Grisette: Symbolic Compilation as a Functional Programming Library

SIRUI LU, University of Washington, USA
RASTISLAV BODÍK, Google Research, USA

The development of constraint solvers simplified automated reasoning about programs and shifted the engineering burden to implementing symbolic compilation tools that translate programs into efficiently solvable constraints. We describe Grisette, a reusable symbolic evaluation framework for implementing domain-specific symbolic compilers. Grisette evaluates all execution paths and merges their states into a normal form that avoids making guards mutually exclusive. This ordered-guards representation reduces the constraint size 5-fold and the solving time more than 2-fold. Grisette is designed entirely as a library, which sidesteps the complications of lifting the host language into the symbolic domain. Grisette is purely functional, enabling memoization of symbolic compilation as well as monadic integration with host libraries. Grisette is statically typed, which allows catching programming errors at compile time rather than delaying their detection to the constraint solver. We implemented Grisette in Haskell and evaluated it on benchmarks that stress both the symbolic evaluation and constraint solving.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; • **Software and its engineering** → **Functional languages**; **Formal methods**.

Additional Key Words and Phrases: Symbolic Compilation, State Merging

ACM Reference Format:
Sirui Lu and Rastislav Bodík. 2023. Grisette: Symbolic Compilation as a Functional Programming Library. *Proc. ACM Program. Lang.* 7, POPL, Article 16 (January 2023), 33 pages. https://doi.org/10.1145/3571209

1 INTRODUCTION

The development of constraint solvers enabled a variety of advanced reasoning about programs [De Moura and Bjørner 2011; Dennis et al. 2006; Dolby et al. 2007; Jha et al. 2010]. Central to this effort have been tools that translate the semantics of a program into constraints, including analyzers [Xie and Aiken 2005], symbolic execution engines [Cadar et al. 2008a; Havelund and Pressburger 2000; Sen et al. 2013], and symbolic compilers [Clarke et al. 2004; Solar-Lezama 2008; Solar-Lezama et al. 2006]. More recently, symbolic host languages allowed the implementation of DSLs that were automatically translated to constraints, easing the engineering burden of implementing a symbolic DSL compiler by reusing the host language symbolic engines [Porncharoenwase et al. 2022; Torlak and Bodik 2014]. These host languages were also proved useful for general-purpose languages, as they hosted interpreters for LLVM [Lattner and Adve 2004] and RISC-V [Asanović and Patterson 2014], enabling symbolic analysis of programs that compile to these targets [Nelson et al. 2019].

The effectiveness of symbolic host languages [Porncharoenwase et al. 2022; Solar-Lezama 2008; Torlak and Bodik 2013, 2014] rests on the combination of three techniques:

Authors' addresses: Sirui Lu, Paul G. Allen School, University of Washington, USA, siruilu@uw.edu; Rastislav Bodík, Brain Team, Google Research, USA, rastislavb@google.com.

2475-1421/2023/1-ART16
https://doi.org/10.1145/3571209

- *Partial evaluation* is used to evaluate away language constructs that the symbolic compiler is not designed to translate to constraints, such as virtual function calls or complex data types. This allows programmers to use advanced constructs as long as they are used on concrete (i.e., non-symbolic) values [Sen et al. 2015; Torlak and Bodik 2014].
- *All-path symbolic execution* increases the efficiency by simultaneously translating multiple program paths (or all paths up to a certain execution depth, as in bounded model checking) [Nelson et al. 2019]. All-path symbolic execution is particularly important in program synthesis, where the space of candidate programs is represented as a single program whose branches correspond to alternative rules in the grammar of the candidate space [Solar-Lezama 2008; Solar-Lezama et al. 2006]. By translating all program paths to a single constraint system, we can ask the solver for a solution that represents the synthesized program.
- *State merging* is an indispensable mitigation to the path explosion problem caused by all-path symbolic execution [Baldoni et al. 2018; Godefroid 2007]. The specific challenge is how to merge program states so that partial evaluation remains effective. Rosette [Torlak and Bodik 2013, 2014] and MultiSE [Sen et al. 2015] developed algorithms that reduce path explosion while preserving concrete values in the program state; as a result, partial evaluation remains as effective as if each path was evaluated individually.

This paper describes a symbolic compiler that satisfies all three properties *and* can be implemented as a library. In contrast, existing systems that provide all three techniques are based on lifting a language into the symbolic domain, typically by developing a custom compiler or modifying an interpreter. Because it is challenging to translate arbitrary language constructs to constraints, these systems either restrict the host language (CBMC [Clarke et al. 2004] and Sketch [Solar-Lezama 2008; Solar-Lezama et al. 2006]) or lift a subset of the host language (Rosette 3 [Torlak and Bodik 2013, 2014] and Rosette 4 [Porncharoenwase et al. 2022]). Lifting a host language can lead to usability challenges and confusion, for example, because lifted semantics can subtly change the original semantics [Torlak 2016]. Additionally, porting the designs to other languages is nontrivial because they are influenced by the metaprogramming features available in the host language.

Compared to existing systems, our design, Grisette[1], seeks to offer these benefits:

- *Symbolic compiler as a library.* As a library, Grisette does not change the semantics of the language. Instead, the symbolic value is a plain data structure with typed combinators provided to build DSLs that operate on symbolic values. This combinator design serves as an implementation recipe that is language-independent and thus, hopefully, more portable. In languages with type classes and monads, all benefits presented in this paper can be obtained, including the construction of a symbolic compiler for a stateful language via the `StateT` transformer (Section 4.2). In other languages, one can still reimplement Grisette's combinators, while adapting the interface with the languages' native features. This paper presents our design in Haskell, but we have also developed a prototype in Scala following the same architecture.
- *Purely functional design.* Grisette's symbolic value is a purely functional data structure, which simplifies parallelization and memoization of symbolic compilation. In our experiments with Bonsai [Chandra and Bodik 2017] ported to Grisette and a backtracking regex synthesizer, memoization accelerated symbolic compilation by 1.6x–7.9x, and improved overall performance by 1.2x–7.5x (see Section 5.5).
- *Static types.* Static types ensure that external libraries use symbolic values correctly by preventing symbolic values from flowing to where concrete values are expected. This guarantee is challenging to achieve in dynamically typed host languages, causing hard-to-diagnose

[1] https://github.com/lsrcz/grisette

unexpected behaviors [Torlak 2022]. Additionally, Grisette users can extend the system with symbolic variants of user-defined or library data types with type classes. We used type classes to support some common data types, such as tuples, byte strings, and size-tagged vectors. Another benefit provided by a static type system is performance debugging, which boils down to ensuring that the program propagates concrete values to enable partial evaluation (symbolic values may lead to path explosion, slower compilation, and solving) [Bornholt and Torlak 2018]. By rejecting programs with symbolic values in concrete parameters, the type system helps debug and tune symbolic evaluators. In Section 5.2, we show that type checking helped identify performance bugs missed by a dynamic symbolic profiler.

Formulating a library with all properties and benefits requires us to solve the following challenges:

- *Efficient state merging algorithm.* Designing a merging algorithm that is computationally efficient and generates small formulas is nontrivial because slower algorithms can afford to perform more formula optimizations. The challenge is even harder in the functional setting, where the error state is propagated with the normal return values rather than stored in a separate global store. In Grisette, we designed a new representation, called Ordered Guards (ORG), as well as a merging algorithm that runs in $O(n)$ time as opposed to the naive $O(n^2)$ time, and generates much smaller formulas than the previous algorithms. These formulas are also easier to solve. In Section 5.3, we show that on average our algorithm generates about 91% smaller formulas than Rosette on the benchmarks and has about 10.1x overall speedup. This algorithm also gives us assertion optimization nearly for free.
- *Efficient handling of multiple error types in a functional setting.* Rosette 3 generates highly compact formulas thanks to an optimization we call *assertion optimization* (Section 2.2). It relies on two assumptions: (1) assertions are the only kind of program exit (assumptions are not supported), and (2) there exists a global state for tracking assertions. As Rosette 4 extended its support to assertions and assumptions, it gave up part of the assertion optimization even when only assertions were present in the code, generating formulas larger than Rosette 3. Our system generalizes assertions and assumptions to arbitrary error types, using functional programming infrastructures such as `Either`. Importantly, we preserve the Rosette 3 optimization whenever only one error type is present. This requires us to carefully design the structure of the functional symbolic state (Section 3.5).

The rest of the paper is organized as follows: we first overview our algorithm and the user-friendly, highly configurable interface of Grisette in Section 2. We then describe the algorithm in detail in Section 3 and discuss the interface design in Section 4. In Section 5, we evaluate Grisette on verification and synthesis tools ported from Rosette [Porncharoenwase et al. 2022] and G2Q [Hallahan et al. 2019a] as benchmarks. Finally, we discuss some related work in Section 6.

2 OVERVIEW

This section outlines the key contributions of Grisette. We first describe the Ordered Guards (ORG) representation of symbolic values. Next, we show how ORG facilitates purely functional, yet efficient, propagation of the symbolic evaluator state, including assertions and assumptions. Finally, we show how symbolic evaluation with ORG is exposed as a library for verification and synthesis.

2.1 Symbolic Union with the Ordered Guards Representation

Traditional symbolic evaluation executes program paths separately [King 1976]. The resulting path explosion can be alleviated by merging the symbolic states of the program branches into a single symbolic state [Clarke et al. 2004]. Alternatively, states can be partially merged using a *symbolic union*, which is a set of symbolic states guarded by path conditions [Sen et al. 2015; Torlak and

Bodik 2014]. Unions are particularly useful for preserving and evaluating away partially concrete values.

Unions are typically represented as a set of values guarded by mutually exclusive path conditions [Sen et al. 2015; Torlak and Bodik 2014]. We call this particular union representation *Mutually Exclusive Guards* (MEG). The middle column of Fig. 1 symbolically evaluates a program using MEG unions. We start by knowing the symbolic value of `x`, which is a union with three members: `x` is `[a]` when `cond1` is true; `[b,c]` when ¬`cond1` ∧ `cond2` is true; and similarly for the third member. Next, we evaluate the term `y = head x` on the symbolic value of `x`. The term `head x` is evaluated three times: on `[a]`, `[b,c]`, and `[a,c,d]`, once for each member of the union. Since each member was a concrete list (albeit with symbolic elements), the evaluation of `head` was done entirely concretely.

Merging of symbolic values happens when MEG unions are normalized. In Fig. 1, we conceptually decompose normalization into two steps: grouping and merging. The former identifies groups of mergeable values, while the latter replaces each group with a single value and merges the guards in the group.[2] The overhead of normalizing an MEG union is in ensuring that guards are mutually exclusive. This grows some guards, underlined in Fig. 1. In turn, larger conditions can become barriers to term optimizations. For example, the symbolic evaluator may not be able to afford to determine that the path condition for `a` can be simplified from `cond1` ∨ (¬`cond1` ∧ ¬`cond2`) to `cond1`∨¬`cond2`, as such reasoning generally requires expensive invocations of SMT solvers. Although Hansen et al. [2009] proposed to simplify path conditions using heuristic DNF minimization tools (e.g., Espresso [Rudell 1986]), they show that many conditions cannot be optimized. Domain knowledge is needed to further simplify the results, which is not easily available in a general-purpose symbolic evaluator. Therefore, it is desirable to grow terms less during normalization.

```
-- the program under symbolic execution
x = if cond1 then [a] else if cond2 then [b,c] else [a,c,d]
y = head x
```

Variable	Mutually exclusive guards (MEG)	Ordered guards (ORG)

x (merged)	`[a]` if `cond1`
	`[b,c]` if ¬`cond1` ∧ `cond2`
	`[a,c,d]` if ¬`cond1` ∧ ¬`cond2`
		`[a]` if `cond1`
		`[b,c]` else if `cond2`
		`[a,c,d]` otherwise

y (symex)	`a` if `cond1`
	`b` if ¬`cond1` ∧ `cond2`
	`a` if ¬`cond1` ∧ ¬`cond2`
		`a` if `cond1`
		`b` else if `cond2`
		`a` otherwise

y (norm.1)	`a` if `cond1`
	`a` if ¬`cond1` ∧ ¬`cond2`
	`b` if ¬`cond1` ∧ `cond2`
		`a` if `cond1`
		`a` else if ¬`cond2`
		`b` otherwise

y (norm.2)	`a` if `cond1`∨(¬`cond1` ∧ ¬`cond2`) `b` if ¬`cond1` ∧ `cond2`	`a` if `cond1`∨¬`cond2` `b` otherwise

Fig. 1. Two union representations: mutually exclusive guards (MEG) versus ordered guards (ORG). We underline subterms added to maintain the invariants of the respective representations. Same subterms only increase the overall size for 1 and will only be underlined once. The execution for `x` merges three paths, and the program is internally translated to `mrgIf cond1 [a] (mrgIf cond2 [b,c] [a,c,d])`. The execution for `y` is split into three steps: per-case symbolic execution (symex), identification of mergeable values (norm.1), which performs reordering in the ORG representation, and final merging of those values (norm.2).

[2]In this section we restrict ourselves to merging identical values. Different values can also be merged. For example, in Fig. 1, if `a` and `b` are in solvable types (directly representable as SMT formulas), the result will be further merged to `(ite (cond1∨(¬cond1∧¬cond2)) a b)` using `ite` operator. Refer to Section 3 for details on the merging of such values.

This paper proposes a union representation that does not normalize guards into a mutually exclusive form. Instead, the values are ordered in an if-elseif-...else structure, and a value will be chosen only if all the previous guards hold. We call the representation ORG, for *Ordered Guards*. For example, for the value of `x` shown in the third column of Fig. 1, the guards `cond1` and `cond2` are allowed to hold simultaneously, and their ordering determines which alternative is selected.

Unfortunately, the ORG representation cannot entirely avoid growing guards. Although it does not make guards mutually exclusive, it transforms guards when it reorders the alternative values, which is needed to completely merge mergeable values. Reordering happens during the grouping phase of normalization, and the transformation of guards involves the creation of new nodes. For example, in row `y` (norm. 1) in Fig. 1, the `cond2` guard grew to ¬`cond2`. In general, the further a value needs to move during reordering, the more its guard needs to grow.

In summary, MEG grows guards to make them mutually exclusive, while ORG grows guards when it reorders alternative values. Our approach to making ORG efficient is to reduce the need to reorder values by keeping them sorted in the same way in all unions. This allows us to merge two ORG unions using a mergesort-style merge procedure without having to perform any reordering. This merging then produces an ORG union that is normalized and sorted. It can be done in linear time and usually generates compact conditions.

The ORG union and the merging process are illustrated in Fig. 2. Fig. 2a shows a valid ORG union; Fig. 2b shows an invalid union where two mergeable values (two 1's) have not been merged. Fig. 2c shows another invalid example, where the values are not sorted. Since merging is always performed at conditional join points using the `mrgIf` function of the symbolic evaluator, we merge ORG unions in this function. Fig. 2d shows the mergesort-like merging of two ORG unions.

$$\begin{cases} 1 & \text{if} & \text{cond1} \\ 2 & \text{else if} & \text{cond2} \\ 3 & \text{otherwise} \end{cases} \qquad \begin{cases} 1 & \text{if} & \text{cond1} \\ 1 & \text{else if} & \text{cond2} \\ 3 & \text{otherwise} \end{cases} \qquad \begin{cases} 1 & \text{if} & \text{cond1} \\ 3 & \text{else if} & \text{cond2} \\ 2 & \text{otherwise} \end{cases}$$

(a) Valid (b) Invalid, contains mergeable values (c) Invalid, not sorted

$$\texttt{mrgIf}\left[\texttt{c}, \begin{cases} 1 & \text{if} & \texttt{c1} \\ 2 & \text{else if} & \texttt{c2} \\ 4 & \text{otherwise} \end{cases}, \begin{cases} 1 & \text{if} & \texttt{c3} \\ 3 & \text{else if} & \texttt{c4} \\ 4 & \text{otherwise} \end{cases}\right] = \begin{cases} 1 & \text{if} & \text{ite}(\texttt{c}, \texttt{c1}, \texttt{c3}) \\ 2 & \text{else if} & \texttt{c} \wedge \texttt{c2} \\ 3 & \text{else if} & \neg\texttt{c} \wedge \texttt{c4} \\ 4 & \text{otherwise} \end{cases}$$

(d) Mergesort-style merge

Fig. 2. ORG union and the merge procedure

Grisette is not the first symbolic evaluator to represent unions in an ordered manner. Some existing symbolic execution tools have represented their union-like values with nested `ite` operators, effectively using ordered semantics underlying our ORG representation. However, these tools did not attempt to normalize unions by merging all identical values. For example, VERITESTING [Avgerinos et al. 2014], and JAVA RANGER [Sharma et al. 2020] only perform generic optimizations to the nested `ite` structure, such as when one branch is unreachable. The tool by Sinha [2008] employs a set of rewriting rules to merge some mergeable values, but the rules may fail to merge some pairs of values. For example, when evaluating `If a 3 x` where `x` is `If b 1 (If c 2 3)`, the rewriting rules will not identify the two 3's, leaving the result as `If a 3 (If b 1 (If c 2 3))`. In this example, the optimization rules fail because they only search for mergeable terms that are in adjacent `ite` nodes. We achieve complete merging by defining a normal form, which by definition merges all values, as well as the algorithms that can merge two unions while preserving the normalized form.

ORG also admits further optimizations. So far, we have described ORG as a flat list of values. When the list is long, it is beneficial to partition it and merge (some or all) partitions into nested ORG values. This hierarchical ORG representation is particularly useful for complex data types, such as tuples, which otherwise tend to yield long lists. In the following example, v_i are values, while t_i are ORG containers; t_1 and t_1' are merged into a nested ORG by a recursive call to `mrgIf`. We describe the hierarchical merging algorithm for nested ORG containers in Section 3.2 and Section 3.3. The key idea is to create the hierarchical ORG value with a *merging strategy* (s in the example), which is a function that informally partitions the list and orders the partitions. The function also generates sub-strategies or merging functions to control how to merge the ORG trees and values in the next level. In the example, the sub-strategy s' controls the merging of the sub-trees t_1 and t_1', and the merging function f is used to merge the mergeable values v_4 and v_4'.

$$\texttt{mrgIf}\left[s, c, \begin{cases} t_1 & \text{if} \\ v_3 & \text{else if} \\ v_4 & \text{otherwise} \end{cases}^{c_1}_{c_2}, \begin{cases} t_1' & \text{if} \\ t_2' & \text{else if} \\ v_4' & \text{otherwise} \end{cases}^{c_3}_{c_4}\right] = \begin{cases} \texttt{mrgIf}(s', c, t_1, t_1') & \text{if} & \text{ite}(c, c_1, c_3) \\ t_2' & \text{else if} & c \wedge c_2 \\ v_3 & \text{else if} & \neg c \wedge c_4 \\ f(c, v_4, v_4') & \text{otherwise} & \end{cases}$$

This nested representation generates smaller formulas because values in a sub-tree share the sub-tree guard; it can also be merged faster. Our empirical study shows that our system generates 91.2% smaller formulas than Rosette 4 [Porncharoenwase et al. 2022] and 79.2% smaller formulas than our system implemented with the MEG representation (see Section 5). These improvements stem from both the ordered nature of our representation and its hierarchical structure.

2.2 Propagating Purely Functional Symbolic Evaluator States

Symbolic evaluation tools are used to reason about programs, and we need to specify the desired behaviors, usually with assumptions and assertions. The verification condition (VC) generation procedure [King 1969] then translates the behaviors into constraints. This subsection analyzes several existing VC optimizations and shows that our merging algorithm can achieve these optimizations without dedicated algorithms — it suffices to suitably order the values in the ORG representation.

Rosette 3 (a legacy version of Rosette) [Torlak and Bodik 2014] separates the assertion conditions from non-error program states (Rosette 3 does not support assumptions). The following Rosette code is written in a Haskell-like syntax extended with sequential imperative constructs.

```
assert c; x = if c1 then (assert c2; a) else (assert c3; b)
-- x: ite(c1, a, b) for solvable types, or {a if c1, b if not c1} for unsolvable types.
-- assertion store: c∧(¬c1∨c2)∧(c1∨c3).
```

To evaluate the program, Rosette 3 splits paths under `c1` and evaluates each branch separately. On both paths, the path condition pc is carried as a *symbolic evaluator state*. When an `assert` c statement is executed, condition $\neg pc \vee c$ (i.e. $pc \Rightarrow c$) would be added to a *global* assertion store, which is another symbolic evaluator state. The condition indicates that the program will not violate the current assertion if the current path is not executed *or* the asserted condition is true. All conditions will be connected using conjunctions to ensure that no assertion will be violated. The algorithm has two good properties. The first is that `assert` statements would not impact subsequent execution because `x` will not depend on `c2` or `c3`. The second is that the clauses produced by different `assert` statements do not depend on each other, thus the assertions produce non-interfering clauses compactly in the CNF formula. We refer to these properties as *assertion optimizations*.

With more advanced VC generation, assumptions can be used to limit the search space, and if some assumption is violated, the verification would simply pass. Rosette 4 received assumption support with its recent update [Porncharoenwase et al. 2022]. Because assertions and assumptions interact, it may be necessary to duplicate some conditions when they are interleaved. However,

Rosette 4 does not preserve assertion optimization even if the program is assumption-free, and this issue has already been discussed by the authors. As a simple example, the following shows the constraint generated by Rosette 4 for the previous Rosette 3 code, which is much more complex.

```
c∧(¬c1∨(c∧(c2∨(¬(¬c∨c1)))))∧(c1∨(c∧(c3∨(¬(¬c∨¬c1)))))
```

Bounded model checking tools use different algorithms and usually generate better formulas in all-path scenarios. To compile a program to constraints, CBMC [Clarke et al. 2004] transforms the program into SSA form. The following code shows an example, and the second line is the SSA form.

```
assume c1; x = if c2 then (assert c3; a) else (assume c4; b); assert (g x)
assume c1; assert (c2⇒c3); x1 = a; assume (¬c2⇒c4); x2 = b; x = ite c2 x1 x2; assert (g x)
```

Unlike concrete executions, here the `if` statement executes both branches, and the results are merged with `ite`. Path conditions are added to all assertions and assumptions; for example, `assert c3` would become `assert (c2 ⇒ c3)`. The program is now flattened, and the statements can be processed one by one. For each `assert` statement, the tool collects all previous assumptions as preconditions. The assumptions are then eliminated, and the assertions will be independent statements.

```
assert(c1⇒(c2⇒c3)); assert((c1∧(¬c2⇒c4))⇒g(ite(c2, a, b)))
```

It is easy to see that this encoding would have the assertion optimization properties when only assertions are present, but it would duplicate the assumptions, e.g. `c1` in the above code.

Based on our observations of previous work and our needs, we have the following design goals.

(1) We want a purely functional VC generation so that we can benefit from functional programming techniques, e.g., memoization, and functional handling of errors.
(2) We want to extend VC generation to support richer error models in a simple and unified way, including assertions, assumptions, and user-defined errors, without dedicated algorithms.
(3) We want the errors to be propagated as efficiently as if they were kept in a global state with a specialized algorithm. We should preserve assertion optimization when only assertions are present, and generate compact formulas when different errors interleave.

Our approach models assertions and assumptions as abnormal terminations. Instead of a centralized verification condition store, we use the basic error handling technique in functional programming to track verification conditions. We use `Either` to model executions that can terminate with an error, and optimizations can be achieved with an appropriate hierarchical ordering for merging.

Execution of sequential programs can be achieved by tree substitution on the `Right` values in ORG containers. Unlike previous tools, our system does not need to hardcode the semantics of `assert` and `assume` or build them into the core semantics with a dedicated optimization algorithm. They can be defined by the user as simple data types, and we can implement the Rosette 3 example:

```
data Error = Assert | Assume
assume c = if ¬c then Left Assume else Right ()
assert c = if ¬c then Left Assert else Right ()
assert c; x = if c1 then (assert c2; Right a) else (assert c3; Right b)
```

With tree substitution semantics, the code is translated into the following `mrgIf` representation.

```
mrgIf (¬c) (Left Assert) (mrgIf c1 (mrgIf (¬c2) (Left Assert) (Right a))
                                   (mrgIf (¬c3) (Left Assert) (Right b)))
```

It will be normalized to the compact representation with assertion optimization. If we rewrite `ite` with conjunctions and disjunctions, we will get the same (negated) assertion store as Rosette 3.

```
If (¬c∨ite(c1, ¬c2, ¬c3)) (Left Assert) (Right (ite(c1, a, b)))
```

With configurable merging rules, the VC generation behavior can be configured when there are assumptions. One simple way is to order `Left Assert` before `Left Assume`. Then we can get the following result for the example program we used for CBMC, in which the guard for `Left Assert` is a simplified and negated version of the encoding by CBMC.

```
If (c1∧((c2∨¬c3)∧((c2∨c4)∧¬g(ite(c2, a, b))))) (Left Assert)
   (If (¬(c1∧(c2∨c4))) (Left Assume) (Right ()))
```

Another way to generate VCs is to use the tree structure of `If`, and normalize the container to the following form. This can be achieved with our hierarchical merging algorithm and will be our default VC generation procedure. We will discuss this in Section 3.5.

```
If anyErrorHappens (If theErrorIsAssert (Left Assert) (Left Assume)) variousRightValues
```

Our empirical study shows that the two encodings may work well in different scenarios and that neither is superior. Since our system is configurable, we only show two possible efficient ways to handle the VC, and the user may handle the VC in a more efficient way with domain knowledge.

Another important advantage of our system is that it is not limited to the common assertion/assumption model and supports arbitrary error types. The user will not have to encode correctness criteria as assertions and assumptions. They can define their errors with domain knowledge and program with familiar FP constructs to handle errors. The decoupling of the assertion and assumption constructs from the program is useful for building more versatile tools, as the errors do not have predefined semantics and can be interpreted in different ways in different scenarios. The user will also be able to do things difficult before. For example, in previous systems, when an assertion is made, there is no way to transform it into an assumption or handle it with error handlers. However, it would be easy to do so in our system by substituting the `Left` values.

2.3 Symbolic Compilation as a Library

Grisette is a tool to build other solver-aided tools, and usability is one of the most important design goals. Grisette provides a configurable versatile symbolic evaluator, which can be combined with domain-specific semantics to build high-quality solver-aided tools for various tasks. Previous versatile symbolic evaluation systems lifted a subset of the host language to the symbolic domain (e.g., Rosette) or built a new language with symbolic features (e.g. Sketch [Solar-Lezama et al. 2006]). For example, Rosette lifts a subset of Racket, allowing *some* syntax and library functions to work on symbolic values. Our tool Grisette is not a lifted language. Instead, it is a statically-typed monadic library and integrates nicely with the existing host language ecosystems. We believe that this can make the tool more accessible. In this section, we will demonstrate several advantages of it.

2.3.1 Integration with the Host Language Ecosystem through Static Types: Safety and Efficiency. It is desirable to create symbolic versions of host language data structures, such as hash tables that contain concrete keys and symbolic values. In dynamically typed systems such as Rosette and MultiSE, such data structures are not automatically available, although a dynamically typed hash table should in principle accept any type of value, including symbolic values. This can result in unmerged values and unexpected behavior in Rosette:

```
(if a (make-hash (list (cons 1 b))) (make-hash (list (cons 1 c))))
; (union [a #hash((1 . b))] [(! a) #hash((1 . c))]) better: #hash((1 . (ite a b c)))
(equal? (make-hash (list (cons 1 b))) (make-hash (list (cons 1 c)))) ; #f expected: (= b c)
```

In the first line, the system does not know how to merge the two hash tables and would simply keep them concrete in two MEG branches. In the second line, the system lacks knowledge on how to compare the hash tables symbolically. To compare the two hash tables, it falls back to Racket's default `equal?`, which will find that `b` and `c` are different symbolic formulas and return false.

In Grisette, with the type class mechanism, we can allow restricted usage for hash tables. For example, when keys are concrete and values are simply mergeable (any two values of the type can be merged, see Section 3.2), we can implement the symbolic equality relation correctly and let the system merge the hash tables with the same key set. When the user tries to use unsupported operations, no surprising behavior would occur, and the program simply does not type check.

```
instance ConcreteType K => SEq (HashMap K V) where ...
instance (ConcreteType K, SimpleMergeable V) => Mergeable (HashMap K V) where ...
```

Being able to configure functionalities with types helps us integrate Grisette closely with host language libraries, and we implemented `bytestring` [Stewart and Coutts 2021] and `vector-sized` (size-tagged vectors) [Hermaszewski 2021] support without modifying Grisette itself.

Using types to configure state merging is also useful for performance tuning and avoiding *missed concretization* bugs [Bornholt and Torlak 2018; Kuznetsov et al. 2012]. For example, we can have two state representations with similar semantics: ORG container of concrete integers, e.g. `If a 1 (If b 2 3)`, or symbolic integer represented as SMT formulas, e.g. `(ite a 1 (ite b 2 3))`. We can use them as indices to get elements from a list. With the first representation, we can distribute among the three concrete branches and return no infeasible value. However, with the second representation, the system has to compare it with all possible indices `0, 1, ...` as there is no general algorithm to efficiently extract all possible concrete values from an SMT formula. This generates larger results with infeasible values where `(ite a 1 (ite b 2 3))` equals 0. In Grisette, the two representations have the types `UnionM Integer` and `SymInteger`, respectively. The types constrain the desired representation, thus constraining that no performance problem due to undesired representations can occur. We use this feature to implement a backtracking regex synthesizer efficiently, as we need to backtrack and restart matching from concrete indices, and in Section 5.2, we will show that with static types, we successfully found and fixed more performance bugs that cannot be easily found with symbolic profiling tools [Bornholt and Torlak 2018].

2.3.2 Monadic Design with Reduced Implementation Work and Increased Generality. The ORG symbolic union, `Union`, is at the heart of Grisette. Its definition is as straightforward as an if-then-else tree. The `SymBool` here is a symbolic Boolean value for path conditions, which will be elaborated in Section 3.1. Similar to tree monads, `Union` is a monad with tree substitutions, and sequential programs can be modeled with sequential applications of binds or the `do` syntax sugar.

```
data Union a = Single a | If SymBool (Union a) (Union a)
```

Due to the constrained monad problem [Sculthorpe et al. 2013], the bind operation for `Union` cannot perform state merging and normalization. We introduce another monad `UnionM`, which caches the information required for merging, to reduce the boilerplate for manually merging the results. The following shows the Grisette code modeling the program shown in Section 2.1. Here, `mrgReturn` is a wrapper for `return`, but also propagates the merging strategy provided by the `Mergeable` instance to merge the result of the entire `do` block. This is further propagated when the result is used, which helps to keep all intermediate values normalized and avoid path explosions.

```
do x <- mrgIf cond1 (return [a]) (mrgIf cond2 (return [b,c]) (return [a,c,d]))
   mrgReturn (head x)
```

Using monads allows modeling various mechanisms with monad transformers. The error handling mechanism can be supported without boilerplate with `ExceptT`. Stateful programming can also be adapted to Grisette, using `StateT` to manage global states. Our approach is more flexible and general than previous systems. For example, in previous systems, it is hard to symbolically evaluate programs with unstructured control flows [Torlak 2021], while in Grisette, this can be supported easily: we support delimited continuations with `reset/shift` [Danvy and Filinski 1990] through `ContT`, and we used it in our *backtracking* regex synthesizer to implement coroutines. We can even memoize the coroutine computations, which will be evaluated in Section 5.5. Our system is also compatible with *some* effect systems, including the effect system with the fusion techniques [Wu and Schrijvers 2015]. We have implemented support for `fused-effects` [Wu et al. 2022]. This allows the mechanisms to be implemented more flexibly. Other effect systems are inherently unfriendly to

state merging (not just for our system), as they deeply embed intermediate results [Kiselyov et al. 2013]. We can only merge the computation, which cannot reduce as many paths as merging values.

Several monadic or effect stacks have been proposed for various mechanisms in symbolic execution [Darais et al. 2017; Mensing et al. 2019; Wei et al. 2020]. With the configurable nature of Grisette, it is possible to adapt them to use our system to enhance them with state merging.

3 ORG: A REPRESENTATION OF SYMBOLIC VALUES WITH ORDERED GUARDS

In this section, we will give our representation of symbolic values, including the ORG-based union data structure. We will introduce the representation invariant (hierarchical sortedness) and merging algorithm for unions progressively for more advanced types in the union. We will then introduce how errors can be handled easily and flexibly with the union data structure.

3.1 Symbolic Values Representation

We begin our design with the representation of symbolic values. In our system, we distinguish between solvable types and unsolvable types. A solvable type can be represented as a symbolic formula and is directly supported by the underlying SMT solvers. This includes symbolic Booleans, symbolic integers (unbounded mathematical integers), etc. For example, symbolic Booleans has the type `SymBool` in our system. A `SymBool` can be a concrete Boolean value, e.g. `true`, or a symbolic constant (placeholders for concrete constants), e.g. `a`, or a complex symbolic formula with symbolic operations, e.g., `(&& a b)`. To better fit Haskell's ecosystem, we have extended the operations, and these extended operations can be lowered to SMT operations. The solvers can then be used to determine the concrete values for the symbolic constants to satisfy the formula.

For unsolvable types from the host language, e.g., lists (`[SymBool]`), or concrete integers (`Integer`), we introduce `Union` to capture the ORG semantics. Here `a` is a type variable for the contained type, and we can use a container with the type `Union [SymBool]` for the lists of symbolic Booleans.

```
data Union a = Single a | If SymBool (Union a) (Union a)
```

In the following example, the `Union` value has the same semantics as the ORG representation. We will sometimes omit the `Single` constructors for readability.

```
If     cond1  [a]
 (If   cond2  [b,c]
              [a,c,d])
```

versus

```
[a]     if      cond1
[b,c]   elseif  cond2
[a,c,d] otherwise
```

Note that the left branch in our `If` definition is a `Union a` rather than just an `a`, allowing us to have sub-trees there, and allowing for nested ORG representations. In the following example, v_* are values, and t_* are sub-trees. When c_1 is true, the result will be further determined by c_{11} and c_{12}.

$$\begin{cases} t_1 & \text{if} & c_1 \\ v_2 & \text{elseif} & c_2 \\ v_3 & \text{otherwise} \end{cases} \quad \text{where} \quad t_1 = \begin{cases} v_{11} & \text{if} & c_{11} \\ v_{12} & \text{elseif} & c_{12} \\ v_{13} & \text{otherwise} \end{cases}$$

In the next subsection, we will focus on our representation invariant which states that the values and sub-trees should be sorted hierarchically. As the construction for the conditions will be trivial and will be shown in Section 3.3, we will only focus on the values in the invariants in Section 3.2. We introduce a more compact representation for the ORG containers when the conditions are not relevant: the previous example would be represented as $[t_1, v_2, v_3]$, where t_1 would be $[v_{11}, v_{12}, v_{13}]$. For the container, we will refer to the set $\{v_{11}, v_{12}, v_{13}, v_2, v_3\}$ as the set of the leaf values.

3.2 Representation Invariant for Union

For complete merging with ORG semantics, we describe a novel algorithm with `Union` containers. Our goal is to create a modular abstraction working efficiently for both solvable and unsolvable

types, particularly algebraic data types. Our algorithm normalizes the containers to maintain a representation invariant such that the values are sorted hierarchically, so that we can merge them easily by hierarchically performing a mergesort-style merge. The intuition for the invariant will be demonstrated in several sub-subsections, progressively from simple types to complex types. Then in Section 3.3, we will formalize our algorithm and prove some properties of it.

As an overview, Table 1 shows typical representations of `Union`. The solvable values will always be merged into a single value, and concrete primitives will be maintained in sorted non-nested ORG containers. Then we will use product types to show the intuition of a general framework for all kinds of types, which maintains the values in a hierarchically sorted nested ORG container.

Table 1. Typical representation for a `Union` with invariant

Category	Typical type	Typical representation with invariant
Solvable type	`SymBool`	v
Concrete prim	`Int`	$[v_1, v_2, \cdots, v_n]$ where $v_1 < v_2 < \cdots < v_n$
Product type	`(Int, Int, SymBool)`	$[t_1, t_2, \cdots, t_n]$ where $t_i = [v_{i1}, v_{i2}, \cdots, v_{im}]$,such that (1) all values in t_i has the same first tuple element f_i and $f_1 < f_2 < \cdots < f_n$, (2) in each t_i, let s_{ij} be the second element of v_{ij}, then $s_{i1} < s_{i2} < \cdots < s_{im}$
Sum type	`Either SymBool Int`	$[$`Left` $b, t_{right}]$ where $t_{right} = [$`Right` $v_1,$ `Right` $v_2, \cdots,$ `Right` $v_n]$ such that $v_1 < v_2 < \cdots < v_n$

3.2.1 Merging Solvables in Union. As a strongly typed system, we will not automatically unwrap the values with solvable types in a `Union` and merge them directly with SMT `ite` operator. However, merging them in a `Union` is simple, as we can always merge them into a single value with `ite`.

```
mrgIf a (Single b) (Single c) ==> Single (ite a b c)
```

This can be generalized to any type `a` that can be merged with a *merging function* `m :: SymBool -> a -> a -> a`, which can be defined by the user for other data types.

3.2.2 Merging Concrete Integers. Our system supports both symbolic and concrete integers (the integer type in the host language), and they have different types. This distinction is useful for controlling concretization and allowing for more optimizations based on use cases [Bornholt and Torlak 2018]. For example, in a machine code verifier [Nelson et al. 2019], we may want the program counter to be concrete to prune infeasible paths. The user can then use `SymInteger` for symbolic integers, or use `Union Integer` to represent a set of concrete integers under some path conditions.

The merging for `SymInteger` is trivial with `ite`, but the only way to merge concrete integers is to find and merge identical numbers. A naive approach does not scale if it has to make $O(n^2)$ comparisons for a program with n possible execution paths. Sorting the integers before looking for identical ones is a better approach, which reduces the time complexity to the sorting complexity $O(n \log n)$. However, sorting values in ORG containers involves expensive reorderings of values and is not efficient in term size. Our algorithm, instead, maintains the invariant that all `Union` operands for a conditional join are already sorted, so the merging and invariant maintaining can be done easily with a mergesort-style merge if state merging is performed at every conditional join.

Definition 3.1 (Sorted Invariant). A union value u of type `Union a` meets *Sorted Invariant* if there exists a total order R on a, and u is organized as $[v_1, v_2, \cdots, v_n]$ where $v_1 < v_2 < \cdots < v_n$.

We can then merge sorted `Union`s with a mergesort-style merge.

```
mrgIf a (If c1 1 (If c2 2 4)) -- Single constructor omitted
        (If c3 2 (If c4 3 4))
==> If (&& a c1) 1 (If (ite a c2 c3) 2 (If (&& (not a) c4) 3 4))
```

3.2.3 Handling Product Types. After discussing the merging procedures for simple types, we now introduce our merging framework by analyzing the merging of product types.

Let's analyze a product type (`Integer, Integer, SymBool`) with both concrete and solvable fields. One simple generalization of our previous analysis on solvable and concrete primitive types is that we can sort the values lexicographically with the concrete fields, and for those values with the same concrete fields, we can then merge their solvable fields with some merging function. See the following code as an example. The branches are non-nested ORG containers, and they are already sorted by the first and second fields. In the result, as $(1, 1, *)$ exists in both branches, they would be merged together by using the merging function `ite` to merge the `SymBool` fields.

```
mrgIf c (If c1 (1,1,u) (If c2 (1,2,v) (If c3 (2,3,w) (2,4,x)))) -- Single constructor omitted
        (If c4 (1,1,y) (9,1,z))
==> If         (ite c c1 c4)     (1,1,(ite c u y))       -- new term here twice
     (If       (&& c c2)         (1,2,v)                 -- new term here
      (If      (&& c c3)         (2,3,w)                 -- new term here
       (If     c                 (2,4,x)
                                 (9,1,z))))
```

However, this approach still needs improvement for two reasons: (1) inefficient symbolic evaluation for very long lists, and (2) generation of larger and redundant terms. For the inefficiency aspect, to merge the two linked lists, when there are only a few pairs of mergeable values, we may need to skip many values one-by-one, e.g., `Single (2,3,w)` and `Single (2,4,x)`. This would be inefficient when the `Union`s are large. For the redundant terms aspect, although we skipped the values, we need to augment the path conditions for them, which increases the overall term size.

Our approach hierarchically merges values using the tree structure of `Union`, rather than only the degenerate linked list structure. It first sorts the values with the first fields and then gathers all the values with the same first field in some sub-trees. For each sub-tree, the values will then be sorted using the second fields, and then the values can be merged by merging the symbolic fields. The advantage of this hierarchical encoding is that the lists would be shorter at each level, and we do not have to go into sub-trees when unnecessary. In the following example, we can skip the whole sub-tree for $(2, *, *)$ in one step and only augment the path condition once for the whole tree. Such performance differences will be significant when the sizes of the unions are large.

```
mrgIf c (If c1 (If c2 (1,1,u) (1,2,v))
               (If c3 (2,3,w) (2,4,x))) -- no need to look into
        (If c4 (1,1,y) (9,1,z))
==> If       (ite c c1 c4)     (If (|| c2 (not c)) (1,1,ite c u y) (1,2,v)) -- (1,*,*)
     (If     c                 (If c3 (2,3,w) (2,4,x))                       -- (2,*,*)
                               (9,1,z))                                      -- (9,*,*)
```

Our algorithm will be established with several definitions. We will first establish the invariant to maintain by the merging algorithm, and the algorithm would be formalized later. First, we will define the concept of merging strategies, which describes how a set of values can be merged.

Definition 3.2 (Merging strategy). A *merging strategy* *ms* `:: MergingStrategy` *a* is either

(1) a simple strategy with a merging function *mf* `:: SymBool -> a -> a -> a`, or
(2) a sorted strategy with a totally ordered index type *i*, an indexing function *idx* `:: a -> i`, and a sub-strategy function *sub* `:: i -> MergingStrategy a`.

```
data MergingStrategy a where
  SimpleStrategy :: (SymBool -> a -> a -> a) -> MergingStrategy a
  SortedStrategy :: (Ord idx) => (a -> idx) -> (idx -> MergingStrategy a) -> MergingStrategy a
```

The sorted strategy partitions values into subsets with the indexing function, and orders the subsets with indices. For each subset, we have the sub-strategy for its index. After several partitioning, some sets will have simple strategies, and the values can then be merged with the merging function. In an ORG container merged with some sorted strategy, each sub-tree corresponds to a subset for some index. And the sub-trees are further merged with the sub-strategies for their indices.

For `(Integer, Integer, SymBool)`, the merging strategy can be defined as follows. At the top level, the index is the first field, and at the second level, we sort the tuples with the second field. Then we merge the tuples with the same concrete fields by merging the symbolic fields with `ite`.

```
SortedStrategy (\(x,_,_) -> x) (\_ -> SortedStrategy (\(_,x,_) -> x) (\_ ->
  SimpleStrategy (\c (a,b,t) (_,_,f) -> (a,b,ite c t f))))
```

Note that the functions in a merging strategy can be partial and only defined on some subsets of the values of type a. For a merging strategy to properly perform partitioning and merging, we need some laws. We can easily verify that the merging invariant defined before meets these laws.

Definition 3.3 (Proper merging strategy). A merging strategy ms for the type a is a *proper merging strategy* regarding a set $S \subseteq a$ if it have a *finite depth* and meets the *closed law* or the *partition law* on the set S. We use the predicate $\mathrm{PM}(ms, S)$ for ms to be a proper merging strategy regarding S.

Being proper on some set means that it can be used to merge values in that set. The closed law and partition law ensure that we can partition the values using sorted strategies with *partially defined* indexing and sub-strategy functions, and if the values are *eventually* partitioned to small enough sets with simple strategies, the values can then be safely merged with the merging function. The finite depth ensures that partitioning can be done in finite steps and the algorithm terminates.

Definition 3.4 (Closed law). A simple strategy `SimpleStrategy` mf for the type a meets the closed law on some set $S \subseteq a$ if S is closed under the operation $mf\ b$ with any b `::` `SymBool`, i.e., $\texttt{SymBool} \times S \times S \subseteq \mathrm{dom}(mf) \wedge mf(\texttt{SymBool} \times S \times S) \subseteq S$.

Definition 3.5 (Partition law). A sorted strategy `SortedStrategy` idx sub for the type a meets the partition law on some set $S \subseteq a$ if,

(1) it maps every value in S to a sub-strategy, i.e. $S \subseteq \mathrm{dom}(idx) \wedge idx(S) \subseteq \mathrm{dom}(sub)$, and
(2) the sub-strategy for each index should be able to merge the values with that index, i.e., $\forall i \in \mathrm{dom}(sub), \mathrm{PM}(sub(i), \{v \mid v \in S \wedge idx(v) = i\})$.

Definition 3.6 (Strategy depth). We define the depth of a merging strategy as a natural number:

(1) simple strategies have the depth 0, and
(2) the depth for a sorted strategy should be the maximum of the depth of its sub-strategies in the set $\{s \mid \exists v, s = sub(v)\}$ plus one.

We use the function $\mathrm{msdepth}(ms)$ to denote the depth of a merging strategy ms.

Note that the definition is only valid for the strategies without cyclic sub-strategy relation. Otherwise, for example, when a strategy is a sub-strategy of itself, we define its depth as infinity.

Then we can define the hierarchical merging invariant for a union:

Definition 3.7 (Hierarchical Merging Invariant). A union value u of type `Union` a meets the *Hierarchical Merging Invariant* regarding a subset $S \subseteq a$ and a merging strategy ms if u meets the *Hierarchical Simple Merging Invariant* or the *Hierarchical Sorted Merging Invariant* regarding S and ms. We use the predicate $\mathrm{HM}(u, ms, S)$ for u to meet the invariant regarding S and ms.

Definition 3.8 (Hierarchical Simple Merging Invariant). A union value $u =$ `Single` v of type `Union` a meets the *Hierarchical Simple Merging Invariant* regarding a subset $S \subseteq a$ and a simple merging strategy ms if $\mathrm{PM}(ms, S) \wedge v \in S$.

Definition 3.9 (Hierarchical Sorted Merging Invariant). A union value u of type `Union a` meets the *Hierarchical Sorted Merging Invariant* regarding a subset $S \subseteq a$ and a sorted merging strategy ms = `SortedStrategy idx sub` if PM(ms, S) and u is organized as $[u_1, u_2, \cdots, u_n]$ such that

(1) all the leaf values of u are in S, and
(2) for each u_k, there exists an index i_k such that for all leaf value l_{kj} in u_k, $idx(l_{kj}) = i_k$, and
(3) $i_1 < i_2 < \cdots < i_n$, and
(4) sub-trees meet the invariant with sub-strategies, i.e., HM($u_k, sub(i_k), \{v : s \mid idx(v) = i_k\}$).

We can verify that the unions shown in the `mrgIf` example before meets the invariant with the merging strategy defined for `(Integer, Integer, SymBool)`. We will call the type `(Integer, Integer, SymBool)` a *mergeable type*, and the merging strategy here is its *root merging strategy*.

Definition 3.10 (Mergeable type and root merging strategy). A *mergeable type* is a type with a unique *root merging strategy* proper for all the values of the type. A type whose root merging strategy is a simple merging strategy is a simply mergeable type.

```
class Mergeable a where
  mergingStrategy :: MergingStrategy a
mrgIfWithStrategy :: MergingStrategy a -> SymBool -> Union a -> Union a -> Union a
mrgIf :: (Mergeable a) => SymBool -> Union a -> Union a -> Union a
class SimpleMergeable a where
  mrgIte :: SymBool -> a -> a -> a
```

It is easy to see that we can formulate the merging rules with our framework for primitive types discussed before: for `SymInteger`, the root merging strategy would be `SimpleStrategy ite`, and for `Integer`, the root merging strategy would be `SortedStrategy id (SimpleStrategy (\_ t _ -> t))`.

`mrgIf` is implemented by delegating to `mrgIfWithStrategy` with some necessary tricks for usability and efficiency, and we will discuss how to implement these functions later. `mrgIf` is analogous to `if` statement with control flow semantics. It should be called on `Union` values. `mrgIte` is analogous to SMT's `ite`. It is a ternary operator and should be called on simply mergeable types.

For a union wrapping a mergeable type, after merging, it should be a *merged union*.

Definition 3.11 (Merged union). Let a be a mergeable type and ms be its root merging strategy, a union value u of the type `Union a` is *merged* if $HM(u, ms, a)$.

3.2.4 Handling Sum Types. Supporting sum types is simple with our framework. We can simply order the values with the data constructor definition order and use it as the top-level sorting criterion. The *merged invariant* ensures that all values in each subtree have the same data constructor, allowing us to merge them as product types. Here is an example of merged sum type values:

```
data T = A Int | B SymInt | C Int Int
If       condA   (If condA1 (A 1) (A 2))
   (If   condB   (B (ite x y z))
                 (If condC1 (If condC11 (C 1 1) (C 1 2)) (C 2 1)))
```

3.2.5 Handling Arbitrary User-Defined Types. Handling algebraic data types is discussed before. The general merging rules are built into the system via type-class derivation mechanisms. For those complex user-defined non-algebraic data types, the user can implement their own merging strategy as long as their type can be hierarchically partitioned into simply mergeable subsets.

3.3 The Semantics for the Merging Algorithm

This subsection gives details on the ORG merging algorithm. We do so by defining the function `mrgIfWithStrategy` introduced in Definition 3.10. This function is defined with reduction rules for the language of terms given below. Evaluating the function application `mrgIfWithStrategy ms c t f`

is modeled as reducing the term `MrgIf N ms c t f`. The result of the reduction is a `Value` term, which wraps the normalized (i.e., merged) result. The term `MrgIf N` represents **n**on-degenerate `MrgIf` while `MrgIf SS` together with three other terms stand for intermediate terms.

```
data DegenerateVariant = N | SS | SI | IS | II
data Term a = Value (MergingStrategy a) (Union a)
            | MrgIf DegenerateVariant (MergingStrategy a) SymBool (Union a) (Union a)
```

In Fig. 3, we present the big-step semantics of the language; the 'evaluate to' relation is denoted $\Downarrow$. MrgIf* rules are the top-level dispatching rules for `MrgIf N` terms. Rule MrgIfTrue and Rule MrgIfFalse show that when the condition is concrete, merging two symbolic values simply yields the true or false branch. When the condition is not concrete, merging depends on the merging strategy and the shape of the union values. Recall that we have two types of merging strategies: simple and sorted, and the union values can be constructed using `Single` or `If`. For simple strategies, the only invariant-preserving union values are the `Single` values, and Rule MrgIfSimple merges the values using the merging function specified by the strategy. For sorted strategies, we delegate to the other rules shown in Fig. 3. The other rules are organized as four groups (IS* rules are omitted because they are analogous to SI* rules). Each group has its own assumptions and corresponds to a degenerate variant of `MrgIf`. Evaluating a *well-formed term* (Definition 3.12) will always satisfy these assumptions and thus the evaluation will not get stuck (Lemma 3.14).

We briefly describe the rules in the SI* group; other rules work analogously. S and I indicate assumptions on the true and false branches, respectively. S means that all leaf values in the branch have the **s**ame index, and I means that the branch is constructed with **I**`f`. In the group, there are four rules. The rules are based on the comparison of `ti`, `fti`, and `ffi`. When `fti` = `ffi`, assuming that the term `MrgIf SI` is well formed, all the values in `f` must have the same index under `i`, and we degenerate to SS* rules with Rule SIDeg. For example, `Single (Right 2)` and `If a (Right 1) (Right 3)` will be merged with Rule SIDeg under the merging strategy for sum types for `Either Integer Integer` because `ft` and `ff` are both constructed with `Right` and have the same index. If `fti` ≠ `ffi`, we then compare `ti` and `fti` to determine the desired position of `t`. When `ti` < `fti` (Rule SILt), e.g., when `t` = `Single 1` and `f` = `If a 2 3`, we can prove that `ti` is less than the indices of all the values in `f`, we then know that the values in `t` cannot be merged with the values in `f`, and in the sorted result, `t` should go first. When `ti` = `fti` (Rule SIEq), e.g., when `t` = `Single (Left 3)` and `f` = `If a (Left 2) (Right 1)`, the values in `t` and `ft` have the same index `ti`, and we can merge `t` and `ft` using the sub-strategy for `ti`. When `ti` > `fti` (Rule SIGt), e.g., when `t` = `Single 3` and `f` = `If a 1 2`, we know that `ft` has the smallest index and should be the first sub-tree in the result, and we should further merge `t` and `ff` to determine where `t` should go.

The following shows an example derivation tree. `ms` and `msr` are both sorted strategies. `ms` orders `Left` before `Right`, while `msr` sorts `Right` values by the contained concrete integer values. r_{23} is `If c (Right 2) (Right 3)`, and r_{123} is `If (&& (not c) a) (Right 1) (If c (Right 2) (Right 3))`.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\texttt{MrgIf SS msr c (Right 2) (Right 3)} \Downarrow r_{23}}{\texttt{MrgIf N msr c (Right 2) (Right 3)} \Downarrow r_{23}}\,\textsc{SSLt}}{\texttt{MrgIf SI msr c (Right 2) (If a (Right 1) (Right 3))} \Downarrow r_{123}}\,\textsc{MrgIfSS}}{\texttt{MrgIf N msr c (Right 2) (If a (Right 1) (Right 3))} \Downarrow r_{123}}\,\textsc{SIGt}}{\texttt{MrgIf SS ms c (Right 2) (If a (Right 1) (Right 3))} \Downarrow r_{123}}\,\textsc{MrgIfSI}}{\texttt{MrgIf SI ms c (Right 2) (If a (Right 1) (Right 3))} \Downarrow r_{123}}\,\textsc{SSEq}}{\texttt{MrgIf N ms c (Right 2) (If a (Right 1) (Right 3)} \Downarrow r_{123}}\,\textsc{SIDeg}\,\textsc{MrgIfSI}$$

We can prove several properties of our algorithm. First, we define the well-formed terms and the union sizes. Then we show that well-formed `MrgIf` terms can be *deterministically* evaluated to well-formed `Value`s. This ensures that we can implement the algorithm as a terminating function

$$\frac{}{\texttt{MrgIf N ms (Conc True) t f} \Downarrow \texttt{t}} \quad (\textsc{MrgIfTrue}) \qquad \frac{}{\texttt{MrgIf N ms (Conc False) t f} \Downarrow \texttt{f}} \quad (\textsc{MrgIfFalse})$$

$$\frac{\text{notConc(c)}}{\texttt{MrgIf N (SimpleStrategy m) c (Single t1) (Single f1)} \Downarrow \texttt{Single (m c t1 f1)}} \quad (\textsc{MrgIfSimple})$$

$$\frac{\text{notConc(c)} \wedge \texttt{MrgIf SS (SortedStrategy i s) c (Single tv) (Single fv)} \Downarrow r}{\texttt{MrgIf N (SortedStrategy i s) c (Single tv) (Single fv)} \Downarrow r} \quad (\textsc{MrgIfSS})$$

$$\frac{\text{notConc(c)} \wedge \texttt{MrgIf IS (SortedStrategy i s) c (If tc tt tf) (Single fv)} \Downarrow r}{\texttt{MrgIf N (SortedStrategy i s) c (If tc tt tf) (Single fv)} \Downarrow r} \quad (\textsc{MrgIfIS})$$

$$\frac{\text{notConc(c)} \wedge \texttt{MrgIf SI (SortedStrategy i s) c (Single tv) (If fc ft ff)} \Downarrow r}{\texttt{MrgIf N (SortedStrategy i s) c (Single tv) (If fc ft ff)} \Downarrow r} \quad (\textsc{MrgIfSI})$$

$$\frac{\text{notConc(c)} \wedge \texttt{MrgIf II (SortedStrategy i s) c (If tc tt tf) (If fc ft ff)} \Downarrow r}{\texttt{MrgIf N (SortedStrategy i s) c (If tc tt tf) (If fc ft ff)} \Downarrow r} \quad (\textsc{MrgIfII})$$

Assumptions and definitions for SS* rules:
assume notConc(c),
assume ms = SortedStrategy i s,
assume allSameIndex(t), allSameIndex(f),
let ti = i (leftMost t),
let fi = i (leftMost f)

$$\frac{\texttt{ti} < \texttt{fi}}{\texttt{MrgIf SS ms c t f} \Downarrow \texttt{If c t f}} \quad (\textsc{SSLt})$$

$$\frac{\texttt{ti} = \texttt{fi} \wedge \texttt{MrgIf N (s ti) c t f} \Downarrow r}{\texttt{MrgIf SS ms c t f} \Downarrow r} \quad (\textsc{SSEq})$$

$$\frac{\texttt{ti} > \texttt{fi}}{\texttt{MrgIf SS ms c t f} \Downarrow \texttt{If (not c) f t}} \quad (\textsc{SSGt})$$

Assumptions and definitions for SI* rules:
assume notConc(c), ms = SortedStrategy i s,
assume allSameIndex(t), f = If fc ft ff,
let ti = i (leftMost t), fti = i (leftMost ft),
let ffi = i (leftMost ff)

$$\frac{\texttt{fti} = \texttt{ffi} \wedge \texttt{MrgIf SS ms c t f} \Downarrow r}{\texttt{MrgIf SI ms c t f} \Downarrow r} \quad (\textsc{SIDeg})$$

$$\frac{\texttt{fti} \neq \texttt{ffi} \wedge \texttt{ti} < \texttt{fti}}{\texttt{MrgIf SI ms c t f} \Downarrow \texttt{If c t f}} \quad (\textsc{SILt})$$

$$\frac{\texttt{fti} \neq \texttt{ffi} \wedge \texttt{ti} = \texttt{fti} \wedge \texttt{MrgIf N (s ti) c t ft} \Downarrow t'}{\texttt{MrgIf SI ms c t f} \Downarrow \texttt{If (|| c fc)} \; t' \; \texttt{ff}} \quad (\textsc{SIEq})$$

$$\frac{\texttt{fti} \neq \texttt{ffi} \wedge \texttt{ti} > \texttt{fti} \wedge \texttt{MrgIf N ms c t ff} \Downarrow f'}{\texttt{MrgIf SI ms c t f} \Downarrow \texttt{If (\&\& (not c) fc) ft} \; f'} \quad (\textsc{SIGt})$$

Assumptions and definitions for II* rules:
assume notConc(c), ms = SortedStrategy i s, t = If tc tt tf, f = If fc ft ff,
let tti = i (leftMost tt), tfi = i (leftMost tf), fti = i (leftMost ft), ffi = i (leftMost ff)

$$\frac{\texttt{tti} = \texttt{tfi} \wedge \texttt{MrgIf SI ms c t f} \Downarrow r}{\texttt{MrgIf II ms c t f} \Downarrow r} \quad (\textsc{IIDeg1})$$

$$\frac{\texttt{tti} \neq \texttt{tfi} \wedge \texttt{fti} = \texttt{ffi} \wedge \texttt{MrgIf IS ms c t f} \Downarrow r}{\texttt{MrgIf II ms c t f} \Downarrow r} \quad (\textsc{IIDeg2})$$

$$\frac{\texttt{tti} \neq \texttt{tfi} \wedge \texttt{fti} \neq \texttt{ffi} \wedge \texttt{tti} < \texttt{fti} \wedge \texttt{MrgIf N ms c tf f} \Downarrow f'}{\texttt{MrgIf II ms c t f} \Downarrow \texttt{If (\&\& c tc) tt} \; f'} \quad (\textsc{IILt})$$

$$\frac{\texttt{tti} \neq \texttt{tfi} \wedge \texttt{fti} \neq \texttt{ffi} \wedge \texttt{tti} = \texttt{fti} \wedge \texttt{MrgIf N (s tti) tt ft} \Downarrow t' \wedge \texttt{MrgIf N ms tf ff} \Downarrow f'}{\texttt{MrgIf II ms c t f} \Downarrow \texttt{If (ite c tc fc)} \; t' \; f'} \quad (\textsc{IIEq})$$

$$\frac{\texttt{tti} \neq \texttt{tfi} \wedge \texttt{fti} \neq \texttt{ffi} \wedge \texttt{tti} > \texttt{fti} \wedge \texttt{MrgIf N ms c t ff} \Downarrow f'}{\texttt{MrgIf II ms c t f} \Downarrow \texttt{If (\&\& (not c) fc) ft} \; f'} \quad (\textsc{IIGt})$$

Fig. 3. Evaluation rules for Term, IS* rules are omitted as they are analogous to SI* rules

`mrgIfWithStrategy` by sequentially pattern matching with the rules, and the function completely merges mergeable values in the union operands. We also show that the evaluation will run in *bilinear time* on the depth of the merging strategy and the sizes of the union operands. We will only present some key definitions and lemmas; our proof[3] is mechanized and verified in Coq 8.16.0.

The following defines a well-formed term. For `Value`, this notion of well-formedness ensures that the resulting union value meets the hierarchical merging invariant (HM, Definition 3.7). For `MrgIf`, it ensures that the operands meet the invariants, and there is a rule in Fig. 3 to evaluate the term.

Definition 3.12 (Well-formed term). A term t is *well-formed* with respect to a merging strategy ms and a set S if t is

(1) `Value` ms u such that $\mathrm{HM}(u, ms, S)$, or
(2) `MrgIf N` ms c t f such that $\mathrm{HM}(t, ms, S) \wedge \mathrm{HM}(f, ms, S)$, or
(3) a degenerate term `MrgIf` dv ms c t f such that $dv \neq$ `N`, $\mathrm{HM}(t, ms, S) \wedge \mathrm{HM}(f, ms, S)$, and ms, c, t, f meet the shared assumptions in each group, as shown in Fig. 3.

We will prove our lemmas by induction on the construction of the derivation trees for evaluating well-formed terms with the rules in Fig. 3, and we need to establish the induction principle.

Lemma 3.13 (Induction principle for the derivation tree of well-formed terms (most cases omitted)). *To prove that a property P holds for every well-formed term t, we need to:*

(1) (`Value` case) prove that P holds for all well-formed `Value` terms,
(2) (MrgIfTrue case) prove that for every ms, t, f such that `MrgIf N` ms (`Conc True`) t f is well-formed, if P holds for `Value` ms t, then P holds for `MrgIf N` ms (`Conc True`) t f.
(3) (SSEq case) prove that for every ind sub c t f i s, such that `MrgIf SS` (`SortedStrategy` ind sub) c t f is well-formed, if $ind(\mathrm{leftMost}(t)) = i$, $ind(\mathrm{leftMost}(f)) = i$, $sub(i) = s$, and P holds for `MrgIf N` s c t f, then P holds for `MrgIf SS` (`SortedStrategy` ind sub) c t f

In the induction principle, except for the `Value` case, each case corresponds to an evaluation rule. We have omitted most of them. To establish the validity of the induction principle so that we really prove that the property holds for *any* well-formed terms, we need to prove that

(1) the cases covers all the well-formed terms, and
(2) in each case, the assumptions are on "smaller" terms, and
(3) for each case, the "smaller" terms are well-formed.

(1) and (3) can be routinely proved. (2) can be proved by observing that the terms in the assumptions have smaller strategy depth or union operands, or are more degenerate forms of `MrgIf` than the ones in the conclusion. With this induction principle, we can prove the following lemmas.

Lemma 3.14 (Termination). *For all well-formed terms t for type a regarding some strategy ms and subset $S \subseteq a$, there exists a union u such that* ($t =$ `Value` ms $u \vee t \Downarrow$ `Value` ms u) $\wedge \mathrm{HM}(u, ms, S)$.

Lemma 3.15 (Deterministic). *For all well-formed terms t for type a regarding some strategy ms and subset $S \subseteq a$, if $t \Downarrow$ `Value` ms $u1$ and $t \Downarrow$ `Value` ms $u2$, then $u1 = u2$.*

These lemmas imply that we can implement the function `mrgIfWithStrategy` with pattern matching without constructing the intermediate terms, and the result would meet the hierarchical merging strategy; thus, the algorithm will perform a complete merging. Assuming that the extraction of the leftmost leaf takes $O(1)$ time, the complexity of the algorithm would be proportional to the size of the derivation tree and would be bilinear with respect to the depth of the merging strategy and the size of the union operands. As the merging strategy depth is usually a small constant for each type, the algorithm would have a linear-time complexity over the size of the union operands.

[3] https://github.com/lsrcz/grisette-coq

Definition 3.16 (Union size). The size of a union u (usize(u)) is defined as follows:

(1) if $u =$ `Simple` v, then the size is 1, and
(2) if $u =$ `If` $c\ t\ f$, then the size is the sum of the sizes of t and f.

Definition 3.17 (Degenerate Level). The degenerate level deglevel(tm) of a term $tm =$ `MrgIf` $dv\ ms\ c\ t\ f$ is 4 if dv is `N`, or 3 if dv is `II`, or 2 if dv is `SI` or `IS`, or 1 if dv is `SS`. Smaller means more degenerate.

Definition 3.18 (Tree size bound). For a term t for type a that is well-formed regarding some merging strategy ms and subset $S \subseteq a$, we define a bound for the size of its derivation tree:

(1) 0 if $t =$ `EvalValue` $ms\ u$,
(2) $\text{deglevel}(dv) + 4 * \text{msdepth}(ms) * (\text{usize}(t) + \text{usize}(f))$ if $t =$ `MrgIf` $dv\ ms\ c\ t\ f$,

This bound is indeed a bound of the size of the derivation tree, which can be proven by induction.

Lemma 3.19 (Bilinear derivation tree size). *For a term* `MrgIf` $ms\ c\ t\ f$ *that is well-formed regarding some merging strategy ms with depth n and subset* $S \subseteq a$*, the asymptotic bound of the derivation tree size is* $O(\text{msdepth}(ms) \cdot (\text{usize}(t) + \text{usize}(f)))$.

3.4 Extra Information in Union for Implementing the Algorithm

To achieve the desired complexity and implement the merging algorithm safely, we need to maintain additional information along with the `Union` structure.

First, our complexity result assumes that extraction of the leftmost leaf in the tree takes $O(1)$ time. This can be achieved by caching the value of the leftmost leaf in the tree in the `If` constructor.

`If` leftMost(t_1) $c_1\ t_1$ (`If` leftMost(t_2) $c_2\ t_2$ (... (`If` leftMost(t_n) $c_n\ t_n\ t_{n+1}$) ...))

Second, our algorithm assumes that `mrgIfWithStrategy` is always called on merged `Union`s. If our algorithm is implemented in a language or standalone tool, the invariants can be enforced internally and not exposed to the user. However, if we implement our algorithm in a library, the user will be responsible for normalizing and maintaining the invariants, and it is possible that they forget to do so for some values. Our algorithm should be able to handle this. We keep a Boolean variable in the `If` data constructor indicating whether it is already normalized. We can perform our algorithm if this is the case, or we can normalize the whole `Union` bottom-up to regain the invariant.

```
mrgIf cond l r | notMerged l = mrgIf cond (normalizeWithStrategy mergeStrategy l) r
mrgIf cond l r | notMerged r = mrgIf cond l (normalizeWithStrategy mergeStrategy r)
mrgIf cond l r = mrgIfWithStrategy mergeStrategy cond l r -- call our main algorithm
normalizeWithStrategy st u@(If cond l r) | notMerged u =
  mrgIfWithStrategy st cond (normalizeWithStrategy st l) (normalizeWithStrategy st r)
normalizeWithStrategy _ u = u
```

In later sections, to minimize distraction, we will not show these fields.

3.5 Error Handling with Ordered Union

`Either` is a standard functional error handling tool: `Left` values represent failed computations, while `Right` values represent successful computations with result values. This allows us to represent multipath computation with failing paths with `Union (Either err res)`. The user can then arbitrarily define error types to record information about the failure and choose whether to allow the failures to happen when calling the solvers. We provide the following solver interface. The `config` argument configures the SMT solver, and the third `Union` argument is the symbolic evaluation result. The key here is the second. It maps the `Either err res` values (execution results from different paths) wrapped in the result to symbolic Boolean values, and `true` means that the given path is desired.

```
solveFallable :: config -> (Either err res -> SymBool) -> Union (Either err res) -> IO fail model
```

This interface decouples symbolic evaluation and error mechanisms, allowing users to reason about different properties of programs while only writing most of the code once. For example, for verification applications where we want to find the inputs that cause assertion errors, while no assumption errors should occur, the desired paths are the paths that lead to assertion violations. In this case, we can simply treat the errors as assumption or assertion violations by mapping them to `false` or `true`, respectively. We do not have to hardcode `assert` or `assume` in the code and change them when we want to reason about a different property, and we can perform symbolic evaluation once and get many tools by translating the results with different mappings.

If the differences between the errors are not useful for some queries and all of the errors will be translated to the same Boolean value, we can achieve the assertion optimization and avoid paying for the information recorded but not used if we merge things properly. For example, in a system with three types of errors, the user can define the errors as follows. The error type is an algebraic data type, so we can derive the `Mergeable` instance using the rules discussed before.

```
data Error = Error1 | Error2 | Error3 deriving (..., Generic, Mergeable)
```

As the `Either` is also an algebraic data type, it can also be merged with the merging rules described above. The structure of the merge result is `If` $err\_cond\ t_{\text{left\_err}}\ t_{\text{right\_val}}$. All errors are collected in the left subtree, and all values are collected in the right subtree. When all the left values are mapped to the same Boolean values, we can just drop the whole left tree and all the disambiguation conditions. The err_cond is the condition for any error to occur. It does not contain the information to disambiguate the different types of errors, so it is very compact and clean.

In the following example, we show a sequential program with conditionals and errors along with the equivalent Grisette code and merging result. If the user wants to find out if an error could happen but does not care about the type of error, the whole left tree can be translated to `true`, and the whole right tree will be translated to `false`. Then the formula to send to the solver would be `(|| a (ite b c e))`, and there is no duplication in the query. This shows the assertion optimization. Note that even if we do not treat all errors in the same way, we can still generate compact encoding, since the disambiguation conditions are also very compact.

```
-- if a then error Error1 else ()
-- if b then (if c then error Error2 else d) else (if e then error Error3 else f)
mrgIf a (Single (Left Error1)) $ mrgIf b (mrgIf c (Single (Left Error2)) (Single (Right d)))
                                         (mrgIf e (Single (Left Error3)) (Single (Right f)))
If (|| a (ite b c e))
  (If a (Single (Left Error1)) (If b (Single (Left Error2)) (Single (Left Error3))))
  (Single (Right (ite b d f)))
```

The CBMC encoding in Section 2.2 is another way to encode errors, which can be achieved with a different merging strategy. There is no single encoding that beats all other encodings, so we leave it configurable, and provide encodings that are generally efficient. The user can design their own error-handling mechanism with their domain knowledge for further optimization if needed.

4 THE PROGRAMMING INTERFACE

In this section, we outline how Grisette exposes the ORG representation to programmers with convenience type class interfaces (Section 4.1); how it integrates ORG with host language libraries using a monadic interface (Section 4.2); and how these interfaces can be combined to write a sample symbolic application, a synthesizer (Section 4.3).

4.1 Type Classes for Creation and Manipulation of Symbolic Values

To simplify the authoring of symbolic programs, Grisette provides support for the creation, conversion, evaluation, and merging of symbolic values using type classes listed in Fig. 4. The

`Union` data type from Section 3 already abstracts the values stored in the union; in particular, it supports both solvable and unsolvable types that are merged in various ways. To decouple the ORG representation from the particulars of alternative solver back-ends, we further abstract `Union` over the symbolic Boolean type representing internal guards. The user can replace the built-in `SymBool` type with their own representation of symbolic Booleans using the type parameter `bool` in Fig. 4. We also abstract the representation of symbol sets and models using type parameters `symbolSet` and `model`. Grisette asks that these types support some generic operations; for example, `bool` needs to implement `SymBoolOp`, which provides symbolic Boolean operations. We expect users to rarely switch solvers, and hence we hide this generalization with specialized versions of type classes and functions. For example, `Union` specializes the generalized `UnionBase` to the built-in `SymBool` type:

```
data UnionBase bool a = Single a | If bool (UnionBase bool a) (UnionBase bool a)
type Union a = UnionBase SymBool a
```

```
class ToCon sym con where
  toCon :: sym -> Maybe con
class ToSym con sym where
  toSym :: con -> sym
class (Monoid symbolSet) =>
  ExtractSymbolics symbolSet a where
  extractSymbolics :: a -> symbolSet
class EvaluateSym model a where
  evaluateSym :: Bool -> model -> a -> a
class SEq bool a where
  (==~) :: a -> a -> bool
  (/=~) :: a -> a -> bool
class (SEq bool a) => SOrd bool a where
  (<~) :: a -> a -> bool
  (<=~) :: a -> a -> bool
  (>~) :: a -> a -> bool
  (>=~) :: a -> a -> bool
```

```
class Mergeable bool a where
  mergingStrategy :: MergingStrategy bool a
class Mergeable bool a => SimpleMergeable bool a where
  mrgIte :: bool -> a -> a -> a
class (SimpleMergeable1 bool u, Mergeable1 bool u,
  SymBoolOp bool) => UnionLike bool u | u -> bool where
  single :: a -> u a
  unionIf :: bool -> u a -> u a -> u a
  mergeWithStrategy :: MergingStrategy bool a -> u a -> u a
  mrgIfWithStrategy :: MergingStrategy bool a -> ...
  mrgSingleWithStrategy :: MergingStrategy bool a -> ...
class (SymBoolOp bool, Mergeable bool a) =>
  GenSym bool spec a where
  genSymFresh :: (MonadGenSymFresh m, Monad u,
    UnionLike bool u) => spec -> m (u a)
class GenSymSimple spec a where
  genSymSimpleFresh :: (MonadGenSymFresh m) => spec -> m a
```

Fig. 4. Grisette type classes for support of symbolic value manipulations.

The conversion between concrete types and their corresponding symbolic representations is provided by the type classes `ToCon` and `ToSym`. For example, they can convert between concrete `Integer` and symbolic `SymInteger` or `Union Integer` values. The conversion of a symbolic value v to a concrete value is successful only when v is fully concrete. For example, the concrete value `[1,2] :: [Integer]` can be converted to the symbolic value `[1,2] :: [SymInteger]`, after which it can be converted back to `[1,2] :: [Integer]`. In contrast, the conversion of the symbolic value `[a,b] :: [SymInteger]` will fail, returning a `None` value. `ExtractSymbolics` extracts all symbolic constants into a set; for example, the set of symbolic constants of `a+b+c` is $\{a, b, c\}$. This set can be used to build advanced reasoning procedures, such as Grisette's built-in counterexample-guided inductive synthesis (CEGIS) procedure [Solar-Lezama et al. 2006]. `EvaluateSym` evaluates a symbolic value by substituting its symbolic constants with the concrete values given by a model. For example, with the model $\{a \mapsto 1, b \mapsto 2\}$, we can evaluate `a+b+c` to `3+c`. This function can be used to get a concrete result or to refine the constraints and data structures. `SEq` and `SOrd` are symbolic variants of `Eq` and `Ord` type classes. They construct symbolic formulas rather than concrete Boolean results. `Mergeable` and `SimpleMergeable` are generalized versions of the ones in the previous section.

The `GenSym` and `GenSymSimple` are for composable generation of fresh symbolic values. The user can build `genSymFresh` for complex types by composing `genSymFresh` for simple types. The monad `m` with the `MonadGenSymFresh` type class ensures that two calls to `genSymFresh` will return distinct values that contain distinct symbolic constants and allows for the creation of fresh symbolics in a

purely functional way. The spec argument specifies the value space; for example, we can generate a symbolic value representing symbolic lists of symbolic Booleans of lengths 1 to 3 with the specification `ListSpec 1 3 ()` (here `()` specifies the value space for the list elements, and `SymBool` does not require any specification), and the result will be similar to the following `Union` value.

```
If a3 (Single [a2]) (If a4 (Single [a1,a2]) (Single [a0,a1,a2])) :: Union [SymBool]
```

4.2 The Union Monad

The `Union` is a tree with path conditions maintained with internal nodes. It is a monad, and its bind operation is tree substitution. When we call `u >>= f`, the function `f` is applied over the leaf nodes of `u`, and the path conditions in `u` will be kept. This helps to model sequential programs with tree substitution semantics, and conditionals can be modeled with `mrgIf`.

```
instance SymBoolOp bool => Monad (Union bool) where
  Single a >>= f = f a
  If cond ifTrue ifFalse >>= f = If cond (ifTrue >>= f) (ifFalse >>= f)
do v1 <- mrgIf a (return [x1]) (return [y1, y2]) -- If a (Single [x1]) (Single [y1, y2])
   v2 <- mrgIf b (return [x2]) (return [y3])     -- Single [(ite b x2 y3)]
   return $ v1 ++ v2 -- result: If a (Single [x1, ite b x2 y3]) (Single [y1, y2, ite b x2 y3])
```

Note that >>= may not correctly maintain the invariant. As an unconstrained monad, we cannot make any assumption on the result type of >>=, including the `Mergeable` constraint, so we can only use `If` instead of `mrgIf` to construct the result. This forces the user to merge the result explicitly for the binds (do-blocks) by normalization. To reduce this boilerplate, we introduce another monad called `UnionM` (generalized as `UnionMBase`), which is a `Union` with a possibly cached merging strategy.

```
data UnionMBase bool a = UMrg (MergingStrategy bool a) (Union bool a) | UAny (Union bool a)
instance SymBool bool => UnionLike (UnionMBase bool) where -- more cases and functions omitted
  unionIf cond (UMrg st ul) (UAny ur) = UMrg (mrgIfWithStrategy st cond ul ur)
  unionIf cond (UAny ul) (UAny ur) = UAny (unionIf cond ul ur)
```

The `UnionLike` class in Fig. 4 generalizes common operations for union-like containers, including `Union`, `UnionM` and containers transformed with monad transformers. The `unionIf` function generalizes the `If` constructor and can be called in >>=, and the `mrg*` functions are generic interfaces that try to cache the merging strategy. For `UnionM`, `unionIf` merge the results with cached merging strategies when possible, and can be used in >>= to maintain the invariant. This technique is known as knowledge propagation [Kiselyov 2013]. We generalize `mrgIf` for `UnionLike`, and also provide monadic combinator wrappers, for example, `mrgReturn`, to capture the strategies. When the user sticks to the `mrg*` variants, they do not have to manually normalize and maintain the invariants.

Being monadic helps us compose various features onto our core symbolic evaluation semantics, and `UnionLike` provide a unified interface for transformed union containers. For example, we can support errors with `ExceptT err UnionM a`, which wraps a `UnionM bool (Either err a)` value, by implementing the following `UnionLike` instance (only `mrgIfWithStrategy` is shown):

```
instance (SymBoolOp b, UnionLike bool m, Mergeable b e) => UnionLike b (ExceptT e m) where
  mrgIfWithStrategy s cond (ExceptT t) (ExceptT f) =
    ExceptT $ mrgIfWithStrategy (liftMergingStrategy s) cond t f -- lift provided by Mergeable1
```

This approach can also be applied to other monad transformers, and we have implemented the `UnionLike` instance for all transformers from `mtl` and most of the carriers in `fused-effects`.

4.3 Example Program Synthesizer

The usual workflow to build a tool in Grisette is (1) to define data structures (e.g., abstract syntax trees) and type class instances for them, (2) to define the interpreter for the language, (3) to define the symbolic program, its inputs and desired behaviors, and call the solver. In the following code, we show a sample skeleton of a program synthesizer in Grisette following this workflow.

```
1 data CProgram = CInt Integer | CAdd CProgram CProgram | ...
2   deriving (Show, Generics) deriving (ToCon SProgram, ...) via (Default CProgram)
3 data SProgram = SInt SymInteger | SProgram (UnionM SProgram) (UnionM SProgram) | ...
4   deriving (Show, Generics) deriving (ToSym CProgram, EvaluateSym Model, ...) ...
5 data Error = ... deriving (Show, Generics) deriving (Mergeable SymBool, SEq SymBool, ...) via ...
6 data Result = ... deriving (Show, Generics) deriving (Mergeable SymBool, SEq SymBool, ...) via ...
7 interpreter :: SProgram -> ExceptT Error UnionM Result
8 instance GenSym SymBool ProgramSpaceSpec SProgram where ...
9 sketch :: UnionM SProgram
10 sketch = genSym programSpaceSpec "sketch"
11 desiredBehavior :: Either Error Result -> SymBool
12 do res <- solveFallable config desiredBehavior (lift sketch >>= interpreter)
13   case res of
14     Left failure -> print failure
15     Right m -> do -- a model. Values for irrelevant symbolic constants may be missing
16       print m -- The model can be printed as a mapping from symbolic constants to values
17       let program = evaluateSym True m sketch -- True: assign default values for missing constants
18       print (interpret program) -- The evaluation result of the synthesized program
19       let Just concreteProgram = toCon program :: Maybe ConcreteProgramType
20       print concreteProgram -- concrete programs usually have better pretty printers
```

The first six lines define the program syntax, errors, and results. The `CProgram` and `SProgram` are for concrete and symbolic programs, respectively. They have the same AST structure, but `SProgram` has symbolic fields (with solvable values) and `UnionM`s to select among sub-nodes. These types can derive the type classes shown in Section 4.1 with GHC `DerivingVia` extension, including `ToCon` and `ToSym`, which are available for types with aligned structures. In line 7, we write a monadic interpreter for `SProgram`. The `ExceptT` transformer extends `UnionM` with error handling mechanisms, as shown in the last section. The user can call `throwError` to raise an error, and internally the errors and values would be merged as `Either`. We then define how to generate a program sketch from a program space specification in line 8, and use it to generate a sketch. `genSym` calls `genSymFresh` and runs the resulting monad with a unique name to generate fresh symbolic constants. The result is a symbolic program with holes (symbolic constants) to be filled in by the solver. In line 11, we specify the desired behavior by mapping `Either Error Result` to `SymBool`. `SEq` and other type classes are handy for this. In line 12, we interpret the program sketch and call the solver. The result can be pattern-matched for a failure or a model. If the solver returns a model, we can obtain the synthesized program using `evaluateSym`. We can then interpret the synthesized program for the evaluation result. We may also convert it to a concrete program using `toCon`.

5 EVALUATION

We evaluate the Grisette tool in this section to understand the following research questions.

RQ1 Is Grisette expressive compared to the state-of-the-art symbolic host language?
RQ2 How can a static type system help write code with fewer performance and safety bugs?
RQ3 How does Grisette perform compared to the state-of-the-art symbolic host language?
RQ4 How does the default encoding compare with the CBMC encoding for errors?
RQ5 How can memoization further accelerate symbolic compilation?
RQ6 How does Grisette perform compared to single-path systems?

5.1 RQ1: Expressiveness of Grisette

To evaluate Grisette's expressiveness and performance, we ported several benchmarks from the state-of-the-art tool Rosette 4 [Porncharoenwase et al. 2022].[4] Rosette 4's benchmark set

[4] https://github.com/lsrcz/grisette-benchmarks

consists of 16 benchmarks in total. We only ported five of them with less than 1000 lines of code because it takes much effort to fully understand the original code and port it. Table 2 shows the statistics. Grisette code is expected to be longer than the Rosette version because type annotations, Haskell library organization, and do-blocks require more lines. Although the Rosette 3 and Rosette 4 codes have the same LoC, the code may be slightly different. As different solvers have different performances in different projects, we list the solvers in the original benchmark and evaluate Grisette and Rosette both with them. The Rosette versions have already been tuned for performance with SymPro [Bornholt and Torlak 2018] by the SymPro's authors, so we assume that symbolic profiling cannot easily find more performance bugs. To ensure that the performance differences are caused by Grisette itself, including merging algorithms, evaluation models, and various mechanism implementations, all benchmarks are ported by line-by-line translation, unless we need to restructure the code to fit Grisette's type system. Another unavoidable factor that affects performance is that Rosette and Grisette are implemented in different languages. To mitigate this, we implemented a new MEG-based `Union` with the Rosette algorithm adapted to static type settings. This approach reuses all other infrastructure that is not coupled with ORG algorithm, eliminates the impact of dynamic type checking and different SMT term optimizers, and is a better choice for comparing the performance of the core algorithms in a typical Haskell setting.

Table 2. Statistics of the benchmarks

Name	Grisette LoC	Rosette 3/4 LoC	LoC diff (G-R)	Solver
Bonsai-NanoScala [Chandra and Bodik 2017]	663	439	+224 (+51%)	Boolector
Bonsai-LetPoly [Chandra and Bodik 2017]	750	427	+323 (+76%)	Boolector
Cosette [Chu et al. 2017]	661	532	+129 (+24%)	Z3
Ferrite [Bornholt et al. 2016]	538	348	+190 (+55%)	Z3
Fluidics [Willsey et al. 2019]	141	98	+43 (+44%)	Z3
IFCL [Torlak and Bodik 2014]	653	483	+170 (+35%)	Boolector
RegexCont	239	N/A	N/A	Z3
RegexFree	219	N/A	N/A	Z3

The benchmark programs are designed for a variety of different tasks, showing that Grisette is expressive enough for a wide range of tasks previously built with state-of-the-art tools. Ferrite is a tool to specify and check crash-consistency models for file systems. It can verify a file system implementation against a model or synthesize sufficient barriers to ensure that a program with file I/O system calls has the desired crash-safety properties. IFCL is a functional symbolic DSL that specifies and verifys the executable semantics of abstract stack-and-pointer machines that track dynamic information flow to enforce security properties. Fluidics is a small DSL for synthesizing microfluidic array manipulation. Cosette is an automated SQL prover. It executes SQL queries symbolically to decide query equivalence. Cosette heavily relies on Racket's macro system for staged computation. Similar functionalities can be implemented with Template Haskell [Sheard and Jones 2002] as everything in the host language is available in Grisette. We only implemented part of the original system that is relevant to the benchmark and removed irrelevant lines from the Rosette versions, so our LoC metrics differ from those reported in the original Rosette 4 paper. We also found a typographical error in the original implementation. Fixing it makes the Rosette version generate smaller formulas and run faster. Our performance results are based on the fixed version. Bonsai is a data structure specialized for efficient synthesis-based reasoning for type systems. It allows more syntax tree merging and is capable of representing large search spaces in a very compact way. This enables reasoning for real-world type systems and can detect soundness issues in the DOT type system (the underlying type system of Scala 3) [Amin et al. 2016]

with null references and a historical bug involving let-polymorphism with mutable references, for example, ML [Tofte 1990; Wright and Felleisen 1994]. Only the DOT type system is used as a Rosette 4 benchmark. However, as we shall see in later sections, Grisette runs fast on the 5 Rosette 4 benchmarks. Therefore, to further evaluate the performance of Grisette with a larger-scale problem, we also implemented the let-polymorphism benchmark from the original Bonsai paper, based on its repository.

We additionally implemented a backtracking regex synthesizer with both delimited coroutines and free monads. This shows that Grisette is capable of expressing tasks that cannot be easily expressed with Rosette, and we will use them to evaluate the effectiveness of memoization.

5.2 RQ2: Porting Cosette

It is known that tuning the performance or debugging symbolic evaluators can be tricky [Bornholt and Torlak 2018; Torlak 2022], and we have discussed the use of static type systems to avoid performance bugs in Section 2.3.1. In this subsection, we will show that we can improve the development experience with static types with a case study on porting Cosette to Grisette.

In a symbolic host language, to implement a tool, we first design the concrete type and algorithms in mind and then make them symbolic. In Rosette, there is no boundary between concrete and symbolic types, and everything can automatically be symbolic, so we can write code like concrete code and get a symbolic evaluator. This is very convenient, but it can lead to performance problems when too much symbolicness is accidentally introduced with inappropriate representations. Bornholt and Torlak [2018] showed that such problems can be prevalent and provided a profiler-based approach to identify them. For the Cosette project, they successfully identified and fixed one problem, causing a 75-fold speedup, making previously infeasible queries feasible.

The fix is based on the observation that we can avoid introducing some symbolicness with a multiset-based table encoding in Cosette. The equivalent Grisette version of the fix is as follows:

```
type RawTable = [(Data, Multiplicity)]
data Table = Table { ... , tableContent :: UnionM RawTable }
symFilter (\(rowdata, mult) -> pred rowdata) tbl :: UnionM RawTable                -- original
fmap (\(rowdata, mult) -> (rowdata, mrgIte (pred rowdata) mult 0)) tbl :: RawTable -- fix
```

The predicate `pred` is symbolic, so the `filter` call in Rosette works symbolically, which is equivalent to the `symFilter` function in Grisette. It introduces symbolicness in the length of the table type, and the result has to be wrapped in a `UnionM`. The fix avoids this extra symbolicness by setting the multiplicity for the rows that do not meet the predicate to 0 instead of removing it from the table. This does not change the length of the table, and no extra symbolicness is introduced.

Although SymPro is capable of finding some performance bugs like this, it is not complete, and more such bugs can exist. In Grisette, however, if we understand that we do not have to introduce this symbolicness upfront, we can constrain that no such extra symbolicness exists with the types. With the following type, the inefficient overly-symbolized code will be identified by the type checker. We changed the code to make it type-check, and this brings another 8.7x speedup as shown in Table 3 where Cosette-1 is the optimized version and Cosette is the original version.

```
data Table = Table { ... , tableContent :: ~~UnionM~~ RawTable } -- No UnionM here
symFilter :: (UnionLike m, Mergeable a) => (a -> SymBool) -> [a] -> m [a] -- Does not work.
```

This approach to constrain the representation with data type declarations is the most general and easiest way to control merging with domain knowledge in Grisette. More fine-grained control can be performed by manually implementing a `Mergeable` instance, although this is rarely needed.

After we implemented the optimized version in Grisette, which was just simple changes to fix the type errors, we backported the optimized algorithm to Rosette. Without static type checking, our first version was buggy, as we used `car` instead of `cdr` to extract the multiplicity from the

data-multiplicity pairs. In Grisette, these errors can be detected at compile time, but in Rosette, the errors would be caught by Rosette at runtime and converted to assertions, and the only uninformative message we get is that the verifier cannot find a counterexample. Although we easily found this bug with the `symtrace` tool [Torlak 2022] to collect and analyze the execution trace, we believe it is better to have the ability to report bugs to the user statically during compilation.

5.3 RQ3: Running the Benchmarks

To evaluate the performance of Grisette, we mostly followed the benchmark methods of Rosette 4. We compared the running time and encoding size of Grisette, Grisette (MEG), Rosette 4 and Rosette 3 for each benchmark. We also compared the optimized Cosette and its backported versions, which are shown as Cosette-1. We collected all performance metrics using GHC 9.0.2 and Racket v8.1 on an AMD Ryzen 1950X processor at 3.4 GHz with 128 GB of RAM. The SMT solvers we use are Z3 v4.8.8 [Moura and Bjørner 2008] and Boolector v3.2.1 [Brummayer and Biere 2009] compiled locally on Ubuntu 22.04 LTS. Table 3 shows the results. The total time (s), symbolic evaluation time (s), solving time (s), and encoding size ($\times 10^3$) are reported. The symbolic evaluation time includes the time to generate the solvable type encoding (pure evaluation time), lower the solvable terms to SMT terms, and send the terms to the solver. It is collected as the CPU time consumed by Haskell or Racket. Our current lowering implementation is slow (see the E time and PE time of LetPoly in Table 3), and the bottleneck is the external package `sbv` which is much more high-level and does more things than we need. Due to this, for the Grisette benchmarks, the pure evaluation time is also reported. This number will only be used to compare different Grisette approaches. The solving time is calculated by the difference between the total wall clock time and the CPU time. Table 4 shows the best, worst, and geometric mean performance ratios of Grisette over Grisette (MEG), Rosette 3 and Rosette 4 on the benchmarks, including optimized ones, excluding the regex benchmarks, as they are only available in Grisette. The time metrics are shown as speedup ratios, and the term count metric is shown as percentages.

Table 3. Performance results for the ported and optimized benchmarks. T (s), E (s), PE (s), S (s), and Tm ($\times 10^3$) represent total time, evaluation time, pure evaluation time, solving time, and term count, respectively.

Benchmark	Grisette					Grisette (MEG)					Rosette 3				Rosette 4			
	T	E	PE	S	Tm	T	E	PE	S	Tm	T	E	S	Tm	T	E	S	Tm
Ferrite	2.4	.79	.42	1.6	12	8.8	2.3	.53	6.5	56	25	17	8.0	34	34	17	17	40
IFCL	33	4.1	.67	29	91	38	9.4	1.6	29	222	194	14	180	383	147	14	133	438
Fluidics	19	3.3	.52	15	84	198	122	4.3	77	1371	23	8.4	15	284	29	8.8	20	308
Cosette	.27	.094	.045	.18	1.5	1.6	.44	.18	1.1	8.4	16	7.9	7.8	114	12	8.1	4.1	128
DOT	1.8	1.1	.34	.68	21	4.9	4.0	1.2	.92	71	27	23	3.8	664	52	47	4.3	1222
LetPoly	77	13	7.8	64	102	1009	833	108	176	1725	1396	186	1210	4908	1284	142	1142	5152
Cosette-1	.031	.01	.004	.021	.077	.03	.011	.007	.019	.077	.16	.14	.025	.13	.15	.13	.025	.13
RegexCont	3.9	3.6	2.9	.3	16	60	53	13	7.1	311								
RegexFree	4.8	4.5	3.7	.29	16	63	54	14	8.9	318								

Table 3 and Table 4 show that Grisette almost always outperforms Rosette 3/4 in both time metrics and encoding size on the benchmarks, except that Grisette produces a slightly harder-to-solve formula for Fluidics. On average, Grisette accelerates symbolic compilation over the state-of-the-art Rosette 4 tool by approximately 14.1x, generates approximately 91% smaller formulas, and accelerates the solving of the resulting formula by about 5.7x. For some benchmarks, for example, Cosette, the running time and size of the formulas are even more reduced. Considering that Grisette has different algorithms in all kinds of mechanisms from Rosette, to evaluate the effectiveness of ORG semantics, we compared Grisette with a variant of Grisette with MEG

semantics, which shares all infrastructures with Grisette but ORG semantics. To compare the two methods, we suggest comparing the pure evaluation time, solving time, and term sizes, as the MEG version is greatly affected by the slow back-end. Compared to Grisette with MEG semantics, Grisette with ORG semantics is almost always better, except that the MEG version compiled IFCL into larger but easier-to-solve formulas. We noticed that the MEG semantics perform badly, especially on the Fluidics, LetPoly, and regex benchmarks. A reasonable explanation for this is that the three benchmarks create union containers with many branches; for example, in the regex benchmarks, the concrete integers need to be kept separate if they are not equal. This would cause the MEG approach to generate extra large disambiguation conditions.

Table 4. Performance ratios of Grisette over Grisette (MEG), Rosette 3 and Rosette 4

Version	Total time			Eval time			Solve time			Term count		
	best	worst	mean	best	worst	mean	best	worst	mean	best	worst	mean
Grisette (MEG)	13.1x	1.0x	3.7x	64.4x	1.1x	6.1x	6.3x	0.9x	2.4x	5.9%	100.0%	20.8%
Rosette 3	57.2x	1.2x	9.3x	83.3x	2.5x	13.0x	43.5x	0.9x	5.5x	1.3%	60.6%	10.4%
Rosette 4	44.3x	1.6x	10.1x	85.9x	2.7x	14.1x	22.5x	1.2x	5.7x	1.2%	59.2%	8.8%

5.4 RQ4: Compare the Default Error Encoding with the CBMC Error Encoding

To evaluate error encodings, we manually analyzed the benchmark programs and found that IFCL, NanoScala and LetPoly are affected by it. Other programs only use a single error type, and there should be no large differences between the two encodings, so we only ported the three benchmarks to the CBMC error encoding. Table 5 shows that in general the CBMC encoding generates slightly smaller formulas but may be easier or harder to solve than the default encoding based on the ADT merging rules. This indicates that no single error encoding is suitable for every scenario, and the user may try different encodings and choose the fastest one.

Table 5. Performance result for the selected benchmarks with the default encoding and CBMC error encoding

Benchmark	Grisette					Grisette (CBMC error encoding)				
	T	E	PE	S	Tm	T	E	PE	S	Tm
IFCL	33	4.1	.67	29	91	72	4.1	.67	68	91
DOT	1.8	1.1	.34	.68	21	1.6	1.0	.32	.57	20
LetPoly	77	13	7.8	64	102	71	12	7.7	59	91

5.5 RQ5: Effectiveness of Memoization

To evaluate the effectiveness of memoization, we analyzed the ported projects and found that Bonsai had the substructures needed for memoization. The backtracking regex synthesizer can also be optimized, so it is also added for benchmarking. The lowering has performance issues in our scenario: in the NanoScala and Regex benchmarks, lowering is even slower than solving, and this part cannot be accelerated with memoization. Therefore, in this subsection, the reported evaluation time does *not* contain the lowering and the solver interaction times.

Table 6 shows that memoization is capable of improving the performance of symbolic evaluation, further accelerating the symbolic evaluation by 1.6–7.9x and overall performance by 1.2–7.5x.

5.6 RQ6: Performance of All-Path Symbolic Evaluation

To evaluate the performance of all-path symbolic evaluation in various tasks, we compared our system with G2Q [Hallahan et al. 2019a], which is the user-friendly quasiquoter interface to

Table 6. Benchmarks for memoization

Benchmark	no memoization				with memoization				speedup ratio	
	Total	Eval	Lower	Solv	Total	Eval	Lower	Solv	Total	Eval
DOT	2.4	1.8	.71	.68	1.8	1.1	.74	.68	1.4x	1.6x
LetPoly	96	30	4.9	66	77	13	5.1	64	1.2x	2.3x
RegexCont	25	25	.5	.29	3.9	3.6	.64	.3	6.5x	6.9x
RegexFree	36	35	.5	.3	4.8	4.5	.82	.29	7.5x	7.9x

the G2 [Hallahan et al. 2019b] symbolic execution engine in Haskell. Note that G2 is a search-based engine and is not an all-path symbolic evaluator in our definition. G2Q was evaluated with four programs that demonstrate a wide range of applications: an n-queen solver, a solver to find matching strings for regexes, a program verifier for an imperative arithmetic language, and a program synthesizer for lambda expressions with de Bruijn indexing [de Bruijn 1972]. We ported the benchmarks for the four programs to Grisette and compared the performance.[5]

Table 7. G2Q benchmarks

Task	Paper (s)	Reproduction (s)	Grisette (s)
`badEnvSearch prog`	59.32	45.47	0.02
Search for non-zero `Mul x y == Add x y`	0.56	0.44	0.10
`solveDeBruijn` for id	0.04	Runtime Error	0.02
`solveDeBruijn` for const	1.06	Runtime Error	0.02
`solveDeBruijn` for `NOT`	Timeout	Runtime Error	0.02
`solveDeBruijn` for `OR`	86.22	Runtime Error	0.15
`solveDeBruijn` for `AND`	Timeout	Runtime Error	0.40
`solveQueens 4`	0.36	0.27	0.03
`solveQueens 5`	0.55	0.47	0.03
`solveQueens 6`	0.90	0.76	0.03
`solveQueens 7`	1.47	1.26	0.03
`solveQueens 8`	2.30	2.05	0.04
`stringSearch` for `(a+b)*c(d+(ef)*)`	0.26	0.20	0.03
`stringSearch` for abcdef	4.23	3.91	0.12
`stringSearch` for a+b+c+d+e+f	0.05	0.04	0.03
`stringSearch` for a*b*c*d*e*f*	0.02	0.04	0.02

Table 7 shows the performance of G2Q and Grisette. As we failed to reproduce the de Bruijn benchmarks due to runtime errors, we list all the numbers from both the original paper [Hallahan et al. 2019a] and our reproduction. Being a single-path tool allows G2 to be used to reason about unbounded data structures with built-in search strategies, as the paths do not have to be exhausted before we can generate the constraints. This works well for test generation, while poor performance is expected for some nontraditional tasks, including synthesis.

In contrast, Grisette can solve all benchmarks in less than 1 second. We do not have a built-in search procedure in Grisette, but we can easily implement some search strategies directly in the benchmark program. We search iteratively with larger sketches based on some program size measure. This is one of the most common ways to search for similar tools.

6 RELATED WORK

Many programming language and software engineering problems, including program verification, synthesis, model checking, and angelic execution, are enabled by the development of constraint

[5] https://github.com/lsrcz/grisette-g2q-benchmarks

solvers. In these applications, the semantics of the program are generally encoded as constraints, and a constraint solver is called to find solutions to the problem.

The hardest part when implementing these tools is usually reducing program semantics to constraints, as programming to manipulate the constraints can be cumbersome and error-prone. This brings about the need for reusable symbolic evaluators to hide the details, and the user can get symbolic evaluators specialized for their task for free after implementing some interpreters.

Languages with reuseable symbolic evaluators are called symbolic host languages, and they should be versatile enough to support different types of queries. The original single-path symbolic execution may not scale in some scenario, especially for synthesis and bounded model checking queries, in which we are required to reason about all execution paths simultaneously, and path selection techniques do not apply. A proven to work approach to make a symbolic host language versatile is to use all-path symbolic execution with state merging [Torlak and Bodik 2014]. It balances the symbolic evaluation cost and complexity of solver queries and enables large-scale verification and synthesis applications with reusable symbolic host languages.

6.1 Constraint Solving Based Program Reasoning

Constraint solvers are widely used in all kinds of program reasoning. Klee [Cadar et al. 2008a] is a dynamic symbolic execution tool built on top of LLVM. The system uses constraint solvers to automatically generate high-coverage tests. Saturn [Xie and Aiken 2005] is a tool that uses SAT solvers and symbolic execution for path-sensitive intraprocedural analysis with pointers and heap data. CBMC [Clarke et al. 2004] is a model checker for ANSI C that compiles C programs into SMT formulas. Sketch [Solar-Lezama et al. 2006] is a program synthesis tool that compiles programs with holes into SMT formulas and reduces quantifiers with a counterexample-guided inductive synthesis approach. Java Pathfinder [Havelund and Pressburger 2000] is a system that verifies Java bytecode programs with constraint solvers. Angelix [Mechtaev et al. 2016] is a semantics-based automated program repair tool. It generates repairs for large-scale real-world software with angelic execution, where patched expressions are inferred by the constraint solvers.

6.2 Reusable Symbolic Evaluators

Many symbolic evaluators are reusable and capable of hosting DSLs. Boogie [Leino and Rümmer 2010] is an intermediate verification language whose symbolic compiler compiles programs to the input language of the Z3 solver [Moura and Bjørner 2008]. Dafny [Leino 2010] and Spec# [Barnett et al. 2004] are standalone languages for program verification built on top of Boogie. Leon [Blanc et al. 2013] and Kaplan [Köksal et al. 2012] are high-level embedded languages in Scala. Leon supports verification and deductive synthesis queries for programs in a purely functional subset of Scala. Kaplan uses Leon's symbolic compiler to support a restricted form of angelic execution. They support unbounded domains and recursions, while in Grisette, data types or recursions must be bounded with self-finitization. However, as an embedded DSL rather than a simple library, they are not as extensible as Grisette, which works with various functional programming infrastructures and meta-programming constructs. Rubicon [Near and Jackson 2012] lifts a subset of Ruby with symbolic values to specify bounded verification queries about Rails web applications. Rosette [Porncharoenwase et al. 2022; Torlak and Bodik 2013, 2014] is a versatile symbolic host language that lifts part of Racket with symbolic constructs; it generates high-performance formulas for synthesis, verification, angelic execution, and fault localization queries with symbolic DSLs. The system provides the full Racket language including metaprogramming to the user for flexible implementation of SDSLs. Smten [Uhler and Dave 2013, 2014] is a GHC plugin for lifting Haskell with symbolic constructs and solver-aided queries. The Compiling to Categories [Elliott 2017] work is not designed specifically for symbolic compilations. It supports compiling to SMT constraints via

transforming Haskell code by its GHC plugin to computation graphs. G2Q [Hallahan et al. 2019a] uses the G2 [Hallahan et al. 2019b] symbolic execution engine to build solver queries from Haskell code. Another work [Wei et al. 2020] builds a symbolic execution engine compiler in Scala with staging and algebraic effects, but it is not intended to perform an all-path symbolic compilation.

6.3 State Merging Strategies

Although state merging techniques effectively tackle the path explosion problem and make symbolic compilation faster, they also make the resulting constraint harder to solve [Godefroid 2007; Hansen et al. 2009], and may cause more path exploration [Bornholt and Torlak 2018; Kuznetsov et al. 2012]. Kuznetsov et al. [2012] discussed the design space for state merging as a spectrum with two extremes: (1) complete separation of paths, as in the King-style search-based symbolic execution [Boyer et al. 1975; Cadar et al. 2008a,b; Godefroid et al. 2005; King 1975, 1976], and (2) complete static state merging, as in whole-program program reasoning tools [Clarke et al. 2004; Solar-Lezama 2008; Xie and Aiken 2005]. Query count estimation [Kuznetsov et al. 2012] relies on static analysis to identify the states that may be beneficial to merge. Veritesting [Avgerinos et al. 2014] only performs state merging on a subset of states. It classifies the statements into easy statements and difficult statements, and it only performs state merging for easy statements. Rosette [Torlak and Bodik 2014] implements a lightweight symbolic virtual machine which merges states at each control-flow join. The values in the program state are merged with a MEG data structure in a type-driven way. Those values that have the same type (shape) will be merged, such as two lists having the same length. The data structures are merged recursively, requiring the ability to break the boundaries between symbolic types and concrete types, thus only works in dynamic-typed scenario or inside a compiler. MultiSE [Sen et al. 2015] develops a similar approach as Rosette, but it does not merge data structures recursively. In our scenario, where algebraic data types are extensively used, this approach cannot merge well. Sinha [Sinha 2008] also develops state merging strategies similar to Rosette or MultiSE, but uses nested `ites` to represent symbolic states and perform merging with rewrite rules. It does not normalize and merge all mergeable states.

7 CONCLUSION

In this paper, we described the design and implementation of a statically-typed monadic symbolic evaluation library that is efficient and user-friendly. The key to its efficiency is a novel state merging and normalization algorithm with ordered guards state representation. The algorithm provides a clean abstraction for all kinds of data types, which enables high-performance and configurable verification condition generation for free. With static types, the whole system is configurable with type class abstractions, allowing nice integration with the host language libraries. Static types also allow the user to tune the performance more easily and confidently. The monadic abstraction provides a clean interface to symbolic evaluation with configurable mechanisms and unstructured control flows almost for free. Compared to state-of-the-art systems, our system generates more compact formulas in a shorter time, while still being very expressive and user-friendly.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for a careful assessment of the work and very useful comments. This work has been supported in part by the National Science Foundation (NSF) awards No. 2132318, 2122950, 2029457, 1936731, 1918027, 1924435, the Intel and National Science Foundation joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA) under Grant No. 1723352, the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA CMU 1042741-394324 AM01, a grant DARPA FA8750–16–2–0032, as well as gifts from Adobe, Facebook, Google, Intel, and Qualcomm.

REFERENCES

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. *The Essence of Dependent Object Types.* Springer International Publishing, Cham, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Krste Asanović and David A Patterson. 2014. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1083–1094. https://doi.org/10.1145/2568225.2568293

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

Mike Barnett, K Rustan M Leino, and Wolfram Schulte. 2004. The Spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer, 49–69. https://doi.org/10.1007/978-3-540-30569-9_3

Régis William Blanc, Etienne Kneuss, Viktor Kuncak, and Philippe Suter. 2013. *On Verification by Translation to Recursive Functions.* Technical Report.

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 Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 83–98. https://doi.org/10.1145/2872362.2872406

James Bornholt and Emina Torlak. 2018. Finding Code That Explodes under Symbolic Evaluation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 149 (oct 2018), 26 pages. https://doi.org/10.1145/3276519

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

Robert Brummayer and Armin Biere. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 174–177. https://doi.org/10.1007/978-3-642-00768-2_16

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

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008b. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 10 (dec 2008), 38 pages. https://doi.org/10.1145/1455518.1455522

Kartik Chandra and Rastislav Bodik. 2017. Bonsai: Synthesis-Based Reasoning for Type Systems. *Proc. ACM Program. Lang.* 2, POPL, Article 62 (dec 2017), 34 pages. https://doi.org/10.1145/3158150

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *CIDR*.

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Kurt Jensen and Andreas Podelski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. Association for Computing Machinery, New York, NY, USA, 151–160. https://doi.org/10.1145/91556.91622

David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 12 (aug 2017), 25 pages. https://doi.org/10.1145/3110256

N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (sep 2011), 69–77. https://doi.org/10.1145/1995376.1995394

Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. 2006. Modular Verification of Code with SAT. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis* (Portland, Maine, USA) *(ISSTA '06)*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/1146238.1146251

Julian Dolby, Mandana Vaziri, and Frank Tip. 2007. Finding Bugs Efficiently with a SAT Solver. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 195–204. https://doi.org/10.1145/1287624.1287653

Conal Elliott. 2017. Compiling to Categories. *Proc. ACM Program. Lang.* 1, ICFP, Article 27 (aug 2017), 27 pages. https://doi.org/10.1145/3110271

Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 47–54. https://doi.org/10.1145/1190216.1190226

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

William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. 2019b. Lazy Counterfactual Symbolic Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 411–424. https://doi.org/10.1145/3314221.3314618

William T. Hallahan, Anton Xue, and Ruzica Piskac. 2019a. G2Q: Haskell Constraint Solving. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell* (Berlin, Germany) *(Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 44–57. https://doi.org/10.1145/3331545.3342590

Trevor Hansen, Peter Schachte, and Harald Søndergaard. 2009. State Joining and Splitting for the Symbolic Execution of Binaries. In *Runtime Verification*, Saddek Bensalem and Doron A. Peled (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 76–92. https://doi.org/10.1007/978-3-642-04694-0_6

Klaus Havelund and Thomas Pressburger. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 366–381. https://doi.org/10.1007/s100090050043

Joe Hermaszewski. 2021. `vector-sized`. https://hackage.haskell.org/package/vector-sized

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

James Cornelius. King. 1969. *A program verifier*. Ph. D. Dissertation.

James C. King. 1975. A New Approach to Program Testing. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California). Association for Computing Machinery, New York, NY, USA, 228–233. https://doi.org/10.1145/800027.808444

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

Oleg Kiselyov. 2013. *Efficient Set Monad*. https://web.archive.org/web/20201112030922/http://okmij.org/ftp/Haskell/set-monad.html archived 2020-11-12.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell* (Boston, Massachusetts, USA) *(Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 59–70. https://doi.org/10.1145/2503778.2503791

Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. 2012. Constraints as Control. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 151–164. https://doi.org/10.1145/2103656.2103675

Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. (2012), 193–204. https://doi.org/10.1145/2254064.2254088

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

K Rustan M Leino and Philipp Rümmer. 2010. A polymorphic intermediate verification language: Design and logical encoding. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 312–327. https://doi.org/10.1007/978-3-642-12002-2_26

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

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

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Joseph P. Near and Daniel Jackson. 2012. Rubicon: Bounded Verification of Web Applications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 60, 11 pages. https://doi.org/10.1145/2393596.2393667

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

Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. 2022. A Formal Foundation for Symbolic Evaluation with Merging. *Proc. ACM Program. Lang.* 6, POPL, Article 47 (jan 2022), 28 pages. https://doi.org/10.1145/3498709

Richard L Rudell. 1986. *Multiple-valued logic minimization for PLA synthesis.* Technical Report. CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB.

Neil Sculthorpe, Jan Bracker, George Giorgidze, and Andy Gill. 2013. The Constrained-Monad Problem. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 287–298. https://doi.org/10.1145/2500365.2500602

Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 842–853. https://doi.org/10.1145/2786805.2786830

Vaibhav Sharma, Soha Hussein, Michael W. Whalen, Stephen McCamant, and Willem Visser. 2020. Java Ranger: Statically Summarizing Regions for Efficient Symbolic Execution of Java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/3368089.3409734

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

Nishant Sinha. 2008. Symbolic Program Analysis Using Term Rewriting and Generalization. In *2008 Formal Methods in Computer-Aided Design*. 1–9. https://doi.org/10.1109/FMCAD.2008.ECP.23

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

Don Stewart and Duncan Coutts. 2021. `bytestring`. https://hackage.haskell.org/package/bytestring

Mads Tofte. 1990. Type inference for polymorphic references. *Information and Computation* 89, 1 (1990), 1–34. https://doi.org/10.1016/0890-5401(90)90018-D

Emina Torlak. 2016. *unsound behavior?* https://web.archive.org/web/20201122222820/https://github.com/emina/rosette/issues/36 archived 2020-11-22.

Emina Torlak. 2021. *Use early return in rosette?* https://web.archive.org/web/20220223043310/https://github.com/emina/rosette/issues/201 archived 2022-02-22.

Emina Torlak. 2022. *Debugging, Rosette Guide.* https://web.archive.org/web/20220520032819/https://docs.racket-lang.org/rosette-guide/ch_error-tracing.html#%28part._sec~3aerrors-in-rosette%29 archived 2022-05-20.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

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

Richard Uhler and Nirav Dave. 2013. Smten: Automatic Translation of High-Level Symbolic Computations into SMT Queries. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 678–683. https://doi.org/10.1007/978-3-642-39799-8_45

Richard Uhler and Nirav Dave. 2014. Smten with Satisfiability-Based Search. (2014), 157–176. https://doi.org/10.1145/2660193.2660208

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

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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 183–197. https://doi.org/10.1145/3297858.3304027

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Nicolas Wu and Tom Schrijvers. 2015. Fusion for Free. In *Mathematics of Program Construction*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, Cham, 302–322. https://doi.org/10.1007/978-3-319-19797-5_15

Nicolas Wu, Tom Schrijvers, Rob Rix, and Patrick Thomson. 2022. `fused-effects`. https://hackage.haskell.org/package/fused-effects

Yichen Xie and Alex Aiken. 2005. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 351–363. https://doi.org/10.1145/1040305.1040334

Received 2022-07-07; accepted 2022-11-07