Symbolic Compilation as a Monadic Library

ANONYMOUS AUTHOR(S)

The development of constraint solvers has enabled automated reasoning about programs, shifting the engineering burden to implementing symbolic compilation tools that translate programs into efficient constraints. We describe a monadic formulation of a reusable symbolic compiler architecture, Grisette, with which the user can easily implement high-performance symbolic compilers by configuring our system with domain knowledge. Grisette is based on a novel algorithm that efficiently merges and normalizes program states, reducing constraint size and allowing configurable high-performance verification condition generation algorithms to be easily implemented. Being functional, Grisette enables the memoization of symbolic compilation. Being monadic, Grisette allows various mechanisms needed for symbolic compilation of a real-world language, including VC generation and imperative state management, to be implemented almost free. Grisette also allows reasoning about unstructured control flows with efficient state merging, also with monadic abstractions. Compared to state-of-the-art tools, we show acceleration due to efficient merging algorithm and memoization, and faster solving due to more efficient formulas.

Additional Key Words and Phrases: Symbolic Compilation, State Merging, Monadic Programming

1 INTRODUCTION

The development of constraint solvers has enabled a wide 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 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] that translate the semantics of a program into constraints. Recently, symbolic host languages provided reusable symbolic evaluators and allowed the implementation of DSLs that were automatically translated to constraints, easing the engineering burden of implementing a symbolic compiler for the DSL [Torlak and Bodik 2013, 2014]. These host languages were also useful for general purpose languages, by hosting interpreters for LLVM [Lattner and Adve 2004] and RISC-V [Asanović and Patterson 2014], enabling symbolic analysis of programs that compile to these target languages [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:

- *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 of analysis by simultaneously translating into constraints for multiple program paths (or all paths up to a certain execution depth, as in bounded model checking) [Nelson et al. 2019]. All-path execution is particularly important in program synthesis, where the space of candidate programs is represented as a single program with symbolic branches corresponding to decisions in a grammar that describes the candidate space [Solar-Lezama 2008; Solar-Lezama et al. 2006]. By translating all program paths to a single constraint system, a single call to the solver returns a solution that represents the synthesized program. A versatile symbolic host language, supporting various queries and tasks, including high-performance program synthesis, needs to support all-path symbolic execution.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

- *State merging* is an indispensable mitigation to the problem of path explosion that arises in all-path symbolic execution [Baldoni et al. 2018; Godefroid 2007]. The specific challenge is how to merge the program states so that partial evaluation remains effective in the merged state. Rosette [Torlak and Bodik 2013, 2014] and MultiSE [Sen et al. 2015] developed merging algorithms that significantly reduce path explosion despite preserving a concrete program state that allows partial evaluation to remain as effective as if each path was symbolically evaluated on its own.

This paper describes a monadic design with all three properties. It is not a new idea to use monads, algebraic effects, or categories to model symbolic execution. However, we are not aware of any work that defines a symbolic monad that merges symbolic states, which is necessary for advanced analysis and synthesis. Existing work satisfies only two of the properties. For example, symbolic monads for multipath execution have been designed using nondeterministic algebraic effects or monads augmented with state transformer to maintain the path condition for each path [Darais et al. 2017; Mensing et al. 2019; Wei et al. 2020], or to reinterpret the programs with Cartesian closed categories to compile the whole program into a computation graph and then translate it to SMT [Elliott 2017] (as far as we can tell, this work does not support merging of recursive data structures).

The systems that apply all three techniques are non=monadic symbolic compilers: CBMC [Clarke et al. 2004], Sketch [Solar-Lezama 2008; Solar-Lezama et al. 2006], Rosette 3 [Torlak and Bodik 2013, 2014], and Rosette 4 [Porncharoenwase et al. 2022]. Rather than providing a library, these systems lift a subset of their language to the symbolic domain, typically by overriding the interpreter or extending the compiler. Among these systems, Rosette is the state-of-the-art symbolic host language, performing the most advanced merging, thus taking full advantage of partial evaluation.

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

- *Purely functional design.* Purely functional programming 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 2.0x–8.9x, and improved overall performance by 1.2x–8.7x (see Section 5.5).
- *Static types.* Static types prevent unsafe uses of system libraries by preventing symbolic values to flow where concrete values are expected. In the dynamically typed Rosette, some such errors are silently treated as assertion failures, causing unexpected and hard to diagnose behaviors [Torlak 2022]. Additionally, Grisette users can use type classes to extend the system with symbolic variants of user-defined or library data types in a familiar, principled way. We have implemented support for the common data types such as tuples, sum types, bytestrings, and even size tagged vectors. Another benefit provided by the static type system is performance debugging, which boils down to ensuring that the program is written such that concrete values are used wherever possible (symbolic values lead to path explosion, slower compilation and solving). By rejecting programs with symbolic values in concrete parameters, it is easier to debug and tune symbolic evaluators in Grisette. In Section 5.2, we show that static type checking identified performance bugs missed by a dynamic symbolic profiler.
- *Monadic symbolic values.* Since the data structure representing a symbolic value is a monad, we can use monadic frameworks, which simplifies the implementation compared to existing systems. For example, we do not need to implement support for stateful computation with global symbolic evaluator states, as in Sketch or Rosette. We obtain a symbolic compiler for stateful programs for free by applying the `StateT` transformer and implementing a type

class instance as shown in Section 4.2. Second, we can symbolically reason about language constructs that are not supported by previous tools, such as unstructured control flow. With Grisette, we can implement *backtracking* regex synthesizers with coroutines, implemented using delimited continuations or free monad transformers.

- *Symbolic compiler as a library.* As mentioned above, existing systems lift a subset of a language in the symbolic domain, which may cause programmer confusion as to which constructs accept symbolic values. As a library, Grisette does not change the semantics of the language constructs but instead provides typed combinators. Also, being a library, our monadic design serves as an implementation recipe that is language-independent, and thus more portable. In contrast, porting Rosette or other previous symbolic host languages is nontrivial because their design decisions are specific to the host language or the external symbolic evaluation engines. This paper presents our design in Haskell, but we have also developed a prototype in Scala. Both implementations follow the same architecture.

Formulating a monadic library that satisfies all three properties requires us to solve the following challenges:

- *Efficient state merging algorithm.* Designing a merging algorithm that is both computationally efficient and generates small formula 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 separated 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 90% smaller formulas than Rosette on the benchmarks and has about 11.4x overall speedup. This algorithm also gives us the 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 3.3).
- *The constrained-monad problem.* To avoid path explosion, we may need to perform state merging for every intermediate states. However, the monad abstracting the symbolic evaluation semantics cannot make any assumptions about the values in it, in particular that the values support merging, and thus cannot merge states automatically. (This problem is analogous to why sets are not monads [Sculthorpe et al. 2013].) Instead, the user has to merge values by writing much boilerplate code, otherwise facing severe performance issue due to path explosion. We addressed this problem by adapting the knowledge propagation [Kiselyov 2013] technique to our symbolic evaluation monad, enabling automatic merging.

The rest of the paper is organized as follows: we will first give an overview of the efficient algorithm and the user-friendly, highly configurable interface of Grisette in Section 2, and then describe our algorithm in detail in Section 3. Then we will discuss the monadic structure of

Grisette's symbolic state representation and its extensiblity for various programming models in Section 4. We will finally evaluate our tool on several verification and synthesis tools ported from Rosette as benchmarks in Section 5. We show that Grisette is safe, expressive, runs much faster and generates much smaller formulas than the state-of-the-art symbolic host-language.

2 OVERVIEW

This section will discuss the fundamental ideas of Grisette. We will discuss our novel state merging and normalization algorithm based on ordered guards state representation, followed by a discussion of how we can use it to propagate the symbolic evaluator state and generate verification conditions in a purely functional, efficient, and configurable way. Finally, we will show how to integrate the algorithms into a user-friendly library.

2.1 Ordered Guards

State representation is critical in symbolic evaluation systems with state merging. Traditional symbolic evaluation systems separate execution paths and maintain path conditions along each path. State merging allows multiple paths to be executed simultaneously and to be merged. To distinguish execution results among different paths, we need a container to represent the values of each path and record the corresponding path conditions.

One way to design the state representation comes from the original single-path symbolic execution semantics. With this representation, each value is guarded by its path conditions. The path conditions for different values are mutually exclusive, and an individual path condition determines whether the corresponding value is chosen. We call this approach *Mutually Exclusive Guards* (MEG). The system can perform a symbolic evaluation with the MEG semantics by translating `if`s into a library function called `mergeIf`. The results are shown in the second column of the table in Fig. 1. For example: the first line creates a symbolic variable called `x`, whose value is `[a]` when `cond1` is true, or `[b,c]` when ¬`cond2` ∧ `cond2` is true.

The MEG representation has been used in previous symbolic evaluation tools [Kuznetsov et al. 2012; Sen et al. 2015; Torlak and Bodik 2014]. All these tools are a combination of static state merging and path separation [Kuznetsov et al. 2012]. To evaluate `y`, we can perform a local path separation and apply the *concrete* function `head` on each path. This would not change the path conditions, and we will get the result `y` shown in Fig. 1. One of the nice features of our combined approach is that static state merging helps avoid path explosion, and local path separation helps deal with advanced constructs, for example, the lists in Fig. 1, almost free with partial evaluation.

One notable constraint for our system is that we need to minimize the number of unique values in the container so that future path separation can be done without path explosion. This is done with a normalization algorithm. To make it easier to understand, we conceptually decompose it into two steps: grouping and merging. In the grouping step, the values are put into mergeable groups and, in the merging step, the values in each group are combined into a single value.

Despite the nice partial evaluation feature and the easy construction of the merged path conditions, there are some problems. As mutually exclusive path conditions are needed to avoid ambiguity, many conditions are duplicated, which can be barriers for future term optimizations. Here, the system may not be able to determine that the path condition for `a` is equivalent to `cond1` ∨ ¬`cond2` as such reasoning is NP-hard in general. Although Hansen et al. [2009] proposed using heuristic DNF minimization tools (e.g., Espresso [Rudell 1986]) to simplify path conditions and mitigate the problem, they show that many conditions cannot be optimized. Domain knowledge is needed to further simplify the results, which is not easily available in a reusable tool with minimal assumptions on tasks. We can also expect that conditions could be more complicated and deeply nested if there are more guarded values and beyond any available term optimization tool's ability.

```
-- 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)	$\begin{cases} \texttt{[a]} & \text{if} & \texttt{cond1} \\ \texttt{[b,c]} & \text{if} & \underline{\neg\texttt{cond1} \land} \texttt{ cond2} \\ \texttt{[a,c,d]} & \text{if} & \underline{\neg\texttt{cond1} \land \neg}\texttt{cond2} \end{cases}$	$\begin{cases} \texttt{[a]} & \text{if} & \texttt{cond1} \\ \texttt{[b,c]} & \text{else if} & \texttt{cond2} \\ \texttt{[a,c,d]} & \text{otherwise} & \end{cases}$
y (symex)	$\begin{cases} \texttt{a} & \text{if} & \texttt{cond1} \\ \texttt{b} & \text{if} & \neg\texttt{cond1} \land \texttt{cond2} \\ \texttt{a} & \text{if} & \neg\texttt{cond1} \land \neg\texttt{cond2} \end{cases}$	$\begin{cases} \texttt{a} & \text{if} & \texttt{cond1} \\ \texttt{b} & \text{else if} & \texttt{cond2} \\ \texttt{a} & \text{otherwise} & \end{cases}$
y (norm.1)	$\begin{cases} \texttt{a} & \text{if} & \texttt{cond1} \\ \texttt{a} & \text{if} & \neg\texttt{cond1} \land \neg\texttt{cond2} \\ \texttt{b} & \text{if} & \neg\texttt{cond1} \land \texttt{cond2} \end{cases}$	$\begin{cases} \texttt{a} & \text{if} & \texttt{cond1} \\ \texttt{a} & \text{else if} & \underline{\neg}\texttt{cond2} \\ \texttt{b} & \text{otherwise} & \end{cases}$
y (norm.2)	$\begin{cases} \texttt{a} & \text{if} & \texttt{cond1}\underline{\lor}(\neg\texttt{cond1} \land \neg\texttt{cond2}) \\ \texttt{b} & \text{if} & \neg\texttt{cond1} \land \texttt{cond2} \end{cases}$	$\begin{cases} \texttt{a} & \text{if} & \texttt{cond1}\underline{\lor}\neg\texttt{cond2} \\ \texttt{b} & \text{otherwise} & \end{cases}$

Fig. 1. Two representations of symbolic values: mutually exclusive guards (meg) versus ordered guards (org). We underline subterms added to maintain the invariants of the respective representations. The execution for x performs merging of three program paths. The program is internally translated to `mergeIf cond1 [a] (mergeIf 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 finally merging of those values (norm.2).

Another way to represent the state is to use *Ordered Guards* (ORG). For example, for the x value shown in the third column in Fig. 1, the branches are ordered, and a branch can be taken only if all previous branches have not been taken. Then the result is `[a]` if `cond1` is true, or `[b,c]` if $\neg$`cond1` $\land$ `cond2` is true, etc. The advantage of this encoding is that the guards are implicitly mutually exclusive by ordering, and no duplication is needed for disambiguation.

Although using ORG (i.e., nested if-then-else) for state merging in symbolic execution is not new, no previous tool using ORG aims to normalize the representation to minimize the number of values and paths, thus would cause path explosion if we are going to perform local path separation and partial evaluation. For example, Veritesting [Avgerinos et al. 2014], and Java Ranger [Sharma et al. 2020] would only perform obvious optimizations. The tool by Sinha [2008] employs a set of rewriting rules to try to merge possible mergeable terms as much as possible, but it is still incomplete.

Normalizing ORG can be done similarly to normalizing MEG, but it has to be more complicated. This is shown in the third column in Fig. 1. Recall that, in the MEG approach, grouping is easy because the values can be reordered without changing the path conditions. However, with ORG semantics, to reorder the guards, one has to rebuild the path conditions, which may contain duplicate terms. For example, in Fig. 1, to move `a` from the third place to the second place, we need to collect all the path conditions between the current position and the final position, then the condition for the second `a` would be $\neg$`cond2` as shown in `y` (norm. 1). If `a` needs to be moved more than once, the final condition would be even larger, with some conditions duplicated. In this case, the benefit of not requiring duplicate disambiguation conditions is greatly diminished. This approach would work even worse than MEG as we have to do this normalization every time we merge the states.

There is a trade-off between MEG and ORG. MEG requires disambiguation conditions, while ORG may generate large formulas because normalization reorders the values. The naive ORG approach based on a variant of the MEG algorithm may require a lot of expensive reorderings and is not preferable. The ORG approach can win if we do not have to reorder the values significantly.

Based on the observation, we designed our algorithms based on ORG semantics. We aim to reduce reorderings and make each reordering less expensive. We observed that merging with ORG can always be formulated as merging at conditional join points. Normalizing the values for `y` (symex) in Fig. 1 can be viewed as the execution of nested `mergeIf` s. Thus, we must design how to perform merge and normalization with `mergeIf`. We reduce the number of reorderings by maintaining the invariant that every ORG container is normalized in the same sorted way. For example, if the states are *concrete* integers, and different integers will not be merged using SMT `ite` operators as in previous systems, we can sort them by values. Our systems support both encodings, and this concrete integers in ORG container encoding can be used to avoid *missed concretization* [Bornholt and Torlak 2018] statically with types, as further evaluations would have the exact set of possible concrete values and perform path separation. Fig. 2a shows a valid container. Fig. 2b shows an invalid example where two mergeable values (two 1s) coexist. Fig. 2c shows another invalid example, where the values are not sorted in the container. When all the states that passed to `mergeIf` maintain the invariant, we can normalize very quickly using a mergesort-style merge procedure without reordering, shown in Fig. 2d. Moreover, this merging can be done in linear time and generates very compact conditions.

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

(a) Valid

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

(b) Invalid, contains mergeable values

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

(c) Invalid, not sorted

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

(d) Mergesort-style merge

Fig. 2. ORG with sorted invariant

In this section, all the discussion on ORG assumes that the values and conditions are maintained as a simple list. That is, if the container is represented as nested `If`s, the then branches are always a single value and nested `If`s can only occur in the else branch. See the following as an example.

```
If a 1 (If b 2 (If c 3 4))
```

Our algorithm will also take advantage of the fact that nested `If` can be a tree, rather than just a simple list, and will further improve performance and generate smaller formulas with a hierarchical merge and normalization algorithm. We will elaborate on this in Section 3. Our empirical study shows that our system generates 91.5% smaller formulas than the previous state-of-the-art system Rosette 4 [Porncharoenwase et al. 2022], or 80% smaller formulas than our system implemented with MEG semantics.

2.2 Propagating purely functional symbolic evaluator states

With symbolic evaluation tools, we specify the desired behavior of a program with assumptions and assertions. The verification condition (VC) generation procedure [King 1969] translates the behavior into constraints. This subsection analyzes several existing VC optimizations and shows that our merging algorithm can achieve these optimizations without dedicated optimizations algorithms — it suffices to suitably order the values in the ORG representation.

Rosette 3 (a legacy version of Rosette) [Torlak and Bodik 2014] developed a method to separate the assertion conditions from non-error program state (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 primitive types, or {a if c1, b if not c1} for complex types.
-- assertion store: c∧(¬c1∨c2)∧(c1∨c3).
```

To evaluate the program, Rosette 3 splits paths under the condition `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 $\neg pc \lor c$ (i.e. $pc \implies c$) would be added to a global assertion store. This indicates that the program will violate the current assertion if the current path is not executed or the asserted condition is true. All the assertions will then be connected using conjunctions to make sure that no assertions will be violated. The algorithm has two good properties. The first is that the `assert` statements would not impact subsequent execution because the symbolic value of `x` does not depend on `c2` or `c3`. The second is that the clauses produced by different `assert` statements do not depend on each other, and thus the conditions for the assertions produce different clauses in the DNF 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 is necessary to duplicate some conditions when they are interleaved. However, Rosette 4 does not preserve the assertion optimization even if the program is assumption-free, and this issue has already been discussed by the authors [Porncharoenwase et al. 2022]. As a simple example, the following shows the constraint generated by Rosette 4 for the previous Rosette 3 code, and 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 scenario. 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, 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.

The execution of sequential programs can be achieved by tree substitution on the `Right` values in ORG containers. We will show that this operation is monadic and is easy to program in the next section. 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 algorithm for optimization. They can be defined by the user as simple data types, and we can implement the Rosette 3 example with them.

```
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; a) else (assert c3; b)
```

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

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

It will be further normalized into the following 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 `ITE`, 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 3.3.

```
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.

In addition to efficient generation of VCs, 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 think about how to encode correctness criteria as assertions and assumptions. They can define their errors, using their domain knowledge, and program with familiar FP constructs to handle errors. The decoupling of the assertion and assumption constructs from the program is very useful for building more versatile tools, as the errors do not have predefined semantics and can be interpreted in different ways to fit different scenarios. The user will also be able to do things that were impossible or hard in previous systems. 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, this would be very easy 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 static 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 Monadic design with reduced implementation work and increased generality. `Union` is an ORG-based symbolic program state container to represent path-condition-guarded unions of values, and is at the heart of Grisette. The definition of it is straightforward and is simply an if-then-else tree.

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

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. The bind operation of the `Union` data structure does not perform state merging and normalization, and to use the container, the user is responsible to normalize it with much boilerplate. To reduce this boilerplate, we get into the constrained monad problem [Sculthorpe et al. 2013] because it is not possible to resolve the merging rules with the bind operation. We adapted the knowledge propagation [Kiselyov 2013] approach to our `Union` and got the `UnionM` monad, which will be discussed in Section 4.3. The simple idea is that `UnionM` captures and propagates the `Mergeable` knowledge to the bind operation if possible, and the bind operation can use the captured `Mergeable` instance to merge the results.

The following shows the Grisette code modeling the program shown in Section 2.1, with the `UnionM` container. Instead of using a lifted language, the user writes code with combinators. Here, `mrgIf` is `mergeIf` from Section 2.1, and `mrgReturn` is a wrapper for `return`, but it also captures and propagate the `Mergeable` knowledge to merge the result of the whole do-block. It is worth noting that this knowledge will be propagated further when the result of the do-block is used, and this helps to keep all intermediate values merged and 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 us to model various mechanisms with the monad transformer ecosystems, and the error handling mechanism defined in the last section can be easily supported boilerplate-free with the `ExceptT` transformer. Imperative programming can also be adapted to Grisette almost for free, using `StateT` to manage the global state with user-defined data structures. Our approach is more flexible and general than previous systems. For example, it is hard to symbolically evaluate programs with complex unstructured control flows. This requires special handling in previous systems [Torlak 2021], but can be supported with monad transformers in Grisette. We can support delimited continuations with `reset/shift` [Danvy and Filinski 1990] through `ContT`, and we used it to build a *backtracking* regex synthesizer with coroutines. We can even memoize the monadic coroutine computations, which is evaluated in Section 5.5. Our system is also compatible with *some* effect systems, for example, the effect system with fusion techniques [Wu and Schrijvers 2015]. This allows the user to implement various mechanisms in a more flexible way, and we have implemented support for the `fused-effects` [Wu et al. 2022] library. Other effect systems are inherently unfriendly to state merging (not just for our system) as they are deeply embedding the intermediate results. We can only merge the computation, which can not reduce as many paths as merging the values, and we cannot get the intermediate values to merge effectively.

Several monadic or effect stacks have been proposed for the 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.

2.3.2 *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, even though a dynamically typed hash table should in principle accept any types of value, including symbolic values. This can result in an unmerged state and unexpected behavior:

```
(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 static typing and 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), 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
  a ==~ b = ...
instance (ConcreteType K, SimpleMergeable V) => Mergeable (HashMap K V) where
  mergingStrategy = ...
```

Being able to configure functionalities through types helps us integrate Grisette closely with the host language libraries. To support this claim, we added support for the `bytestring` [Stewart and Coutts 2021] and `vector-sized` (Sized tagged vectors) [Hermaszewski 2021] packages without modifying the core Grisette package.

Using types to configure state merging is useful for performance tuning and avoiding *missed concretization* bugs [Bornholt and Torlak 2018; Kuznetsov et al. 2012]. For example, the following are two different state representations, where the first is represented as `UnionM` of concrete integers, and the second is a symbolic integer represented as an SMT formula.

$$\begin{cases} 1 & \text{if} \quad \texttt{a} \\ 2 & \text{else if} \quad \texttt{b} \\ 3 & \text{otherwise} \end{cases} \quad \text{or} \quad \texttt{(ite a 1 (ite b 2 3))}$$

If we use them to index a list, for the first representation, we can distribute the list indexing function through the three concrete branches in `UnionM`, and no infeasible value would be returned. However, with the second representation, the system has to compare it with all possible indexes `0, 1, ...` as there is no general algorithm to efficiently extract all possible concrete values from an SMT formula. This would generate larger results with infeasible values where `(ite a 1 (ite b 2 3))` equals 0. In Grisette, the two representations have the type `UnionM Integer` and `SymInteger`, respectively. The type constrains the desired representation, and thus constraints that no performance problem due to undesired representation can happen. We used this feature to implement a backtracking regex synthesizer efficiently without path explosion as we need to backtrack and restart matching from concrete indexes, and in Section 5.2, we will show that with static types, we successfully found and fixed more performance bugs that have already been tuned for performance with symbolic profiling tools [Bornholt and Torlak 2018].

3 UNION TYPE WITH ORDERED GUARDS SEMANTICS

In this section, we will describe our union data structure with ORG semantics without taking advantage of its monadic nature. The algorithm is not dependent on the monadic structure and may be applied to nonmonadic, nonlibrary systems, while is designed for statically typed libraries.

3.1 Union data structure with ordered guards semantics

We begin our design with the container to represent the merged states of the program with the ORG semantics. Our container is defined as the following ADT:

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

For a single value, under the path condition `True`, we can wrap it in `Single`. For an ordered guard with symbolic condition, the `If` data constructor models its semantics. As discussed before, the branchings will be modeled using `If`, and sequential programs will be modeled by substituting the `Single` terms with new `Union`s. This encoding has the same structure as the control flow, and the ORG semantics ensures that it is unambiguous, so we do not have to duplicate the conditions.

3.2 State merging and normalization with ORG Union

In this section, we will describe our state merging and normalization algorithm for the ORG `Union` data type. Although several state merging and normalization algorithms have been proposed in previous work, our system requires a new one. One reason is that previous normalization algorithms that meet our need are all for MEG semantics, and another reason is that our system is a statically and strongly typed library without any modification to the compiler, and we need to merge the values in a type-safe way. For example, to normalize `If a (Single (1, [1]) (Single (2, [2, 3]))`, which has the type `UnionM (Int, [Int])`, the Rosette's type-driven algorithm may merge the values structurally into a tuple similar to `((ite a 1 2), If a (Single [1]) (Single [2, 3]))`, which has the type `(SymInt, UnionM [Int])`, and is not compatible with `UnionM (Int, [Int])` type. Our goal is to create a modular abstraction that works efficiently for both primitive types and user-defined types, particularly algebraic data types. For efficient hierarchical merging of arbitrary algebraic data types or types with user-defined merging rules, our algorithm normalizes the containers to maintain a representation invariant, and the intuition will be demonstrated in several subsections. In subsequent sections, we will show that this algorithm allows us to treat errors and values in a unified manner, allows us to use arbitrary error types, allows assertion optimization, and allows our `Union` monad to work with various monad transformers.

3.2.1 Merging Symbolic Primitives in Union. In our strongly typed system, we will not automatically unwrap the symbolic primitives wrapped in a `Union` and translate the `if` statement directly to `ite` as previous algorithms do. However, merging symbolic primitives in a `Union` is simple, as we can always merge them into a single value with SMT's `ite` operator.

```
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 Dealing with Concrete Integers efficiently. As shown in Section 2.3.2, our system supports both symbolic integers (represented as SMT terms) and concrete integers (the primitive integer type in the host language), and they have different types. This distinction is useful for controlling the concretization and allowing for more optimization 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 use `SymInteger` for symbolic integers, or use `UnionM Integer` to represent a set of concrete integers under some path conditions.

The merging for `SymInteger` is trivial with `ite`, but that won't work for the `Integer`s. The only way to merge concrete integers is to find and use rewriting rules to merge identical numbers:

```
    If a (Single 1) (If b (Single 2) (Single 1))
==> If a (Single 1) (If (neg b) (Single 1) (Single 2))
==> If (a || (neg b)) (Single 1) (Single 2)
```

A naive approach can be difficult to scale if it has to make $O(n^2)$ comparisons for a program with n possible execution paths to find all identical numbers. Sorting the integers before looking for identical ones is a better approach. This can reduce the time complexity to the sorting complexity $O(n \log n)$. However, sorting involves expensive reordering of the values in the ORG semantics, and is not efficient. We can do even better in our scenario. Because we may perform state merging every time there is a conditional join, we can maintain some invariant to speed up the algorithm.

```
mrgIf a (mrgIf ...) (mrgIf ...)
```

Definition 3.1 (Sorted Invariant). A value `v :: Union Integer` meets the *Sorted Invariant* if `v` is organized in the following form

`If` c_1 `(Single` i_1`) (If` c_2 `(Single` i_2`) (... (If` c_n `(Single` i_n`) (Single` i_{n+1}`)) ...))`

where i_k are distinct and sorted in ascending order.

A `Union` satisfying *Sorted Invariant* can be viewed as a sorted list, and a mergesort-style merge can be performed in $O(n)$ time. Similar to a recursive functional mergesort-style merge, the merge can be implemented as a set of rewriting rules that preserve the symbolic semantics and maintain the correct path conditions. Due to the limited space, we will not show the complete algorithm, and the following is part of the algorithm.

```
mrgIf cond l@(Single a) r@(Single b) |a < b = If cond l r |a == b = l |a > b = If (not cond) r l
mrgIf cond l@(If cl sl@(Single vl) ul) r@(If cr sr@(Single vr) ur)
  | vl < vr = If (symand cond cl) sl $ mrgIf cond ul r
  | vl = vr = If (ite cond cl cr) sl $ mrgIf cond ul ur | ...
```

The following shows the clean and compact merging result.

```
mrgIf a
  (If cond1 (Single 1) (If cond2 (Single 2) (Single 4)))
  (If cond3 (Single 2) (If cond2 (Single 3) (Single 4)))
==> If (a && cond1) (Single 1)
      (If (ite a cond2 cond3) (Single 2) (If (neg a && cond2) (Single 3) (Single 4)))
```

Sorted Invariant can also be generalized to a type a if the values of the type a can be sorted by an injective mapping from a to a type with a `Ord` instance.

Definition 3.2 (Generalized Sorted Invariant). A union value u of type `Union` a meets *Generalized Sorted Invariant* with respect to the indexing function idx `::` a `->` b if (1) idx is injective, and (2) there exists a total order R on b, and (3) u is organized in the following form.

`If` c_1 `(Single` v_1`) (If` c_2 `(Single` v_2`) (... (If` c_n `(Single` v_n`) (Single` v_{n+1}`)) ...))`

where $idx(v_k)$ are distinct and sorted by R in ascending order.

3.2.3 Handling Product Types. Algebraic data types are nice abstractions of data, and we will start with product types in this section.

A simple approach to merging product types is to define an indexing function that maps the products to orderable values. This approach, however, will not work well for tuples containing symbolic terms because it is uncommon for two terms to be identical. In this case, the entire `Union` will degenerate into a very long list that collects all the single values while merging almost nothing.

A better approach is to define an indexing function that only lexicographically orders the concrete fields. This indexing function is no longer injective; instead, it groups the values. The concrete fields in each group are identical, so we can safely merge the values in each group by merging the symbolic fields with some merge function. In practice, however, this approach is still suboptimal for two reasons: (1) inefficient symbolic evaluation for very long lists and (2) generation of larger and redundant terms. Take the following code as an example. To find a correct position for `Single (9, 1, f)`, the algorithm has to iterate through the entire list for the then branch and duplicate the condition cond everywhere in the result.

```
mrgIf cond
  (If cond1 (Single (1, 1, a)) (If cond2 (Single (1, 2, b))
    (If cond3 (Single (2, 3, c)) (Single (2, 4, d)))))
  (If cond4 (Single (1, 1, e)) (Single (9, 1, f)))
==> If (ite cond cond1 cond4) (Single (1, 1, (ite cond a e)))    -- cond duplicated here twice
       (If (cond && cond2) (Single (1, 2, b))                    -- cond duplicated here
           (If (cond && cond3) (Single (2, 3, c))                -- cond duplicated here
               (If cond (Single (2, 4, d)) (Single (9, 1, f))))) -- cond duplicated here
```

Our approach hierarchically merges values using the tree structure of `Union`, rather than only the degenerate linked list structure. To do this, we first define the hierarchical merging strategy.

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

Then we can define the hierarchical merging invariant. The idea is that we can hierarchically partition the values in a `Union`, and the value sets generated by each partition should be orderable. The values will eventually be partitioned into small sets that can be merged into single values.

Definition 3.3 (Hierarchical Merging Invariant). A union value u of type `Union` a meets the *Hierarchical Merging Invariant* regarding a subset s of values of type a and a merging strategy ms if

(1) u meets the *Hierarchical Simple Merging Invariant* regarding s and ms, or
(2) u meets the *Hierarchical Sorted Merging Invariant* regarding s and ms

The `SimpleStrategy` wraps a merge function. The merge function does not have to be total; it only captures the fact that a subset of values of the type can be merged simply with a merge function, and the result should still lie in that subset.

Definition 3.4 (Hierarchical Simple Merging Invariant). A union value u of type `Union` a meets the *Hierarchical Simple Merging Invariant* regarding a subset s of values of type a and a simple merging strategy `SimpleStrategy` mf if

(1) there exists a value v with the type a, such that $u =$ `Single` $v \wedge v \in s$, and
(2) let the set of all symbolic booleans be b, $b \times s \times s \subseteq \mathrm{dom}(mf) \wedge mf(b \times s \times s) \subseteq s$.

The `SortedStrategy` consists of two functions: an indexing function and a substrategy function. The indexing function maps a subset s of values of type a to values of an orderable type. The indexing function is no longer injective in this case, but instead partitions the set s into smaller subsets, with each partition corresponding to a subtree in the `Union`. The subtrees will be sorted with the index and then merged by the strategies given by the substrategy function.

Definition 3.5 (Hierarchical Sorted Merging Invariant). A union value u of type `Union` a meets the *Hierarchical Sorted Merging Invariant* regarding a subset s of values of type a and a sorted merging strategy `SortedStrategy` idx sub if

(1) $s \subseteq \text{dom}(idx)$, and $idx(s) \subseteq \text{dom}(sub)$, and
(2) u is organized in the following form
    `If` c_1 t_1 (`If` c_2 t_2 (... (`If` c_n t_n t_{n+1}) ...))
  and
  (a) for each t_k, there exists an index i_k such that for all `Single` $l_{k,j}$ in t_k, $idx(l_{k,j}) = i_k$, and
  (b) i_k is sorted in ascending order, and
  (c) t_k meets the *Hierarchical Merging Invariant* regarding the subset $\{v : s \mid idx(v) = i_k\}$ and the merging strategy $sub(i_k)$.

Then we can define a mergeable type, as well as its root merging strategy and *Merged Invariant.* After merging, a `Union` value should satisfy the *Merged Invariant.*

Definition 3.6 (Mergeable type). A *mergeable type* is a type with a unique root merging strategy. The root merging strategy is resolved with the `Mergeable` type class. A type whose root merging strategy is `SimpleStrategy` is an instance of `SimpleMergeable`.

```
class Mergeable a where
  mergingStrategy :: MergingStrategy a
class SimpleMergeable a where
  mrgIte :: SymBool -> a -> a -> a
```

Definition 3.7 (Merged Invariant). Let a be a *mergeable type*, A union value u of the type `Union` a meets the *Merged Invariant* if u meets the *Hierarchical Merging Invariant* regarding the set of all values of type a and the root merging strategy ms of type a.

The merging rule for the types discussed before can be formalized using the framework. For symbolic integers, the root merging strategy is `SimpleStrategy ite`. And `SortedStrategy id (\_ -> SimpleStrategy $ \_ v _ -> v)` is the root merging strategy for concrete integers.

Now, we can define the merging strategy for product types by hierarchically sorting concrete fields. For example, for `(Integer, Integer, SymInteger)`, the root strategy is a sorted strategy with an indexing function extracting the first element. The substrategies will also be sorted strategies with indexing functions extracting the second elements. Then the sub-substrategies will be simple strategies with the merge function `\cond (i1, i2, i3) (_, _, j3) -> (i1, i2, ite cond i3 j3)`.

For a *mergeable type a*, to perform `mrgIf` for `Union` a that conforms to *merged invariant*, we can perform hierarchical merge-sort style merging. In the following code, condition `cond` is duplicated only when necessary. For those subtrees whose index only exists in one branch, the algorithm does not have to iterate through and duplicate the conditions for that tree, for example, in the following code, we do not have to look into the `(2,*,*)` subtree because it only exists in then branch.

```
mrgIf cond
  (If cond1 (If cond2 (Single (1, 1, a)) (Single (1, 2, b)))
            (If cond3 (Single (2, 3, c)) (Single (2, 4, d)))) -- no need to look into
  (If cond4 (Single (1, 1, e)) (Single (9, 1, f)))
==> If (ite cond cond1 cond4)
      (If (|| cond2 (neg cond)) (Single (1, 1, ite cond a e)) (Single (1, 2, b))) -- (1,*,*)
      (If cond (If cond3 (Single (2, 3, c)) (Single (2, 4, d)))                   -- (2,*,*)
               (Single (9, 1, f)))                                                -- (9,*,*)
```

3.2.4 *Handling Sum Types.*

Supporting sum types is simple with hierarchical encoding. 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. The values are first sorted by the constructors, and then merged as product types.

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

3.2.5 Extra information in `Union` *for implementing the algorithm.* We will not go into detail about the hierarchical merging algorithm because it is simply a composition of merge-sort style merge algorithms, but we will highlight some extra information that needs to be maintained along with the `Union` in order to implement the merging algorithm efficiently and safely.

When the algorithm needs to merge two unions that satisfy *Hierarchical Sorted Merging Invariant*, it needs to determine the indices for each subtree t_k by extracting some leaf element from t_k and applying the indexing function on it. Because the algorithm must traverse some path to find a leaf, extracting a leaf from the tree can be inefficient. To speed things up, we can cache the value of the leftmost leaf in the tree in the `If` constructor, and the extraction will take $O(1)$ time.

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

If our algorithm is implemented in a language or a tool for symbolic evaluation, 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 in each step, and it is possible that they forget to do so for some states. Our algorithm should be able to handle this, though there may be efficiency loss. To address this, we keep a Boolean variable in the `If` data constructor that indicates 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@NotMerged r = mrgIf cond (normalizeWithStrategy mergeStrategy l) r
mrgIf cond l r@NotMerged = mrgIf cond l (normalizeWithStrategy mergeStrategy r)
mrgIf cond l r = mrgIfWithStrategy mergeStrategy cond l r -- call our main algorithm
normalizeWithStrategy st (NotMergedIf cond l r) =
  mrgIfWithStrategy st cond (normalizeWithStrategy st l) (normalizeWithStrategy st r)
normalizeWithStrategy _ u = u
normalize = normalizeWithStrategy mergeStrategy
```

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

3.3 Error Handling with Ordered Union

`Either` is a standard tool for handling errors in functional programming languages. The left values represent failed computations, while the right values represent successful computations with some result values. This allows us to represent multipath computation with failing paths using the type `Union (Either err res)`. The user can arbitrarily define the error type to record some useful information about the failure. After symbolic compilation, the user can treat these errors as assertion or assumption violations to construct solver queries with the following type classes.

```
class SolverErrorTranslation spec e | spec -> e where
  translateError :: spec -> e -> Bool
class SolverErrorTranslation spec e => SolverTranslation spec e v | spec -> e v where
  translateValue :: spec -> v -> SymBool
```

The `translateError` function interprets errors, while the `translateValue` function specifies the correctness condition for successful computations. If the user wants to know if a specific error can be thrown, they can use the two functions to translate that error into `True`, other errors into `False`, and all values into `False`. If the user wants to find a specific input that causes the result to meet some criteria, they can build that criteria into `translateValue` function, and translate all errors into `False`. The user does not have to hardcode `assert` or `assume` into the code, and can perform symbolic evaluation once and get many tools by translating the results with different `spec` specifications.

If the differences between the errors are not useful for some query 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 may 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 be merged with the merging rules described before too. The structure of the merging result is `If` err_cond $t_{\text{left\_err}}$ $t_{\text{right\_val}}$. All the errors are collected in the left subtree, and all the 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 happens. It does not contain the information for disambiguating the different type of errors, thus 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 any error could happen, but they do not care about the type of the 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, as the disambiguation conditions are also very compact.

```
-- if a then error Error1 else ()
-- if b then (if c then error Error2 else d) (if e then error Error3 else f)
mrgIf a (Single (Left Error1)) $
  mrgIf b (mrgIf c (Single (Left Error1)) (Single d))
          (mrgIf e (Single (Left Error2)) (Single 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. The user can feel free to design their own way to handle errors with their domain knowledge for more efficient VC generation. There is no single encoding that beats all other encodings, so we believe that we should leave it configurable, and provide some encodings that we believe are efficient in many scenarios.

4 MONADIC PROPAGATION OF SYMBOLIC STATE

In the last section, we showed how `Union` can be used to model program states. However, it can be difficult to directly manipulate the `Union` data structure, and when the whole system is built as a library, we need a user-friendly interface. In this section, we will talk about the monadic structure of `Union` and how it can be used to model a whole program. We found that the main usability problem with using `Union` in the real world is that we have to merge the results by normalizing everywhere. We will discuss how we overcome this and justify our design choice.

4.1 The Union monad and control flow

The `Union` is simply a tree with path conditions maintained with the internal nodes, and the bind operation of its monad instance is tree substitution. When we call `u >>= f`, the function `f` is mapped over the leaf nodes of `u`, and the path conditions in `u` will be kept.

```
instance Monad Union where
  Single a >>= f = f a
  If cond ifTrue ifFalse >>= f = If cond (ifTrue >>= f) (ifFalse >>= f)
```

Then we can model sequential programs with ease, and conditionals can be modeled with `mrgIf`.

```
-- v1 = if a then [x1] else [y1, y2] -- x*, y* are SymInteger's
-- v2 = if b then [x2] else [y3]
-- v3 = v1 ++ v2
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 merging invariant in some cases. 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 do-blocks (binds) by normalizing it, and the code can be very verbose. See the following code as an example, and we will revisit this issue later.

```
normalize $ (normalize $ a >>= f) >>= g
```

4.2 Monad transformers, Error Handling and Stateful Computations

We've seen that we can model the execution with errors by nesting `Either` in a `Union` in Section 3.3. This works but forces us to extract values using pattern matching in `Union` do-blocks, resulting in rather verbose code. To reduce verbosity and improve modularity, monad transformers are another tool in monadic programming's arsenal, and using `ExceptT` can help reduce the boilerplates.

Before we can easily program with monad transformers, we first generalize and abstract the operations for `Union`-like data structures. `merge` function performs our merge and normalization algorithm. `unionIf` and `mrgIf` are branching functions, and `mrgIf` ensures that the results are merged. `single` and `mrgSingle` wrap a single value. We will discuss the rationale for having both `unionIf` and `mrgIf` and introducing `mrgSingle` later. The `Union` is clearly a `UnionLike` container, and applying `ExceptT` with a `Mergeable` error type to any `UnionLike` container is also `UnionLike`.

```
class UnionLike m where
  merge :: (Mergeable a) => m a -> m a
  unionIf :: SymBool -> u a -> u a -> u a
  single :: a -> u a
  mrgIf :: (Mergeable a) => SymBool -> u a -> u a -> u a
  mrgSingle :: (Mergeable a) => a -> u a
instance UnionLike Union where
  merge = normalize
  mrgIf = -- the hierarchical merging algorithm
  mrgSingle = Single
instance (Mergeable err, UnionLike u) => UnionLike (ExceptT err u) where
  merge = ExceptT . merge . runExceptT
  mrgIf cond (ExceptT l) (ExceptT r) = ExceptT $ mrgIf cond l r
  mrgSingle = ExceptT . mrgSingle . return
```

Then we can write generic code for `unionError`, and error handling program without boilerplates.

```
unionError :: (MonadError u, UnionLike u) => err -> SymBool -> u ()
unionError err c = mrgIf c (return ()) (throwError err)
assert = unionError Assert
assume = unionError Assume
mrgIf a (assert x1 >> return y1) (assume x2 >> return y2)
```

Supporting stateful symbolic computations is also easy by applying `StateT` to a `UnionLike` container.

```
instance (Mergeable s, UnionLike u) => UnionLike (StateT s u) where
  merge (StateT f) = StateT $ \s -> merge $ f s
  mrgIf cond (StateT l) (StateT r) = StateT $ \s -> mrgIf cond (l s) (r s)
  mrgSingle a = StateT $ \s -> mrgSingle (a, s)
```

4.3 Propagating Mergeable knowledge

In Section 4.1, we have shown that to use the Union monad efficiently, we need to call normalize (or merge defined in Section 4.2) for many times to maintain the merging invariant. This would not be a problem if Grisette were implemented as a language or a standalone tool and hide these details, but this will bother the user a lot if Grisette is a user-oriented library. It is better to reduce this boilerplate, making the results automatically merged, and less prone to inefficiency due to missing merge calls. To do this, we need a way to resolve Mergeable instances when intermediate results are constructed. This is related to the constrained monad problem [Sculthorpe et al. 2013], but the normalization approach proposed there does not work in our scenario because it would transform the entire computation to merge only once at the end of the computation, whose performance is no better than naive all-path symbolic evaluation without state merging. This incompatibility with merging semantics is innate and is not rare in monadic systems. The key is that we need to merge values rather than computations, and this rules out implementing our system with general CPS-based monads or most effect systems.

Instead, we adapt the knowledge propagation approach [Kiselyov 2013] to our system. This is not a perfect approach, can not eliminate all required merge calls, but can still save us from calling merge many times. The idea for the knowledge propagation approach comes from partial evaluation: use the statically available knowledge and propagate it. We define UnionM as follows. The UnionM is a GADT with two constructors: UMrg for the Unions that are already merged, and we know the Mergeable constraint. UAny is for the case where such knowledge is not available.

```
data UnionM a where
  UMrg :: Mergeable a => Union a -> UnionM a
  UAny :: Union a -> UnionM a
instance UnionLike UnionM where
  merge (UAny u) = UMrg (merge u)
  mrgSingle a = UMrg (Single a)
  unionIf cond (UMrg ul) (UAny ur) = UMrg (mrgIf cond ul ur) -- Mergeable instance from UMrg
  unionIf cond (UAny ul) (UAny ur) = UAny (unionIf cond ul ur) -- No Mergeable instance
```

The UnionLike instance shows how the knowledge propagation approach works. Those operations with Mergeable constraints would store the instance in UMrg constructor, and the unionIf operation, which does not have a Mergeable constraint, will try to resolve the Mergeable constraint from its operands. Then we can easily implement the Monad instance for UnionM with unionIf. The bind function for UnionM can build the underlying Union with mrgIf if any of the leaves is mapped to a UMrg and propagates the Mergeable knowledge. This ensures that if the last expression in a do-block is merged, the result will be merged. For example, in the following code, safeHead would throw an error monadically if the input list is empty. The result is merged and carries the Mergeable knowledge. This knowledge can propagate to the subsequent do-block to merge the result.

```
mrgThrowError :: (MonadError err m, MonadUnion m, Mergeable a) => err -> m a
mrgThrowError = merge . throwError
safeHead :: (Mergeable a) => [a] -> ExceptT Errors UnionM a
safeHead [] = mrgThrowError $ ...
safeHead (a:_) = mrgSingle $ a -- mrgSingle propagates the knowledge
j = do m <- mrgIf (return [a]) (return [b, c])
       safeHead m
```

```
  k = do j1 <- j ... -- Extracting values from j will not cause path explosion
  l = do ... ; j     -- using j as the last statement will cause the result to be merged
```

The `merge . throwError $ ...` pattern is common, so we provide it as a Grisette library function. We also provided the `mrg*` variants for other standard combinators, and most of them can be easily implemented by composing `merge` with the original version. If the user sticks to `mrg*` versions in the leaf computations or uses the original combinators wisely, the knowledge can be propagated throughout the whole computation, and they can easily get an efficient symbolic evaluator.

One thing worth noting for the approach is that `UnionM` is not strictly a monad:

```
  UMrg (Single a) >>= return -- UAny (Single a)
```

We do not consider this problematic because `UnionM` still complies with the monad laws under symbolic semantics. That is, the following formulas are valid, where ==~ is symbolic equality.

```
  (return a >>= h) ==~ h a                        -- left identity
  (m >>= return) ==~ m                            -- right identity
  ((m >>= g) >>= h) ==~ (m >>= (\x -> g x >>= h)) -- associativity
```

This can be easily proven, given that merging is a symbolic semantic-preserving transformation.

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?

5.1 RQ1: The Subjects

To show Grisette's expressiveness and performance, we ported several benchmarks from the state-of-the-art tool Rosette 4 [Porncharoenwase et al. 2022]. Rosette 4's benchmark set 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 1 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. 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 performance differences between Grisette and Rosette are caused by the Grisette tool, 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. Given that no previous merging algorithm has been designed for a typed library and that we cannot directly implement Rosette in Haskell, we believe that the performance comparison can still show the advantages of Grisette. We also implemented MEG-based `Union` and reimplemented the benchmarks with it. This approach shares all infrastructures with the core Grisette and can be used to compare the MEG and ORG approaches.

The benchmark programs are designed for very different tasks, which shows that Grisette is expressive enough to express a wide range of tasks that can be expressed using 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 the 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 to specify and verify the executable semantics of abstract stack-and-pointer machines that track the dynamic information flow to enforce security properties. Fluidics is a small DSL for synthesizing microfluidic array manipulation. Cosette is a SQL automated prover. It executes SQL queries symbolically and is intended to decide SQL query equivalence. Cosette heavily relies on Racket's macro system for staged computation. Because everything in the host language is available in Grisette, similar functionalities can be implemented with Template Haskell [Sheard and Jones 2002]. We only implemented part of the original system that are used in the Cosette benchmark, and removed the irrelevant lines from the Rosette versions, so the LoC metrics differ from the ones reported in the original Rosette 4 paper. We also discovered a typographical error in the original benchmark, and fixing it makes the Rosette version generate smaller formulas and run faster. Our performance results will be based on the fixed version. Bonsai is a data structure specialized in efficient synthesis-based reasoning for type systems. It allows more syntax tree merging and is capable of representing even larger search space 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, we also implemented the let-polymorphism benchmark from the original Bonsai paper, based on the repository of the paper.

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.

Table 1. Statistics of benchmarks

Name	Grisette LoC	Rosette 3 LoC	Rosette 4 LoC	Solver
Bonsai-NanoScala [Chandra and Bodik 2017]	663	439	439	Boolector
Bonsai-LetPoly [Chandra and Bodik 2017]	750	427	427	Boolector
Cosette [Chu et al. 2017]	661	532	532	Z3
Ferrite [Bornholt et al. 2016]	538	348	348	Z3
Fluidics [Willsey et al. 2019]	141	98	98	Z3
IFCL [Torlak and Bodik 2014]	653	483	483	Boolector
RegexCont	239	N/A	N/A	Z3
RegexFree	219	N/A	N/A	Z3

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 2.3.2. 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, people first design the concrete type and algorithms in mind and then make it symbolic. In Rosette, there is no boundary between concrete types and symbolic types, and everything can automatically become symbolic, so the user 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 algorithms. 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, bringing about a 75-fold speedup and making previously infeasible queries feasible.

The fixes are 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 Table = [(Data, Multiplicity)]
symFilter (\(rowdata, mult) -> pred rowdata) tbl :: UnionM Table              -- original
fmap (\(rowdata, mult) -> (rowdata, mrgIte (pred rowdata) mult 0)) tbl :: Table -- 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 rather than removing it. 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 things type check. This brings another 20x speedup.

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

After we implemented the optimized version in Grisette, which was just simple changes to make things type check, we backported the optimized algorithm to Rosette. Without a good IDE for Racket that performs type checking on the fly, 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, when we run the code, the errors would be caught by Rosette 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 execution trace, we believe that a better approach is having the ability to report bugs to the user during compilation.

5.3 RQ3: Running the benchmarks

To evaluate the performance of Grisette, we mostly followed the benchmark methods of Rosette 4 [Porncharoenwase et al. 2022]. We compared the running time and encoding size of Grisette, Grisette with MEG semantics, Rosette 4 and Rosette 3 for the ported and original version of 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 8.10.7 and Racket v8.1 on an AMD Ryzen Threadripper 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 2 shows the performance 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 for generating the encoding, lowering to SMT terms, and sending the terms to the solver, and is collected as the CPU time consumed by Haskell or Racket. For the Grisette benchmarks, the pure evaluation time (time to generate the encoding) is also reported, as our current implementation for lowering is very slow. This number will not be used to compare with Rosette, but 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 3 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 a percentage.

Table 2. Performance results for the ported and optimized benchmarks. T, E, PE, S, and Tm 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	2.3	.79	.44	1.5	12	8.3	2.1	.48	6.2	56	25	16	8.1	34	34	18	16	40
IFCL	34	4.7	1.6	30	91	39	10	3.0	29	222	197	14	183	383	148	14	134	438
Fluidics	11	2.7	.84	8.7	67	212	137	6.6	75	1371	22	8.4	14	284	28	8.8	20	308
Cosette	.27	.1	.064	.17	1.5	1.2	.54	.32	.63	8.4	16	7.9	7.7	114	12	8.1	3.9	128
NanoScala	1.6	1.2	.47	.41	21	5.4	4.5	1.7	.88	71	27	24	3.9	664	51	46	4.4	1222
LetPoly	127	17	11	110	102	1230	988	151	241	1725	1415	191	1224	4908	1297	142	1154	5152
Cosette-1	.023	.009	.006	.014	.077	.024	.01	.007	.014	.077	.24	.14	.11	.13	.26	.13	.12	.13
RegexCont	6.8	6.5	5.9	.25	16	74	68	23	6.0	311								
RegexFree	8.1	7.9	7.4	.24	16	76	69	25	7.7	318								

Table 2 and Table 3 show that Grisette always outperforms Rosette 3/4 in both time metrics and encoding size. On average, Grisette accelerates symbolic compilation over the state-of-the-art Rosette 4 tool by about 13.6x, generates approximately 91% smaller formulas and accelerates the solving of the resulting formula by about 8.2x. For some benchmarks, for example, Cosette, the running time and size of the formulas are reduced even more. 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 the 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 formulas, although they are slightly easier to solve. We noticed that the MEG semantics performs 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 make the MEG approach generate extra large disambiguation conditions.

Table 3. 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)	18.7x	1.0x	3.8x	59.2x	1.1x	6.3x	8.6x	1.0x	2.5x	4.9%	100.0%	20.1%
Rosette 3	57.0x	2.0x	10.5x	77.7x	2.9x	12.6x	44.7x	1.6x	7.9x	1.3%	60.6%	10.1%
Rosette 4	43.8x	2.5x	11.4x	80.0x	2.9x	13.6x	22.5x	2.3x	8.2x	1.1%	59.2%	8.5%

5.4 RQ4: Compare the default error encoding with the CBMC error encoding

To evaluate the 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 4 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 pick the fastest one.

Table 4. 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	34	4.7	1.6	30	91	72	4.6	1.6	67	91
NanoScala	1.6	1.2	.47	.41	21	1.6	1.1	.44	.51	20
LetPoly	127	17	11	110	102	69	16	11	54	91

5.5 RQ5: The effectness 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. In our current implementation, we are using the sbv [Erkok 2021] package as the solver interface. The package is quite high-level and does much more than what we need; thus, it 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 evaluation time does *not* contain the lowering and solver interaction times.

Table 5. Benchmarks for memoization

Benchmark	no memoization				with memoization				speedup ratio	
	Total	Eval	Lower	Solv	Total	Eval	Lower	Solv	Total	Eval
NanoScala	2.7	2.3	.65	.42	1.6	1.2	.68	.41	1.8x	2.0x
LetPoly	147	40	5.0	108	127	17	5.6	110	1.2x	2.4x
RegexCont	49	49	.45	.25	6.8	6.5	.61	.25	7.3x	7.5x
RegexFree	71	71	.41	.24	8.1	7.9	.54	.24	8.7x	8.9x

Table 5 shows that memoization is capable of improving the performance of symbolic evaluation, further accelerating symbolic evaluation by 2.0–8.9x and overall performance by 1.2–8.7x.

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 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. In program verification, the solvers are used to find a counterexample, while in program synthesis, the solvers are called to search for a program that meets some specification.

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. With symbolic host languages, the complex manipulation of constraints is hidden, and the user can get symbolic evaluators for free by implementing interpreters for new languages.

A reusable symbolic host language should be versatile enough to support different types of queries, while 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 into 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 formula 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, and showed that it was a spectrum and distinguished 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 only perform 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 has 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 ORG representations to represent the symbolic states and perform merging with rewrite rules. It does not aim at normalizing and merging all mergeable states.

7 CONCLUSION

In this paper, we have 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. The static types also allow the user to tune the performance more easily and confidently. The monadic abstraction provides a clean interface to symbolic evaluation, allowing 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.

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.

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.

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.

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

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

Levent Erkok. 2021. `sbv`. https://hackage.haskell.org/package/sbv

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, 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

William T. Hallahan, Anton Xue, and Ruzica Piskac. 2019b. 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.

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.

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.

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.

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.

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.

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.

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. 2021. *Use early return in rosette?* https://web.archive.org/web/20220223043310/https://github.com/emina/rosette/issues/201 achived 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.

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.

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