}_2))) \\
\mathsf{merge}([\,]) ::= \texttt{\#f} & \mathsf{merge}([g]) ::= \mathsf{choice}(g) & & \mathsf{merge}(g_1 :: g_2 :: G) ::= \phi(\mathsf{guard}(g_1), \mathsf{choice}(g_1), \mathsf{merge}(g_2 :: G)) \\
\mathsf{cast}(\hat{b}) ::= [\,] & \mathsf{cast}(\hat{d}) ::= [\,] & \mathsf{cast}(\hat{c}) ::= [\langle \texttt{\#t}, \hat{c} \rangle] & \mathsf{cast}(\phi(\hat{b}, \hat{v}_1, \hat{v}_2)) ::= \textbf{append}(\mathsf{subcast}(\hat{b}, \hat{v}_1), \mathsf{subcast}(\mathsf{not}(\hat{b}), \hat{v}_2)) \\
& & & \mathsf{subcast}(\hat{b}, \hat{v}) ::= \textbf{map}(\lambda g. \langle \mathsf{and}(\hat{b}, \mathsf{guard}(g)), \mathsf{choice}(g) \rangle, \mathsf{cast}(\hat{v}))
\end{array}$$

$$\begin{array}{llll}
\hat{op}(-, \hat{b}) ::= \mathsf{Halt}(\langle \texttt{\#t}, \texttt{\#f} \rangle) & \hat{op}(-, \hat{c}) ::= \mathsf{Halt}(\langle \texttt{\#t}, \texttt{\#f} \rangle) & \hat{op}(-, \hat{v}_1, \ldots, \hat{v}_k) ::= \mathsf{Halt}(\langle \texttt{\#t}, \texttt{\#f} \rangle) \text{ where } k \neq 1 & \\
\hat{op}(-, d) ::= \mathsf{Out}(\langle \texttt{\#t}, \texttt{\#t} \rangle, -d) & \hat{op}(-, \hat{d}) ::= \mathsf{Out}(\langle \texttt{\#t}, \texttt{\#t} \rangle, (-\hat{d})) & \hat{op}(-, \phi(\hat{b}, \hat{v}_1, \hat{v}_2)) ::= \ldots & \hat{op}(\ldots) ::= \ldots
\end{array}$$

Fig. 5. A toy factory for the language $\lambda_{\mathbb{Z}}$ from Example 1. The factory reuses the definition of symbolic closures $\hat{c} \in \hat{C}$ and the interpreter $\llbracket \cdot \rrbracket^{\hat{C}}$ from Figure 4.

errors because it is given the wrong the number of arguments, while $\hat{op}(-, \hat{d})$ terminates normally and produces the right value.

4.2 Evaluation rules for $\mathcal{S}_c$

Given a symbolic factory, we define the symbolic semantics $\mathcal{S}_c$ using the rules shown in Figure 6. The rules lift the concrete semantics of λ_c (Figure 3) to work on symbolic values. The judgment $\langle e, \hat{E}, \sigma \rangle \Downarrow \hat{r}$ says that the expression e produces the symbolic result $\hat{r}$ when evaluated in the symbolic environment $\hat{E}$ and state σ. Intuitively, this judgment encodes a set of concrete executions $\langle e, \llbracket \hat{E} \rrbracket_m \rangle \Downarrow \llbracket \hat{r} \rrbracket_m$, one for each model $m \in M$. The environment $\hat{E} \in \hat{\mathbb{E}}$ maps variables to symbolic values. The input state $\sigma \in \Sigma$ consists of two formulas, $\mathsf{assumes}(\sigma)$ and $\mathsf{asserts}(\sigma)$, 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 $\mathsf{Out}(\sigma', \hat{v})$ or $\mathsf{Halt}(\sigma')$. The former means that e may error, abort, or terminate normally to produce $\llbracket \hat{v} \rrbracket_m$, with the outcome determined by the output state σ'. The latter means that e cannot terminate normally under any model; i.e., $\mathsf{normal}_m(\sigma')$ is guaranteed to be false. We describe the rules of $\mathcal{S}_c$ below, and discuss the main features of our design in Section 4.3.

Lifting constants and variables. The rules for evaluating constants (Literal, Closure) and variables (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 symbolic 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 Literal and Closure rules return the result of lifting a given literal and closure, respectively, via $\mathsf{lift}(i)$.

$$\textsc{Literal}\;\frac{v \in B \cup D}{\langle v, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \mathsf{lift}(v))} \qquad \textsc{Closure}\;\frac{}{\langle(\lambda x.e), \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \mathsf{lift}(\langle \hat{\mathbf{cl}}\; x, e, \hat{E}\rangle))}$$

$$\textsc{Variable}\;\frac{x \in \mathsf{dom}(\hat{E})}{\langle x, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{E}[x])} \qquad \textsc{CallOp}\;\frac{\langle x_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_1) \ldots \langle x_n, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_n)}{\langle(o\; x_1 \ldots x_n), \hat{E}, \sigma\rangle \Downarrow \mathsf{strengthen}(\sigma, \hat{op}(o, \hat{v}_1, \ldots, \hat{v}_n))}$$

$$\textsc{Call}\;\frac{\begin{array}{c}\langle x_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_1) \quad \langle x_2, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_2) \quad G_1 = \mathsf{cast}(\hat{v}_1) \quad n = \mathbf{length}(G_1) = \mathbf{length}(G_2)\\ \gamma = \mathsf{some}(\mathsf{guard}, G_1) \quad \sigma' = \mathsf{assert}(\sigma, \gamma) \quad \gamma \neq \mathsf{ff} \wedge \mathsf{assumes}(\sigma') \neq \mathsf{ff} \wedge \mathsf{asserts}(\sigma') \neq \mathsf{ff}\\ \forall i \in [0, n).\; \mathbf{let}\; \langle \hat{b}_1, \langle \hat{\mathbf{cl}}\; x, e, \hat{E}_1\rangle\rangle ::= G_1[i], \langle \hat{b}_2, \hat{r}\rangle ::= G_2[i]\; \mathbf{in}\; (\hat{b}_1 = \hat{b}_2 \wedge \langle e, \hat{E}_1[x \mapsto \hat{v}_2], \mathsf{assume}(\sigma', \hat{b}_1)\rangle \Downarrow \hat{r})\end{array}}{\langle(x_1\; x_2), \hat{E}, \sigma\rangle \Downarrow \mathsf{merge}_{\hat{R}}(\sigma', G_2)}$$

$$\textsc{CallBad}\;\frac{\begin{array}{c}\langle x_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_1) \quad \langle x_2, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_2) \quad G_1 = \mathsf{cast}(\hat{v}_1)\\ \gamma = \mathsf{some}(\mathsf{guard}, G_1) \quad \sigma' = \mathsf{assert}(\sigma, \gamma) \quad \gamma = \mathsf{ff} \vee \mathsf{assumes}(\sigma') = \mathsf{ff} \vee \mathsf{asserts}(\sigma') = \mathsf{ff}\end{array}}{\langle(x_1\; x_2), \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\sigma')}$$

$$\textsc{Let}\;\frac{\langle e_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma', \hat{v}_1) \quad \langle e_2, \hat{E}[x \mapsto \hat{v}_1], \sigma'\rangle \Downarrow \hat{r}}{\langle(\mathbf{let}\;(x\; e_1)\; e_2), \hat{E}, \sigma\rangle \Downarrow \hat{r}} \qquad \textsc{LetHalt}\;\frac{\langle e_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\sigma')}{\langle(\mathbf{let}\;(x\; e_1)\; e_2), \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\sigma')}$$

$$\textsc{IfTrue}\;\frac{\begin{array}{c}\langle x, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v})\\ \mathsf{truth}(\hat{v}) = \mathsf{tt} \quad \langle e_1, \hat{E}, \sigma\rangle \Downarrow \hat{r}\end{array}}{\langle(\mathbf{if}\; x\; e_1\; e_2), \hat{E}, \sigma\rangle \Downarrow \hat{r}} \qquad \textsc{IfFalse}\;\frac{\begin{array}{c}\langle x, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v})\\ \mathsf{truth}(\hat{v}) = \mathsf{ff} \quad \langle e_2, \hat{E}, \sigma\rangle \Downarrow \hat{r}\end{array}}{\langle(\mathbf{if}\; x\; e_1\; e_2), \hat{E}, \sigma\rangle \Downarrow \hat{r}}$$

$$\textsc{IfSym}\;\frac{\begin{array}{c}\langle x, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}) \quad \mathsf{truth}(\hat{v}) \neq \mathsf{tt} \wedge \mathsf{truth}(\hat{v}) \neq \mathsf{ff}\\ \langle e_1, \hat{E}, \mathsf{assume}(\sigma, \mathsf{truth}(\hat{v}))\rangle \Downarrow \hat{r}_1 \quad \langle e_2, \hat{E}, \mathsf{assume}(\sigma, \mathsf{not}(\mathsf{truth}(\hat{v})))\rangle \Downarrow \hat{r}_2\end{array}}{\langle(\mathbf{if}\; x\; e_1\; e_2), \hat{E}, \sigma\rangle \Downarrow \mathsf{merge}_{\hat{R}}(\sigma, [\langle \mathsf{truth}(\hat{v}), \hat{r}_1\rangle, \langle \mathsf{not}(\mathsf{truth}(\hat{v})), \hat{r}_2\rangle])}$$

$$\textsc{Error}\;\frac{}{\langle(\mathbf{error}), \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\mathsf{assert}(\sigma, \mathsf{ff}))} \qquad \textsc{Abort}\;\frac{}{\langle(\mathbf{abort}), \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\mathsf{assume}(\sigma, \mathsf{ff}))}$$

$$\begin{array}{rl}
\mathsf{assume} : \Sigma \to \hat{B} \to \Sigma & \mathsf{assume}(\sigma, \hat{b}) ::= \langle \mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{imp}(\mathsf{asserts}(\sigma), \hat{b})), \mathsf{asserts}(\sigma)\rangle\\
\mathsf{assert} : \Sigma \to \hat{B} \to \Sigma & \mathsf{assert}(\sigma, \hat{b}) ::= \langle \mathsf{assumes}(\sigma), \mathsf{and}(\mathsf{asserts}(\sigma), \mathsf{imp}(\mathsf{assumes}(\sigma), \hat{b}))\rangle\\
\mathsf{merge}_{\hat{R}} : \Sigma \to \mathbb{G}_{\hat{R}} \to \hat{R} & \mathsf{merge}_{\hat{R}}(\sigma, G) ::= \mathbf{if}\; \mathbf{andmap}(\lambda g.\, \mathsf{choice}(g) \in \mathsf{Halt}(\cdot), G)\\
 & \qquad\qquad\qquad \mathbf{then}\; \mathsf{Halt}(\mathsf{merge}_{\Sigma}(\sigma, G))\\
 & \qquad\qquad\qquad \mathbf{else}\; \mathsf{Result}(\mathsf{merge}_{\Sigma}(\sigma, G), \mathsf{merge}_{\hat{V}}(G))\\
\mathsf{merge}_{\Sigma} : \Sigma \to \mathbb{G}_{\hat{R}} \to \Sigma & \mathsf{merge}_{\Sigma}(\sigma, G) ::= \langle \mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{all}(\lambda\langle \hat{b}, \hat{r}\rangle.\, \mathsf{imp}(\hat{b}, \mathsf{assumes}(\mathsf{state}(\hat{r}))), G)),\\
 & \qquad\qquad\qquad \mathsf{and}(\mathsf{asserts}(\sigma), \mathsf{all}(\lambda\langle \hat{b}, \hat{r}\rangle.\, \mathsf{imp}(\hat{b}, \mathsf{asserts}(\mathsf{state}(\hat{r}))), G))\rangle\\
\mathsf{merge}_{\hat{V}} : \mathbb{G}_{\hat{R}} \to \hat{V} & \mathsf{merge}_{\hat{V}}(G) ::= \mathsf{merge}(\mathbf{map}(\lambda\langle \hat{b}, \hat{r}\rangle.\, \langle \hat{b}, \mathsf{value}(\hat{r})\rangle, G))\\
\mathsf{strengthen} : \Sigma \to \hat{R} \to \hat{R} & \mathsf{strengthen}(\sigma, \mathsf{Halt}(\sigma')) ::= \mathsf{Halt}(\mathsf{compose}(\sigma, \sigma'))\\
 & \mathsf{strengthen}(\sigma, \mathsf{Out}(\sigma', \hat{v})) ::= \mathsf{Result}(\mathsf{compose}(\sigma, \sigma'), \hat{v})\\
\mathsf{compose} : \Sigma \to \Sigma \to \Sigma & \mathsf{compose}(\sigma, \sigma') ::= \langle \mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{imp}(\mathsf{asserts}(\sigma), \mathsf{assumes}(\sigma'))),\\
 & \qquad\qquad\qquad \mathsf{and}(\mathsf{asserts}(\sigma), \mathsf{imp}(\mathsf{assumes}(\sigma), \mathsf{asserts}(\sigma')))\rangle\\
\mathsf{some} : (\alpha \to \hat{B}) \to \alpha^* \to \hat{B} & \mathsf{some}(f, A) ::= \mathbf{foldr}(\lambda a.\hat{b}.\mathsf{or}(f(a), \hat{b}), \mathsf{ff}, A)\\
\mathsf{all} : (\alpha \to \hat{B}) \to \alpha^* \to \hat{B} & \mathsf{all}(f, A) ::= \mathbf{foldr}(\lambda a.\hat{b}.\mathsf{and}(f(a), \hat{b}), \mathsf{tt}, A)
\end{array}$$

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

Lifting error and abort expressions. The rules Error and Abort are more interesting. Each produces a result of the form $\mathsf{Halt}(\sigma')$, since evaluating (**error**) or (**abort**) always leads to abnormal termination. The two rules differ in how they compute the output state σ': Error uses $\mathsf{assert}(\sigma, \mathsf{ff})$ and Abort uses $\mathsf{assume}(\sigma, \mathsf{ff})$. 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 $\mathcal{S}_c$ discussed in Section 4.3. For now, we note that $\mathsf{errors}_m(\mathsf{assert}(\sigma, \mathsf{ff}))$ and $\mathsf{aborts}_m(\mathsf{assume}(\sigma, \mathsf{ff}))$ if σ is normal under m, and the two states are equivalent ($\equiv_m$) to σ 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 Error rule (and, by symmetry, the Abort rule), suppose that we have $\langle(\mathbf{error}), \hat{E}, \sigma\rangle \Downarrow \mathsf{Halt}(\sigma')$, where σ' stands for $\mathsf{assert}(\sigma, \mathsf{ff})$. If the state σ is normal under a model m, then we have $\mathsf{errors}_m(\sigma') = [\![\mathsf{assumes}(\sigma')]\!]_m \wedge \neg[\![\mathsf{asserts}(\sigma')]\!]_m = [\![\mathsf{assumes}(\sigma)]\!]_m \wedge \neg[\![\mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{imp}(\mathsf{assumes}(\sigma), \mathsf{ff}))]\!]_m = [\![\mathsf{assumes}(\sigma)]\!]_m \wedge \neg([\![\mathsf{asserts}(\sigma)]\!]_m \wedge ([\![\mathsf{assumes}(\sigma)]\!]_m \rightarrow [\![\mathsf{ff}]\!]_m)) = \texttt{\#t} \wedge \neg(\texttt{\#t} \wedge (\texttt{\#t} \rightarrow \texttt{\#f})) = \texttt{\#t}$. In this case, $\mathsf{Halt}(\sigma')$ evaluates to Err under m, as expected. Similar reasoning shows that if σ is abnormal under m, then $\sigma' \equiv_m \sigma$, so $\mathsf{Halt}(\sigma')$ evaluates to Err or Abt, depending on σ', and the Error rule propagates the existing cause of abnormal termination.

Lifting procedure calls. The rules Call and CallBad lift the semantics of procedure calls $(x_1\ x_2)$ as follows. Given $\langle x_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_1)$ and $\langle x_2, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_2)$, both rules compute $\mathsf{cast}(\hat{v}_1)$ to extract all possible symbolic closures G_1 from $\hat{v}_1$. Then, they use G_1 to construct the formula $\gamma = \mathsf{some}(\mathsf{guard}, G_1)$, which is true when some guard in G_1 is true, and the state $\sigma' = \mathsf{assert}(\sigma, \gamma)$, which asserts that γ holds. If the factory reduces either γ or a component of σ' to ff, we know that $\hat{v}_1$ is not a closure or σ' is not normal under any model. In this case, CallBad triggers and produces $\mathsf{Halt}(\sigma')$. Otherwise, Call establishes that, for every choice $\langle\hat{b}_i, \hat{c}_i\rangle \in G_1$, applying the closure $\hat{c}_i$ to the value $\hat{v}_2$ in the state $\mathsf{assume}(\sigma', \hat{b}_i)$ produces the result $\hat{r}_i$. The guarded results $\langle\hat{b}_i, \hat{r}_i\rangle$ form the list G_2 of all possible outcomes of applying $\hat{v}_1$ to $\hat{v}_2$. Call merges these outcomes into the result $\mathsf{merge}_{\hat{R}}(\sigma', G_2)$, which evaluates to $[\![G_2]\!]_m$ if σ' is normal under m, and to $[\![\mathsf{Halt}(\sigma')]\!]_m$ otherwise. In a nutshell, Call applies the closures from $\mathsf{cast}(\hat{v}_1)$ separately to $\hat{v}_2$ and merges the results, and CallBad short-circuits this process when the factory is able to determine that $\mathsf{cast}(\hat{v}_1)$ 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 $\sigma = \langle\mathsf{tt}, \mathsf{tt}\rangle$, using the toy symbolic factory from Example 3. In this setting, we have $\langle x_1, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_1)$ and $\langle x_2, \hat{E}, \sigma\rangle \Downarrow \mathsf{Out}(\sigma, \hat{v}_2)$, 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 $G_1 = \mathsf{cast}(\hat{v}_1) = []$, $\gamma = \mathsf{some}(\mathsf{guard}, G_1) = \mathsf{ff}$, and $\sigma' = \mathsf{assert}(\sigma, \gamma) = \mathsf{assert}(\sigma, \mathsf{ff}) = \langle\mathsf{tt}, \mathsf{ff}\rangle$. These preconditions trigger CallBad to return $\mathsf{Halt}(\sigma')$, which evaluates to Err under all models.

Second, suppose that $\hat{v}_1 = \phi(\hat{b}, \hat{c}_1, \hat{c}_2)$, where $\hat{b} = (z < 0)$, $\hat{c}_1 = \langle\hat{\mathbf{cl}}\ x, 0, \hat{E}_1\rangle$, and $\hat{c}_2 = \langle\hat{\mathbf{cl}}\ x, 1, \hat{E}_2\rangle$. We now have $G_1 = \mathsf{cast}(\hat{v}_1) = [\langle\hat{b}, \hat{c}_1\rangle, \langle !\ \hat{b}, \hat{c}_2\rangle]$, $\gamma = !((!(!\ \hat{b}))\ \&\&\ (!\ \hat{b}))$, and $\sigma' = \mathsf{assert}(\sigma, \gamma) = \langle\mathsf{tt}, \gamma\rangle$. This triggers the Call rule to establish that $G_2 = [\langle\hat{b}, \mathsf{Out}(\sigma_1, 0)\rangle, \langle !\ \hat{b}, \mathsf{Out}(\sigma_2, 1)\rangle]$, where $\sigma_1 = \mathsf{assume}(\sigma', \hat{b}) = \langle !(\gamma\ \&\&\ !\ \hat{b}), \gamma\rangle$ and $\sigma_2 = \mathsf{assume}(\sigma', !\ \hat{b}) = \langle !(\gamma\ \&\&\ !\ !\ \hat{b}), \gamma\rangle$. Finally, we have $\mathsf{merge}_{\hat{R}}(\sigma', G_2) = \mathsf{Out}(\mathsf{merge}_{\Sigma}(\sigma', G_2), \mathsf{merge}_{\hat{V}}(G_2)) = \mathsf{Out}(\langle\hat{b}_1, \hat{b}_2\rangle, \phi(\hat{b}, 0, 1))$, where $\hat{b}_1 = !(!\ \hat{b}\ \&\&\ !\ !(\gamma\ \&\&\ !\ !\ \hat{b}))\ \&\&\ !(\hat{b}\ \&\&\ !\ !(\gamma\ \&\&\ !\ \hat{b}))$ and $\hat{b}_2 = \gamma\ \&\&(!(!\ \hat{b}\ \&\&\ !\ \gamma)\ \&\&\ !(\hat{b}\ \&\&\ !\ \gamma))$. Applying basic logical simplifications to γ, $\hat{b}_1$, and $\hat{b}_2$, we see that they are all equivalent to tt. So, the result of the call is $\mathsf{Out}(\langle\mathsf{tt}, \mathsf{tt}\rangle, \phi(\hat{b}, 0, 1))$, which matches the interpretation of G_2 under all models.

Lifting let expressions and conditionals. The rules Let, LetHalt, IfTrue, IfFalse, and IfSym are analogous to Call and CallBad. In particular, Let and IfSym provide a general mechanism for lifting let expressions and conditionals, and the remaining rules short-circuit this mechanism in important special cases. LetHalt ensures that the expression $(\mathbf{let}\ (x\ e_1)\ e_2)$ halts for the same reason as e_1 when e_1 is guaranteed to halt under all models. IfTrue and 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, Let and IfSym, behave similarly to Call.

Lifting operator calls. The rule CallOp lifts the semantics of operator calls using the factory function $\hat{op}$ and the auxiliary function strengthen. The function strengthen$(\sigma, \hat{r})$ updates the result $\hat{r}$ of $\hat{op}$ so that it evaluates to $[\![\hat{r}]\!]_m$ when σ is normal under the model m and to $[\![\mathsf{Halt}(\sigma)]\!]_m$ otherwise. In essence, $\hat{op}$ calculates its result assuming an unconstrained start state, i.e., $\langle \mathsf{tt}, \mathsf{tt}\rangle$, and CallOp strengthens this result to reflect the constraints imposed by the input state σ.

Example 6. Suppose that we want to evaluate $(-\ x)$ in the environment $\hat{E} = \{x \mapsto 0\}$ and state σ, using the toy factory from Example 3. In this case, we have $\hat{r} = \hat{op}(-, 0) = \mathsf{Out}(\langle \texttt{\#t}, \texttt{\#t}\rangle, 0)$. Assuming that neither component of σ is constant, we have $\mathsf{strengthen}(\sigma, \hat{r}) = \mathsf{Result}(\mathsf{compose}(\sigma, \langle \texttt{\#t}, \texttt{\#t}\rangle), 0) = \mathsf{Result}(\langle \mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{imp}(\mathsf{asserts}(\sigma), \texttt{\#t})), \mathsf{and}(\mathsf{asserts}(\sigma), \mathsf{imp}(\mathsf{assumes}(\sigma), \texttt{\#t}))\rangle, 0)$, which simplifies to $\mathsf{Out}(\langle \mathsf{assumes}(\sigma), \mathsf{asserts}(\sigma)\rangle, 0) = \mathsf{Out}(\sigma, 0)$. This result matches $[\![\hat{r}]\!]_m$ when σ is normal under m, and it evaluates to Err or Abt otherwise, depending on σ.

4.3 Properties of $\mathcal{S}_c$

The evaluation rules for $\mathcal{S}_c$ 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, assumes(σ) or asserts(σ), 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:* $\forall e, \hat{E}, \sigma, \hat{r}, m.\ (\mathsf{legal}_m(\sigma) \wedge \langle e, \hat{E}, \sigma\rangle \Downarrow \hat{r}) \rightarrow \mathsf{legal}_m(\hat{r})$.

Preserving legality amounts to guaranteeing that each state σ is free of errors when assumes$(\sigma) \rightarrow$ asserts(σ) 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$'(\sigma, \hat{b})$ ::=$\langle \mathsf{and}(\mathsf{assumes}(\sigma), \hat{b}), \mathsf{asserts}(\sigma)\rangle$ 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 $\mathcal{S}_c$ 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 $\sigma = \langle \mathsf{tt}, \mathsf{tt}\rangle$:

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

Given these inputs, $\mathcal{S}_c$ produces a result that is equivalent to $\hat{r} = \mathsf{Out}(\mathsf{assume}(\mathsf{assert}(\sigma, \hat{b}_1), \hat{b}_2), \cdot)$, and the classic semantics produces one equivalent to $\hat{r}' = \mathsf{Out}(\mathsf{assume}'(\mathsf{assert}(\sigma, \hat{b}_1), \hat{b}_2), \cdot)$, where $\hat{b}_1 = \mathsf{truth}(\hat{v}_1)$ and $\hat{b}_2 = \mathsf{truth}(\hat{v}_2)$. After simplifying the resulting states, we get $\mathsf{state}(\hat{r}) = \langle \mathsf{imp}(\hat{b}_1, \hat{b}_2), \hat{b}_1\rangle$ and $\mathsf{state}(\hat{r}') = \langle \hat{b}_2, \hat{b}_1\rangle$. The former is legal under all models, and it is free of errors when $[\![\mathsf{assumes}(\mathsf{state}(\hat{r})) \to \mathsf{asserts}(\mathsf{state}(\hat{r}))]\!]_m = [\![\hat{b}_1]\!]_m$ 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, $\mathcal{S}_c$ 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, $\mathsf{Halt}(\sigma)$, 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:*

$$
\begin{array}{ll}
\mathsf{truth}(\mathsf{lift}_V(\texttt{\#f})) = \mathsf{ff} & \forall b.\ \mathsf{not}(b) = \mathsf{lift}_B(\neg b) \\
\forall v.\ v \neq \texttt{\#f} \to \mathsf{truth}(\mathsf{lift}_V(v)) = \mathsf{tt} & \forall b_1, b_2.\ \mathsf{and}(b_1, b_2) = \mathsf{lift}_B(b_1 \wedge b_2) \\
\forall v.\ \mathsf{merge}([\langle \mathsf{tt}, \mathsf{lift}_V(v)\rangle]) = \mathsf{lift}_V(v) & \forall b_1, b_2.\ \mathsf{or}(b_1, b_2) = \mathsf{lift}_B(b_1 \vee b_2) \\
\forall x, e, E.\ \mathsf{cast}(\mathsf{lift}_V(\langle \mathbf{cl}\ x, e, E\rangle)) = [\langle \mathsf{tt}, \langle \hat{\mathbf{cl}}\ x, e, \mathsf{lift}_{\mathbb{E}}(E)\rangle\rangle] & \forall b_1, b_2.\ \mathsf{imp}(b_1, b_2) = \mathsf{lift}_B(b_1 \to b_2) \\
\forall v.\ v \notin C \to \mathsf{cast}(\mathsf{lift}_V(v)) = [\,] & \\
\forall v_1, \ldots, v_n.\ \hat{op}(o, \mathsf{lift}_V(v_1), \ldots, \mathsf{lift}_V(v_n)) = \mathsf{lift}_R(op(o, v_1, \ldots, v_n)) &
\end{array}
$$

Here, the lifting functions generalize lift *as follows:*

$$
\begin{array}{lll}
\mathsf{lift}_V(b) ::= \mathsf{lift}(b) & \mathsf{lift}_V(d) ::= \mathsf{lift}(d) & \mathsf{lift}_V(\langle \mathbf{cl}\ x, e, E\rangle) ::= \mathsf{lift}(\langle \hat{\mathbf{cl}}\ x, e, \mathsf{lift}_{\mathbb{E}}(E)\rangle) \\
\mathsf{lift}_B(\texttt{\#t}) ::= \mathsf{tt} & \mathsf{lift}_B(\texttt{\#f}) ::= \mathsf{ff} & \mathsf{lift}_{\mathbb{E}}(E) ::= \{x \mapsto \mathsf{lift}_V(E[x]) \mid x \in \mathsf{dom}(E)\} \\
\mathsf{lift}_R(\mathsf{Err}) ::= \mathsf{Halt}(\langle \mathsf{tt}, \mathsf{ff}\rangle) & \mathsf{lift}_R(\mathsf{Abt}) ::= \mathsf{Halt}(\langle \mathsf{ff}, \mathsf{tt}\rangle) & \mathsf{lift}_R(\mathsf{Ans}(v)) ::= \mathsf{Out}(\langle \mathsf{tt}, \mathsf{tt}\rangle, \mathsf{lift}_V(v))
\end{array}
$$

Theorem 3. *If $\mathcal{S}_c$ is parameterized with a reducing symbolic factory, then it reduces to the concrete semantics of λ_c in all lifted concrete environments: $\forall e, E, r.\ \langle e, \mathsf{lift}_{\mathbb{E}}(E), \langle \mathsf{tt}, \mathsf{tt}\rangle\rangle \Downarrow\!\Downarrow \mathsf{lift}_R(r) \leftrightarrow \langle e, E\rangle \Downarrow r$.*

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

```
(let (x₂ (x₁ x₂))
(let (y (λy . (y y)))
  (y y)))
```

From Definition 5, we see that $\mathsf{cast}(\mathsf{lift}_V(\texttt{\#f})) = [\,]$ so $\mathsf{some}(\mathsf{guard}, [\,]) = \mathsf{ff}$, triggering CallBad to return $\mathsf{Halt}(\sigma') = \mathsf{Halt}(\mathsf{assert}(\sigma, \mathsf{ff})) = \mathsf{Halt}(\langle \mathsf{tt}, \mathsf{ff}\rangle)$. This result then triggers LetHalt to return $\mathsf{Halt}(\langle \mathsf{tt}, \mathsf{ff}\rangle) = \mathsf{lift}_R(\mathsf{Err})$ as the output of the program, matching its concrete execution in the environment $E = \{x_1 \mapsto \texttt{\#f}, x_2 \mapsto \texttt{\#f}\}$. 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, Call would return $\mathsf{Out}(\mathsf{assert}(\sigma, \mathsf{ff}), \cdot)$ and trigger Let, leading to an infinite loop.

Determinism. In addition to preserving legality and reducibility, $\mathcal{S}_c$ 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:* $\forall e, \hat{E}, \sigma, \hat{r}_1, \hat{r}_2. \, (\langle e, \hat{E}, \sigma \rangle \Downarrow \hat{r}_1 \wedge \langle e, \hat{E}, \sigma \rangle \Downarrow \hat{r}_2) \rightarrow \hat{r}_1 = \hat{r}_2$.

4.4 Angelic execution and verification with $\mathcal{S}_c$

Given an implementation of $\mathcal{S}_c$ 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 $\mathcal{S}_c$ to evaluate the input program e in the symbolic environment $\hat{E}$ that represents a set of concrete environments $\mathcal{E} = \{E \mid \exists m. \, [\![\hat{E}]\!]_m = E\}$. 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* $\mathsf{solve}(\hat{b})$ *diverges or produces one of two results: either a model $m \in M$ such that $[\![\hat{b}]\!]_m$ is true, or* **unsat** *if no such model exists.*

Definition 7 (Queries). *Let $\hat{E}$ be a symbolic environment that represents a set of concrete states $\mathcal{E} = \{E \mid \exists m. \, [\![\hat{E}]\!]_m = E\}$. We define* $\mathsf{guess}(e, \hat{E})$ *and* $\mathsf{verify}(e, \hat{E})$ *as follows:*

$$\begin{aligned}
\mathsf{guess}(e, \hat{E}) &::= \textbf{let } \langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle \rangle \Downarrow \hat{r}, \sigma = \mathsf{state}(\hat{r}) \textbf{ in } \mathsf{lower}(\hat{E}, \mathsf{solve}(\mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{asserts}(\sigma)))) \\
\mathsf{verify}(e, \hat{E}) &::= \textbf{let } \langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle \rangle \Downarrow \hat{r}, \sigma = \mathsf{state}(\hat{r}) \textbf{ in } \mathsf{lower}(\hat{E}, \mathsf{solve}(\mathsf{and}(\mathsf{assumes}(\sigma), \mathsf{not}(\mathsf{asserts}(\sigma))))) \\
\mathsf{lower}(\hat{E}, m) &::= [\![\hat{E}]\!]_m \\
\mathsf{lower}(\hat{E}, \textbf{unsat}) &::= \textbf{unsat}
\end{aligned}$$

5 CORRECTNESS OF $\mathcal{S}_c$

This section establishes the correctness of $\mathcal{S}_c$ and the queries implemented on top of it. We formulate and prove the soundness and completeness theorem for $\mathcal{S}_c$. Because $\mathcal{S}_c$ may diverge in the presence of loops, this formulation considers all diverging runs to be trivially correct. We therefore prove an additional theorem showing that $\mathcal{S}_c$ is guaranteed to terminate on all loop-free programs: like the classic merging semantics, $\mathcal{S}_c$ defines a total, sound, and complete symbolic evaluation function for this class of programs. To conclude, we establish that our implementations of $\mathsf{guess}(e, \hat{E})$ and $\mathsf{verify}(e, \hat{E})$ satisfy the definitions of angelic execution and verification given in Section 3.

5.1 Soundness and completeness of $\mathcal{S}_c$

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 $\mathcal{S}_c$. 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 $\mathcal{S}_c$ (Theorem 5) by viewing symbolic environments and results as sets of concrete environments and results. Recall from Section 4 that the factory interpretation functions $[\![\cdot]\!]^{\hat{\alpha}}$ use models $m \in M$ to relate symbolic objects $\hat{a} \in \hat{\alpha}$ to their concrete counterparts $a = [\![\hat{a}]\!]^{\hat{\alpha}}_m \in \alpha$. 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 $a \mathrel{\boldsymbol{\in}} \hat{a}$ to denote that $a = [\![\hat{a}]\!]^{\hat{\alpha}}_m$ for some model $m \in M$ (Definition 8). Given the relation $\boldsymbol{\in}$, our correctness theorem for $\mathcal{S}_c$ states the following. Let $\hat{E}$ and $\hat{r}$ be a symbolic environment and result such that $\langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle \rangle \Downarrow\!\!\Downarrow \hat{r}$. 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 $E \mathrel{\boldsymbol{\in}} \hat{E}$, then $r \mathrel{\boldsymbol{\in}} \hat{r}$; and every $r \mathrel{\boldsymbol{\in}} \hat{r}$ can be produced by a concrete run from some environment $E \mathrel{\boldsymbol{\in}} \hat{E}$. In other words, $\hat{r}$ precisely captures the set of all concrete results that are reachable from $\hat{E}$.

Definition 8 (Concrete membership). *Let $[\![\cdot]\!]^{\hat{\alpha}} : M \to \hat{\alpha} \to \alpha$ be an interpretation function provided by a symbolic factory (Figure 4). This interpreter defines the* concrete membership relation *$\boldsymbol{\in}$ from α to $\hat{\alpha}$ as follows: $\forall a, \hat{a}.\ a \mathrel{\boldsymbol{\in}} \hat{a} \leftrightarrow \exists m.\ a = [\![\hat{a}]\!]^{\hat{\alpha}}_m$.*

Theorem 5 (Soundness and Completeness). *The semantics $\mathcal{S}_c$ is* sound *and* complete *with respect to the concrete semantics of λ_c.*

$$\mathcal{S}_c \text{ SOUNDNESS:} \quad \forall e, \hat{E}, \hat{r}.\ \langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle \rangle \Downarrow\!\!\Downarrow \hat{r} \to \forall E.\ E \mathrel{\boldsymbol{\in}} \hat{E} \to \forall r.\ \langle e, E \rangle \Downarrow r \to r \mathrel{\boldsymbol{\in}} \hat{r}$$

$$\mathcal{S}_c \text{ COMPLETENESS:} \quad \forall e, \hat{E}, \hat{r}.\ \langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle \rangle \Downarrow\!\!\Downarrow \hat{r} \to \forall r.\ r \mathrel{\boldsymbol{\in}} \hat{r} \to \exists E.\ \langle e, E \rangle \Downarrow r \land E \mathrel{\boldsymbol{\in}} \hat{E}$$

As noted earlier, our definition of correctness is partial, and this formulation is necessary to account for programs with loops. But solver-aided tools often target *finite* programs, which are free of loops and procedure calls. For this class of programs, $\mathcal{S}_c$ provides the same strong guarantee as symbolic execution and the classic merging semantics: evaluation always terminates (Theorem 6) with a sound and complete result (Theorem 5).

Theorem 6 (Termination on finite programs). *Let e be a program that contains no procedure calls, and let $\hat{E}$ be a symbolic environment that binds every free variable in e. Then, there exists a symbolic result $\hat{r}$ such that $\langle e, \hat{E}, \langle \mathsf{tt}, \mathsf{tt} \rangle \rangle \Downarrow\!\!\Downarrow \hat{r}$.*

5.2 Correctness of queries based on $\mathcal{S}_c$

Building on the correctness of $\mathcal{S}_c$, 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 $\mathsf{guess}(e, \hat{E})$ produces a correct output whenever it terminates. If the output of $\mathsf{guess}(e, \hat{E})$ is an environment E, then e executes normally in $E \mathrel{\boldsymbol{\in}} \hat{E}$. But if the output is **unsat**, then there is no environment $E \mathrel{\boldsymbol{\in}} \hat{E}$ in which e executes normally. Theorem 8 is symmetric.

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

$$\textit{SAT:} \quad \forall e, \hat{E}, E.\ \mathsf{guess}(e, \hat{E}) = E \to (E \mathrel{\boldsymbol{\in}} \hat{E} \land \mathsf{normal}(e, E))$$

$$\textit{UNSAT:} \quad \forall e, \hat{E}.\ \mathsf{guess}(e, \hat{E}) = \textbf{unsat} \to \forall E.\ (E \mathrel{\boldsymbol{\in}} \hat{E} \to \neg\mathsf{normal}(e, E))$$

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

$$\textit{SAT:} \quad \forall e, \hat{E}, E.\ \mathsf{verify}(e, \hat{E}) = E \longrightarrow (E \in \hat{E} \wedge \mathsf{errors}(e, E))$$

$$\textit{UNSAT:} \quad \forall e, \hat{E}.\ \mathsf{verify}(e, \hat{E}) = \textbf{unsat} \longrightarrow \forall E.\ (E \in \hat{E} \longrightarrow \neg\mathsf{errors}(e, E))$$

6 IMPLEMENTING $\mathcal{S}_c$: 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 $\mathcal{S}_c$. 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 $\mathcal{S}_c$, 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 $\mathcal{S}_c$ in Lean

The reference evaluator, which we call Leanette, is implemented as a generic symbolic interpreter for λ_c. Like $\mathcal{S}_c$, 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, Leanette$(n, e, \hat{E}, \sigma)$ produces either a symbolic result $\hat{r}$ such that $\langle e, \hat{E}, \sigma\rangle \Downarrow \hat{r}$, or **none**, to indicate that the evaluation is stuck. This implementation is both sound and complete with respect to our symbolic semantics (Theorem 9): $\mathcal{S}_c$ admits every result produced by Leanette, and, given enough fuel, Leanette can produce every result admitted by $\mathcal{S}_c$.

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

$$\textit{Soundness:} \quad \forall n, e, \hat{E}, \sigma, \hat{r}.\ \textit{Leanette}(n, e, \hat{E}, \sigma) = \hat{r} \longrightarrow \langle e, \hat{E}, \sigma\rangle \Downarrow \hat{r}$$

$$\textit{Completeness:} \quad \forall e, \hat{E}, \sigma, \hat{r}.\ \langle e, \hat{E}, \sigma\rangle \Downarrow \hat{r} \longrightarrow \exists n.\ \textit{Leanette}(n, e, \hat{E}, \sigma) = \hat{r}$$

We instantiate 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 ϕ values to represent the output of conditional evaluation. In our implementation, a symbolic union is a list of guarded values, $[\langle \hat{b}_1, \hat{v}_1\rangle, \ldots, \langle \hat{b}_k, \hat{v}_k\rangle] \in \mathbb{G}_{\hat{V}}$, that are themselves not unions (i.e., $\hat{v}_i \notin \mathbb{G}_{\hat{V}}$). 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 Leanette is a sound, complete, and reducible symbolic evaluator for λ_c with integers and lists.

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

6.2 Rosette*: an optimized implementation of $\mathcal{S}_c$ in Racket

The optimized evaluator, Rosette*, implements $\mathcal{S}_c$ for the entire Rosette language [Torlak and Bodik 2014], which is a superset of λ_c. The Rosette language extends Racket [Felleisen et al. 2018;

Flatt and PLT 2010] with constructs for creating symbolic values, emitting assertions, and formulating solver-aided queries. Rosette's existing reusable evaluator, which we call Rosette, 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. Rosette* replaces the core evaluation rules of Rosette with those of $\mathcal{S}_c$.

Rosette* also makes two changes to the Rosette interface, to reflect the switch to $\mathcal{S}_c$. First, it extends the Rosette language to include assumptions, (**assume** e), which act as syntactic sugar for the (**abort**) expression in λ_c. This change makes assumptions a first-class construct in Rosette*; they can appear anywhere in a program. In contrast, Rosette has limited support for assumptions via `#: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, Rosette* 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 Rosette, this interface exposes concepts such as the assertion stack and the path condition, which are part of the classic merging semantics. In Rosette*, this interface exposes concepts such as the symbolic state with assumptions and assertions, as defined by $\mathcal{S}_c$.

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* against Leanette with solver-aided differential testing

To gain confidence that Rosette* 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* 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 $\text{Rosette}^*(e, \hat{E}, \langle \text{tt}, \text{tt} \rangle)$ and $\text{Leanette}(n, e, \hat{E}, \langle \text{tt}, \text{tt} \rangle)$, 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., $\text{Rosette}^*(e, \hat{E}, \langle \text{tt}, \text{tt} \rangle) = \hat{r}_R$, $\text{Leanette}(n, e, \hat{E}, \langle \text{tt}, \text{tt} \rangle) = \hat{r}_L$, and $\forall m. [\![\hat{r}_R]\!]^{\hat{R}}_m = [\![\hat{r}_L]\!]^{\hat{R}}_m$. To implement the equivalence check, the oracle imports $\hat{r}_L$ into Rosette*, and uses Z3 [De Moura and Bjørner 2008] to solve the query (**verify** (**assert** (**equal?** $[\![\hat{r}_R]\!]^{\hat{R}}$ $[\![\hat{r}_L]\!]^{\hat{R}}$))). This query searches for a model m, if any, under which the two symbolic results evaluate to different concrete results according to **equal?**. The function **equal?** considers its inputs to be equivalent if they have the same representation; e.g., two lists are **equal?** if they have the same length and **equal?** elements, and two closures are **equal?** if they have **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 λ_c expressions with their symbolic environments. Each generated test is a valid Rosette program that consists of a sequence of (**define-symbolic** x T) expressions, followed by an expression e from the λ_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 η between the set of

```
(define-symbolic n integer?) ; Symbolic integer

(if (zero? (* 0 n)) ; If 0 * n = 0
    #t              ; then true
    (let loop ()    ; else loop infinitely
      (loop)))
```

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

```
(define-symbolic n integer?) ; Symbolic integer

(if (zero? (* 0 n)) ; If 0 * n = 0
    (assert #f)     ; then error
    #t)             ; else true
```

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

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

natural numbers and the set of all possible test programs. The bijection η is set up so that larger numbers tend to correspond to larger programs. Next, the generator uses η 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* and Leanette behave differently on 7a and equivalently on 7b.

When applied to 7a, Rosette* produces $\text{Out}(\langle \text{tt}, \text{tt}\rangle, \text{tt})$, 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* is able to perform this reduction, so it terminates with the expected value. Both of these outcomes are correct according to formalization (Section 5).

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

Test results. We validate Rosette* 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* 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*. 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*, 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* 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

Bucket (by AST size)			Leanette state			Rosette* state			Leanette time		Rosette* time		
Range	Count	Avg	Sym	Max	Avg	Sym	Max	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	224	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	7,523	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	✓

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* 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.

Injected mistake (β)	Location	Caught?
Make $\text{and}(b_a, b_b) = b_a$ where b_a and b_b are non-literal symbolic booleans	symbolic factory	✓
Make $\text{or}(b, b) = \text{tt}$ where b is a non-literal symbolic boolean	symbolic factory	✓
Make $\hat{op}(<, z_a, z_b) = (z_b < z_a)$ where either z_a or z_b is a non-literal symbolic integer	symbolic factory	✓
In CallOp, skip strengthen	symbolic evaluator	✓
In Call, change the guards for evaluation of the body to be tt	symbolic evaluator	✓
In CallBad, use the input symbolic state without asserting γ	symbolic evaluator	✓
In Let, do not bind any variable	symbolic evaluator	✓
In IfSym, swap $\text{not}(\text{truth}(\hat{v}))$ and $\text{truth}(\hat{v})$ when calling $\text{merge}_{\hat{R}}(\cdot, \cdot)$	symbolic evaluator	✓
In IfTrue and IfFalse, swap the conditions	symbolic evaluator	✓
In Err, do Abort	symbolic evaluator	✓
In Abort, do Err	symbolic evaluator	✓

Table 2. Coverage results of the randomly generated programs. Each of the injected faults β violates soundness. "✓" indicates that the injected fault is caught by the generated test set T (i.e., Rosette* and BadLeanette(β) disagree on at least one program in T).

"state" columns show the number of non-trivial symbolic states produced by the evaluator, as well as the maximum and average size of the non-trivial states in each bucket. A state is trivial iff 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 Leanette and Rosette* represent symbolic booleans as abstract syntax graphs. These graphs are trees in Leanette and DAGs in Rosette*.

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

Coverage. To gain further confidence in our test results, we inject 11 faults into Leanette, one at a time, and check that they are uncovered by at least one test in T. For each fault β, we create a new evaluator BadLeanette(β) that applies β to Leanette, and we test Rosette* against

BadLeanette(β) 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 $\mathcal{S}_c$ rules. Our test suite discovers all of them, giving us confidence that Rosette* correctly implements $\mathcal{S}_c$.

7 UTILITY AND PERFORMANCE OF AN $\mathcal{S}_c$ EVALUATOR: EXPERIMENTS

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

(1) Can the interface provided by an $\mathcal{S}_c$ evaluator simplify the implementation of client tools?
(2) Can an $\mathcal{S}_c$ 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* and Rosette

We evaluate the utility of $\mathcal{S}_c$ by porting two sets of benchmarks to Rosette*, 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 (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.

As we saw in Section 6.2, Rosette* extends the Rosette language to include **assume** expressions, and it exposes different constructs for symbolic reflection. To port our benchmarks to Rosette*, 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* 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. 2015], or CVC4 v1.8 [Barrett et al. 2011].

As the data in Table 3 indicates, porting the SymPro benchmarks to Rosette* 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 [Kazerounian 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, 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*.

7.1.2 Porting Jitterbug. Jitterbug is the most complex code base we ported to Rosette* (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

Benchmark	Rosette LoC	Rosette* LoC	LoC diff		Solver
Bagpipe [Weitz et al. 2016]	2,643	2,643	+2	-2	Z3
Bonsai [Chandra and Bodik 2018]	437	437	+1	-1	Boolector
Cosette [Chu et al. 2017]	774	774	+0	-0	Z3
Ferrite [Bornholt et al. 2016]	348	348	+2	-2	Z3
Fluidics [Willsey et al. 2019]	98	98	+0	-0	Z3
GreenThumb [Phothilimthana et al. 2016]	3,554	3,556	+13	-11	Boolector
IFCL [Torlak and Bodik 2014]	483	483	+13	-13	Boolector
MemSynth [Bornholt and Torlak 2017]	13,303	13,307	+8	-4	Z3
Neutrons [Pernsteiner et al. 2016]	37,174	37,174	+2	-2	Z3
Nonograms [Butler et al. 2017]	3,368	3,374	+11	-5	Z3
Quivela [Amazon Web Services 2018]	1,010	1,009	+10	-11	Z3
RTR [Kazerounian et al. 2018]	1,649	1,640	+12	-21	CVC4
SynthCL [Torlak and Bodik 2014]	2,257	2,256	+42	-43	Boolector
Wallingford [Borning 2016]	2,532	2,533	+12	-11	Z3
WebSynth [Torlak and Bodik 2014]	1,181	1,189	+14	-6	Z3
Jitterbug [Nelson et al. 2020]	29,280	28,935	+478	-823	Boolector

Table 3. Differences between the implementations of the ported and original benchmarks.

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 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 $pre \wedge \neg post$, is sound (c.f., Example 7).

When porting Jitterbug to Rosette*, 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* 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* 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* and Rosette

To evaluate the performance of $\mathcal{S}_c$, we compare the running time and encoding size of Rosette* and Rosette 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* and Rosette. 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.

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

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

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

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

Benchmark	Total			Eval.			Solving			Terms		
	best	worst	avg.	best	worst	avg.	best	worst	avg.	best	worst	avg.
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

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

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.

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* and Rosette 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× slower and up to 3.8× faster than the original ones, with an average speedup of 13%.

The analysis in Table 4c shows that, on average, Rosette* generates 6–9% more terms than Rosette, 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 $\mathcal{S}_c$ updates the assumption component of the symbolic state using assume$(\sigma, \hat{b})$, while the classic merging semantics uses assume$'(\sigma, \hat{b})$. In the worst case, the symbolic factory for Rosette* and Rosette creates 3 terms to represent assumes(assume$(\sigma, \hat{b})$) and only 1 term to represent assumes(assume$'(\sigma, \hat{b})$). Additionally, Rosette uses a simpler function for merging two symbolic states, which generates no new terms, while the $\mathcal{S}_c$ function merge$_\Sigma(\sigma, G)$ can generate up to $4|G| + 2$ terms. These two differences are the main reason why Rosette* emits up to 8× more terms than Rosette in our benchmarks. We avoid this blowup in the average case by specializing Rosette* with rewriting rules tailored for the terms generated by assume$(\sigma, \hat{b})$ and merge$_\Sigma(\sigma, G)$.

Given that Rosette* emits more terms than Rosette, it may seem surprising that it is, on average, slightly faster than Rosette, and that the resulting formula is slightly easier to solve. 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* operates on a more constrained symbolic state than Rosette. Both of its state components carry terms that are not available in Rosette and that may trigger additional simplifications. Second, thanks to legality, Rosette* emits the query assumes$(\sigma) \land \neg$asserts(σ) (Definition 7), while Rosette emits $\neg$asserts(σ). It is sound for Rosette* to emit just $\neg$asserts(σ), 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* produces a larger encoding than Rosette, our results show that it matches the runtime performance of Rosette 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 $\mathcal{S}_c$, 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, $\mathcal{S}_c$ admits a wide range is implementations. We use Lean to prove that $\mathcal{S}_c$ is sound and complete with respect to the concrete semantics of its target language, λ_c, which extends Core Scheme with assumptions and assertions. We also prove that $\mathcal{S}_c$ preserves two properties, legality and reducibility, that are key to reusing a symbolic evaluator in a client tool. Leanette and Rosette*, two implementations of $\mathcal{S}_c$ in Lean and Racket, respectively, show that $\mathcal{S}_c$ provides a general contract for building and validating reusable evaluators. By porting 16 published verification and synthesis tools from Rosette to Rosette*, we demonstrate that $\mathcal{S}_c$ provides a practical interface for client tools: Rosette* both simplifies the implementation of two of the benchmarks and matches the performance of Rosette across these tools. All source code accompanying this paper will be publicly available.

REFERENCES

Amazon Web Services. 2018. Quivela. https://github.com/jamesbornholt/quivela

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*. Snowbird, UT, 171–177.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking Without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Amsterdam, The Netherlands, 193–207.

Jasmin Christian Blanchette, Mathias Fleury, and Christoph Weidenbach. 2017. A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*. Melbourne, Australia, 4786–4790.

Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with Angelic Nondeterminism. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Madrid, Spain, 339–352.

James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, USA, 83–98.

James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models from Framework Sketches and Litmus Tests. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Barcelona, Spain, 467–481.

James Bornholt and Emina Torlak. 2018. Finding Code That Explodes Under Symbolic Evaluation. In *Proceedings of the 2018 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Boston, MA, Article 149, 26 pages.

Alan Borning. 2016. Wallingford: Toward a Constraint Reactive Programming Language. In *Proceedings of the Constrained and Reactive Objects Workshop (CROW)*. Málaga, Spain, 45–49.

Eric Butler, Emina Torlak, and Zoran Popović. 2017. Synthesizing Interpretable Strategies for Solving Puzzle Games. In *Proceedings of the 12th International Conference on the Foundations of Digital Games (FDG)*. Hyannis, MA, Article 10, 17 pages.

Kartik Chandra and Rastislav Bodik. 2018. Bonsai: Synthesis-Based Reasoning for Type Systems. *Proc. ACM Program. Lang.* 2, POPL (Jan. 2018), 62:1–62:34.

Adam Chlipala. 2010. A Verified Compiler for an Impure Functional Language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Madrid, Spain, 93–106.

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems (CIDR)*. Chaminade, CA.

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Barcelona, Spain, 168–176.

Edmund Clarke, Daniel Kroening, and Karen Yorav. 2003. *Behavioral Consistency of C and Verilog Programs*. Technical Report CMU-CS-03-126. Carnegie Mellon University.

Lori A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering* 2, 3 (1976), 215–222.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Los Angeles, CA, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. San Antonio, TX, 269–282.

Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. 2017. Efficient Certified RAT Verification. In *Proceedings of the 26th International Conference on Automated Deduction (CADE)*. Gothenburg, Sweden, 220–236.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Budapest, Hungary, 337–340.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover. In *Proceedings of the 25th International Conference on Automated Deduction (CADE)*. Berlin, Germany, 378–388.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (March 2018), 62–71.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the 14th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Albuquerque, NM, 237–247.

Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Design Inc.

Matt Fleming. 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, United Kingdom, 927–942.

Jacques-Henri Jourdan. 2016. *Verasco: a Formally Verified C Static Analyzer*. Theses. Universite Paris Diderot-Paris VII. https://hal.archives-ouvertes.fr/tel-01327023

Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Mumbai, India, 247–259.

Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. 2018. Refinement Types for Ruby. In *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Los Angeles, CA, USA, 269–290.

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

Dorel Lucanu, Vlad Rusu, and Andrei Arusoaie. 2017. A generic framework for symbolic execution: A coinductive approach. *Journal of Symbolic Computation* 80 (May–June 2017), 125–163.

William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Huntsville, Ontario, Canada, 225–242.

Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Banff, Alberta, Canada, 41–61.

Max S. New, Burke Fetscher, Robert Bruce Findler, and Jay McCarthy. 2017. Fair enumeration combinators. *Journal of Functional Programming* 27 (2017), e19. https://doi.org/10.1017/S0956796817000107

Aina Niemetz, Mathias Preiner, and Armin Biere. 2015. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)* 9 (2015), 53–58.

Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. 2016. Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*, Vol. 2. Toronto, ON, Canada, 23–41.

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling Up Superoptimization. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, USA, 297–310.

Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: multi-path symbolic execution using value summaries. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy, 842–853.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. San Jose, CA, USA, 404–415.

Sol Otis Swords. 2010. *A verified framework for symbolic execution in the ACL2 theorem prover.* Ph.D. Dissertation. The University of Texas at Austin.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM Symposium on New Ideas in Programming and Reflections on Software (Onward!)*. Indianapolis, IN, USA, 135–152.

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, 530–541.

Richard Uhler and Nirav Dave. 2014. Smten with Satisfiability-Based Search. In *Proceedings of the 29th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Portland, OR, USA, 157–176.

Konstantin Weitz, Steven Lyubomirsky, Stefan Heule, Emina Torlak, Michael D. Ernst, and Zachary Tatlock. 2017. Space-Search: A Library for Building and Verifying Solver-Aided Tools. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Oxford, United Kingdom, Article 25, 28 pages.

Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *Proceedings of the 31st ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Amsterdam, The Netherlands, 765–780.

Max Willsey, Ashley P. Stephenson, Chris Takahashi, Pranav Vaid, Bichlien H. Nguyen, Michal Piszczek, Christine Betts, Sharon Newman, Sarang Joshi, Karin Strauss, and Luis Ceze. 2019. Puddle: A Dynamic, Error-Correcting, Full-Stack Microfluidics Platform. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Providence, RI, 183–197.

Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, United Kingdom, 718–730.