SYMBOLIC REASONING AS A LIBRARY

SIRUI LU

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington
2025

Reading Committee
Ras Bodik, Chair
Gilbert Bernstein
Zachary Tatlock

Program Authorized to Offer Degree:
Computer Science and Engineering

UNIVERSITY OF WASHINGTON

ABSTRACT

SYMBOLIC REASONING AS A LIBRARY

Sirui Lu

Chair of the Supervisory Committee:
Ras Bodik
Computer Science and Engineering

The increasing complexity of software systems demands robust methods for ensuring correctness and performance. As systems scale, traditional approaches are insufficient for uncovering subtle bugs or optimization opportunities, and automated reasoning has emerged as a critical but challenging solution. However, advances in constraint solvers have shifted the engineering burden to symbolic compilation systems that translate program semantics into efficiently solvable constraints.

This dissertation introduces Grisette, a symbolic compilation framework designed as a statically-typed, purely functional, monadic Haskell library for building domain-specific symbolic compilers. Its Ordered Guards (ORG) representation enables all-path symbolic evaluation while merging symbolic values into a compact normal form. This approach significantly reduces symbolic evaluation time (6.1× speedup), constraint size (79.2% reduction), and solving time (2.4× speedup) compared to traditional representations. Beyond performance, Grisette avoids complications of lifting host languages into the symbolic domain. Its functional design enables memoization, while its monadic interface handles computational effects. We evaluate Grisette on diverse benchmarks, including Rosette applications. Grisette's monadic design further enables symbolic reasoning for benchmarks beyond existing tools, like those with continuations and coroutines. It also serves as the core engine for TensorRight, an automated verification system for tensor graph rewrites.

To demonstrate Grisette's versatility, this dissertation also presents HieraSynth, a parallel framework for super-optimization built upon Grisette. Super-optimizers synthesize high-performance code but face a trade-off between program length (k) and instruction set size (n). HieraSynth introduces a decomposition strategy that adaptively, hierarchically partitions the search space along dimension n rather than k. It employs component-based synthesis with embedded instruction choices, translated into SMT constraints via Grisette, using parallel divide-and-conquer with efficient unrealizability pruning. A RISC-V Vector super-optimizer built with HieraSynth synthesizes programs of greater length ($k \approx 8$) for larger instruction sets ($n \approx 700$) than previously feasible, discovering programs that are both provably optimal under a cost model and empirically superior to human-designed code, while achieving substantial synthesis scalability.

ACKNOWLEDGMENTS

I have had the great privilege of being advised by Ras Bodik, whose enthusiasm, vision, and patience have made my PhD journey both fulfilling and transformative.

My journey with Ras began in 2019 when, as an undergraduate at Peking University, I nervously sent him a cold email about summer internship opportunities. His quick and welcoming response marked the beginning of a mentorship that would shape my academic career. That summer experience evolved into a year of remote collaboration, and when Ras offered me the opportunity to pursue my PhD at the University of Washington, I was thrilled to accept.

Throughout my time at UW, Ras has been an extraordinary mentor. His passion for research and ability to see the bigger picture constantly push me to explore ideas I might never have considered. He has taught me not only how to conduct rigorous research, but also how to effectively deliver it. Whether preparing talks or writing papers, his guidance on clear storytelling has been as valuable as his technical expertise. I hope these skills are reflected in this dissertation.

What sets Ras apart is his genuine investment in his students' growth as both researchers and people. He has devoted countless hours to guiding me with wisdom and patience, and his contagious enthusiasm and enduring support have sustained me through the challenges of graduate school. Through his example, I have learned what it means to be both a rigorous researcher and a supportive mentor. I could not have asked for a better advisor.

I am deeply grateful to the other members of my committee, Zachary Tatlock, Gilbert Bernstein, and Duane Storti, who have provided invaluable feedback that has significantly shaped this work. They have challenged me to think more deeply about my research and its applications across domains. Their thoughtful questions and engagement with my ideas have brought fresh perspectives that have strengthened this dissertation immeasurably.

The Allen School has been an exceptional place to pursue my PhD, and I feel fortunate to have been part of the PLSE group—a vibrant and supportive community. The group's culture of curiosity, mutual support, and genuine enthusiasm for research has been truly inspiring. From discussions to reading groups, every interaction has enriched my understanding of the field. Though I wasn't always physically present, the warmth and inclusiveness of the PLSE community always made me feel connected and valued.

I am forever grateful to my family and friends for their unwavering support and joy. So many thanks to my friends here in Seattle who have made this city home, and to those back home who have maintained our bonds despite the distance. Moving thousands of miles from home, I have been blessed with a loving family whose support has never wavered and friendships that have only grown stronger.

CONTENTS

1 INTRODUCTION

1.1 BACKGROUND ON AUTOMATED REASONING

As software systems become more complex and play increasingly critical roles, it is crucial to ensure their correctness and performance. Software defects in key systems, such as medical devices [38], aircraft [35], space exploration [23], power infrastructure [44], and operating systems [64] can lead to severe consequences. Meanwhile, modern applications in domains such as large-scale data processing and machine learning demand even higher performance and efficiency.

Traditionally, ensuring software correctness and performance has relied on code inspections, extensive testing, and performance profiling. While these approaches remain invaluable, they are often labor-intensive, time-consuming, and can overlook subtle errors or optimization opportunities. As software systems grow in complexity, we need techniques that can provide stronger guarantees with less human effort.

Automated reasoning provides this capability. By applying mathematical and logical reasoning to software systems, automated reasoning tools can check correctness, identify bugs, and optimize performance, often more exhaustively than manual approaches.

- *Automated verification* determines whether a system satisfies specific properties (e.g., functional correctness or safety guarantees) across all possible inputs. Unlike software testing, verification can cover every execution path, ensuring all paths are considered.
- *Automated synthesis* goes one step further by automatically constructing programs from high-level specifications, reducing the manual effort required to develop performance-critical code.

Building effective automated reasoning tools can be challenging. Thus, many systems adopt the approach of transforming or encoding programs and delegating the task to a reasoning engine. Examples of such reasoning engines include autodiff engines, integer linear programming (ILP) solvers, and satisfiability modulo theories (SMT) solvers. Advances in these reasoning engines, especially SMT solvers, have revolutionized the field. By

transforming many advanced analyses and reasoning tasks into constraint-solving problems [20–22, 34], the engineering burden has been shifted to the development of *symbolic compilation* systems. These systems encode high-level programs into efficiently solvable constraints. This paradigm shift has not only streamlined the development of reasoning tools but also significantly expanded the practical applicability of formal methods in software engineering.

1.2 BUILDING REASONING TOOLS WITH SMT SOLVERS

While SMT solvers have become increasingly powerful, constructing symbolic compilers can be tricky. It resembles conventional compiler or interpreter construction but generates logical formulas instead of machine code, requiring domain knowledge and formal methods expertise.

Consider verifying if two arithmetic expressions involving $+$, $*$, and $/$ always yield the same outcome or trigger the same errors. To verify that $x + x \equiv x * 2$, one can use a solver API like z3py [43]:

```
from z3 import *
x = Real("x")
lhs = x + x
rhs = x * 2
> solve(Not(lhs == rhs))
unsat # meaning no counterexample found, expressions are equivalent
```

The solver found no assignment to x where $\texttt{lhs} \neq \texttt{rhs}$, confirming equivalence. For $x/(x * x) \equiv 1/x$, a naive encoding `Not((x/(x*x) == 1/x))` will yield a *model* assigning 0 to x, suggesting that the expressions are not equivalent. This occurs because SMT solvers treat division-by-zero as yielding an unconstrained value. To correctly model division-by-zero semantics, one must explicitly add constraints:

```
lhs_div0 = x * x == 0
rhs_div0 = x == 0
# Checks if outcomes differ:
# one divides by zero and the other does not,
# OR neither divides by zero but their values are different.
> solve(Or(lhs_div0 != rhs_div0, And(Not(lhs_div0), lhs != rhs)))
unsat # The expressions are equivalent
```

This manual encoding quickly becomes tedious for more complex expressions or error conditions. Ideally, a tool should automate this translation. This motivates the use of *symbolic domain-specific language (SDSL)* [71] to abstract semantics and constraint generation.

1.3 AN SDSL FOR ARITHMETIC EXPRESSIONS

Instead of directly manipulating constraints, we define a DSL with arithmetic operations. A Python deep embedding might be:

```
class Expr:
  def __add__(self, other):
    if isinstance(other, int):
      other = Const(RealVal(other))
    return Add(self, other)
  def __radd__(self, other):
    return self.__add__(other)
  # ... other operations omitted for brevity
@dataclass
class Const(Expr):
  value: ExprRef # z3 expression
@dataclass
class Add(Expr):
  lhs: Expr
  rhs: Expr
# ... Mul and Div classes omitted for brevity
x = Const(Real("x"))
> x + x
Add(lhs=Const(value=x), rhs=Const(value=x))
```

An interpreter then translates SDSL expressions into SMT constraints, managing details like division-by-zero flags. The following code returns two values for each expression: the result value and a flag indicating if the expression triggers division-by-zero.

```
def interpret(ex: Expr):
  match ex:
    case Const(value):
      return (value, BoolVal(False))
    case Add(lhs, rhs):
      lhs_value, lhs_div0 = interpret(lhs)
      rhs_value, rhs_div0 = interpret(rhs)
      return (lhs_value + rhs_value, Or(lhs_div0, rhs_div0))
    # ... Mul case omitted
    case Div(lhs, rhs):
      lhs_value, lhs_div0 = interpret(lhs)
      rhs_value, rhs_div0 = interpret(rhs)
      return (
        lhs_value / rhs_value,
        # Sub-expression division-by-zero flags need to be propagated
        Or(Or(lhs_div0, rhs_div0), rhs_value == 0)
      )
```

We combine these constraints in a `verify` function to check equivalence:

```
def verify(a: Expr, b: Expr):
  a_value, a_div0 = interpret(a)
  b_value, b_div0 = interpret(b)
  return solve(
    Or(a_div0 != b_div0, And(Not(a_div0), a_value != b_value))
  )

x = Const(Real("x"))
> verify(x + x, x * 2)
unsat # This implies that x + x == x * 2 is verified
> verify(x / (x * x), 1 / x)
unsat # This implies that x / (x * x) == 1 / x is verified
```

While tidier than raw SMT API calls for each expression, this approach still requires manual tracking of error flags (here, `div0`). For DSLs with more complex error types or control flow, this manual constraint management becomes increasingly burdensome.

1.4 SYMBOLIC HOST LANGUAGES

Symbolic host languages [49, 60, 62, 71, 72] address the tedious problem of manual constraint manipulation. They allow users to write interpreters focusing on the core logic, while the symbolic engine transparently handles path conditions and exceptions.

This dissertation introduces GRISETTE, a symbolic evaluation engine implemented as a Haskell library. An arithmetic expression interpreter in GRISETTE looks like this:

```
data Expr
  = Const SymAlgReal -- Algebraic real numbers for symbolic values
  | Add Expr Expr
  | Mul Expr Expr
  | Div Expr Expr

interpret :: Expr → ExceptT ArithException Union SymAlgReal
-- mrgReturn, (.<$>), and (.<*>) are Grisette's variants for
-- return, (<$>), and (<*>)
interpret (Const n) = mrgReturn n
interpret (Add e1 e2) = (+) .<$> interpret e1 .<*> interpret e2
interpret (Mul e1 e2) = (*) .<$> interpret e1 .<*> interpret e2
interpret (Div e1 e2) = do
  n1 ← interpret e1
  n2 ← interpret e2
  safeFdiv n1 n2 -- Division of real numbers, with exception handling
```

```
verify :: Expr → Expr → IO ()
verify a b = do
  result ← solve (interpret a .== interpret b)
  case result of
    Left Unsat → putStrLn "Expressions are equivalent"
    Right _model → putStrLn "Expressions are not equivalent"
    _ → putStrLn "Solver error"
```

This solution is clean and concise. The "magic" happens during symbolic evaluation, where the interpreter explores multiple execution paths simultaneously while maintaining the path conditions. The exceptional semantics, including the propagation of sub-expression errors, is handled by the `ExceptT` monad transformer.

Let's examine the `safeFdiv` function. In a standard interpreter, division-by-zero would raise an exception and halt the execution. In symbolic execution, however, we must consider both possibilities when dividing by a symbolic value:

```
safeFdiv ::
  SymAlgReal → SymAlgReal → ExceptT ArithException Union SymAlgReal
safeFdiv n d =
  mrgIf
    (d .== 0) -- Symbolic equality check
    (throwError RatioZeroDenominator)
     -- Solver's division (unconstrained on division-by-zero)
    (return (n / d))
```

When `safeFdiv` encounters a potentially symbolic denominator, it uses `mrgIf` to create a conditional branch considering both outcomes. If the denominator is symbolically zero (via `.==`), it throws `RatioZeroDenominator` in the `ExceptT`-transformed `Union` monad; otherwise, it performs the division using `(/)` (the SMT solver's division function, which yields an unconstrained result on division-by-zero).

Crucially, with the `Union` monad and the `mrgIf` combinator, Grisette maintains both possibilities along with their respective path conditions:

```
> safeFdiv "a" "b"
ExceptT {
  If (= b 0.0) (Left RatioZeroDenominator) (Right (fdiv a b))
}
```

This handling of division highlights a key strength of Grisette. The monadic interface makes error handling appear natural. To the programmer, writing `safeFdiv` is just like writing regular error-handling code. Conceptually, the `ExceptT ArithException Union` monad combines exception

tracking with path condition management, exploring all possible execution paths. This monad can then be composed using standard `do`-notation, and the path conditions are automatically managed.

When `solve` is called, all paths, including division-by-zero cases, are considered. This eliminates the need to manually encode these conditions as separate constraints, which is tedious and error-prone with raw solver APIs.

By providing this high-level abstraction, GRISETTE significantly simplifies building symbolic DSLs and reasoning tools.

1.5 THESIS STATEMENT

The effectiveness of symbolic host languages for diverse program reasoning tasks rests on the combination of three techniques: *all-path symbolic execution*, *partial evaluation*, and *state merging*.

- *All-path symbolic execution* is essential because many verification and synthesis applications require analyzing all possible execution paths. A symbolic host language must translate all these paths into constraints to enable holistic reasoning.
- *Partial evaluation* is important because real-world, expressive host languages often have advanced constructs that constraint solvers do not directly support. During symbolic evaluation, these constructs can be removed or simplified by partial evaluation. This allows users to benefit from an expressive host language without needing to encode everything into constraints, which greatly improves the symbolic engine's usability.
- *State merging* mitigates the inherent path explosion problem of all-path symbolic execution. By merging execution paths, the system can efficiently handle the exponential number of potential paths while keeping constraints manageable.

This dissertation demonstrates how building *symbolic compilers as libraries* can lower the barrier to constructing specialized reasoning tools. My thesis statement is as follows:

> **Building a symbolic compiler as a typed, functional, monadic library can facilitate the construction of reusable, extensible, and efficient automated reasoning tools.**

Although GRISETTE is not the first versatile system to offer the three core techniques, its library-based philosophy stands in contrast to two other

paradigms. The first is the *standalone tool* approach, shown in systems like CBMC [19] for C or SKETCH [60, 62] with its own language. These tools are powerful for their specific targets but can be difficult to extend or integrate with other language ecosystems. The second is the *language-lifting* approach, pioneered by ROSETTE [49, 71, 72], which deeply integrates symbolic capabilities into a host language's runtime to add symbolic features along with the original concrete semantics. While powerful, this approach faces challenges: they often limit the lifted features to a subset of the host language, since translating arbitrary language constructs is difficult. Moreover, the symbolic semantics can subtly diverge from their concrete counterparts, causing confusion [67]. Such deep integrations are also not easily portable to other host languages. The library-based approach proposed in this dissertation aims to find an approach that offers portability and a clear semantic contract at the library API level, often enforced by the host language's type system.

I identify the following open questions in previous systems:

1. *How can we more easily support rich programming features?* Partial evaluation simplifies support for many features, but certain features, like exception handling and stateful computations, still require special handling in the symbolic engine. Users typically cannot extend a system with these capabilities without modifying the symbolic engine itself, if they are not provided. For example, ROSETTE lacks native support for reasoning about continuations; users wishing to use continuations in symbolic evaluation must implement a non-trivial interpreter [68].

2. *How can we ensure safe usage of external libraries?* Although partial evaluation allows calling arbitrary libraries without requiring their source code, prior systems may cause confusion or lead to undefined behavior if symbolic values are used in unsupported contexts [68].

3. *How can we handle flexible error types without performance degradation?* Previous systems often trade off expressiveness and performance in symbolic evaluation. Supporting flexible error types, like distinguishing assertions and assumptions, or supporting even more complex error types, is often non-trivial and requires significant changes to the symbolic engine. Such changes sometimes also lead to larger formulas and may thus slow down the solving process [49]. Our goal is to provide richer error-handling capabilities than earlier systems while retaining performance when the advanced features are unused, i.e., to provide abstraction without overhead.

4. *How can we further improve symbolic evaluation performance?* Evaluating all paths is inherently expensive. While techniques like memoization and parallelization exist, they are challenging to implement in systems relying on global states for optimization.

This dissertation argues that building symbolic compilers as *statically-typed, functional, monadic libraries* can address these open questions coherently. To substantiate this thesis, the dissertation is structured around two major, interconnected contributions.

First, it presents Grisette, a novel symbolic compilation library designed from the ground up to realize this philosophy. Constructing such a library in a purely functional setting is not trivial and poses significant challenges, particularly in representing and merging symbolic values. In a functional environment, information like exceptions or mutable states is not stored globally but propagated with return values. This can make the values more complex and harder to merge, potentially leading to larger formulas and slower solving with previous merging algorithms. Grisette's design, particularly its novel ORG (Ordered Guards) representation for symbolic unions and its type-driven merging, is tailored for this scenario and leads to more compact SMT formulas and improved performance. With these challenges solved, Grisette provides a reusable symbolic compilation framework that forms the foundation for specialized program reasoning tasks. I have successfully ported several Rosette tasks to Grisette and collaborated on building TensorRight [4], a tensor graph rewrite verifier built upon it.

Second, to demonstrate the performance and extensibility of this library-based approach, the dissertation presents HieraSynth, a framework for constructing super-optimizers, built upon the Grisette foundation. Super-optimization seeks to find an instruction sequence with the best possible performance for a given task, measured by a formal cost model. Such approaches face a long-standing scalability trade-off between synthesizable program length (k) and instruction set size (n). To address this challenge, HieraSynth introduces a novel *adaptive, hierarchical* parallel divide-and-conquer strategy that decomposes the search space along the instruction set size dimension (n) rather than program length (k). This approach preserves completeness, allowing HieraSynth to exhaustively search vast program space and find programs that are provably optimal with respect to a formal cost model. The synthesis of each subproblem is powered by Grisette's symbolic reasoning engine. As a framework built upon Grisette, HieraSynth inherits its design principles of safety and modularity, making it a highly reusable library for targeting different instruction set architectures (ISAs). To validate this, I have implemented a super-optimizer for the complex RISC-V Vector extension, which not only demonstrates superior

efficiency but also discovers novel, high-performance optimizations. This result serves as a compelling testament to the performance and extensibility of the underlying library-based design.

1.6 DISSERTATION ORGANIZATION

This dissertation is organized as follows:

- Chapter 2 overviews GRISETTE and its programming model.
- Chapter 3 discusses the design principles for a safe, predictable, modular, and extensible symbolic engine.
- Chapter 4 describes the ORG representation for symbolic unions and its benefits.
- Chapter 5 presents our experience using GRISETTE to build TENSORRIGHT.
- Chapter 6 details the HIERASYNTH super-optimization framework.
- Chapter 7 concludes the dissertation.

2 GRISETTE SYSTEM OVERVIEW

This dissertation explores the development of automated reasoning tools for programs via symbolic compilation. I argue that building a foundation for symbolic compilers as a typed, functional library can facilitate the construction of reusable automated reasoning tools. To this end, I present the Grisette system that realizes these design principles. Grisette aims to improve on existing approaches by leveraging static typing for safer usage; functional, monadic programming for modularity; and a library-based approach for reusability. Grisette also introduces a novel symbolic value representation with type-directed merging that generates better constraints, building upon the symbolic union concept pioneered by Rosette [72] and MultiSE [56].

This chapter serves as an introduction to Grisette's core concepts and programming model. We will start by demonstrating basic symbolic programming with constraints; explore how Grisette handles multiple execution paths using symbolic unions; illustrate the construction of symbolic domain-specific languages (SDSLs); and finally show how monadic programming helps manage effects like exceptions during symbolic evaluation. This overview will establish the foundation for understanding Grisette's capabilities and show how to build reasoning tools using it.

2.1 SYMBOLIC PROGRAMMING WITH CONSTRAINTS

Traditionally, automated reasoning was achieved by tools that act as symbolic compilers or interpreters for a specific language, such as a subset of C or Java. These tools translate program semantics into logical constraints for solvers. They might analyze specific program paths, like symbolic execution engines [14, 15, 30, 55], or attempt to cover all paths up to a certain bound, like bounded model checkers [19]. These tools often focus on specific program properties or programming tasks, such as program synthesis [60, 62] or analysis [79].

More recently, a powerful and versatile approach emerged: embedding *symbolic programming* capabilities within a general-purpose *symbolic host language* [71]. Symbolic host languages allow users to implement interpreters for *symbolic domain-specific languages (SDSLs)* using the host language's symbolic programming facilities. The result of the interpretation is symbolic

constraints, similar to those generated by traditional symbolic compilers, making the interpreter a *symbolic evaluator* for the SDSL. Symbolic host languages allow developers to leverage the full power of the host language when constructing symbolic evaluators. This includes its type system, abstraction capabilities, libraries, or even features not directly compilable to constraints—such as complex data structures or higher-order functions—which can be partially evaluated during symbolic execution. This greatly reduces the effort required to build symbolic evaluators and has enabled various program reasoning applications, like JITTERBUG [46] for verifying just-in-time compilers, SERVAL [45] for developing automatic verifiers for system software, COSETTE [18] for reasoning about SQL equivalences, and FERRITE [10] for specifying and checking file system crash-consistency models. ROSETTE [49, 71, 72], built on Racket, is a pioneering example of this approach and provides a user-friendly interface for constructing symbolic evaluators.

While GRISETTE also builds on the idea of building symbolic evaluators within a host language, it aims to address several challenges. First, GRISETTE adopts a statically-typed (in Haskell), purely functional, and library-based approach to provide a modular, safe, and reusable foundation for symbolic programming. Second, it introduces a novel symbolic union type with configurable, flexible type-directed merging which generates more compact constraints. We will analyze the impact of these design choices on safety, modularity, and performance in Chapter 3 and Chapter 4.

Let's now begin exploring GRISETTE's programming model by introducing basic symbolic programming concepts. Traditional programming works with known, concrete values. Symbolic programming extends this model to work with unknown values. Following ROSETTE's terminology, we call values that can contain unknown parts *symbolic values*.

The simplest form of a symbolic value is a *symbolic constant*, which is a typed symbol or, in other words, a placeholder for a concrete value that is not known during program execution and will be determined by a constraint solver later. This is analogous to using variables like x or y to represent unknown values in an algebraic equation. In GRISETTE, we utilize Haskell's `OverloadedStrings` language extension to declare symbolic constants directly from string literals.[1]

From these symbolic constants, we build *symbolic expressions* by applying operations. These operations can combine symbolic values with each other (e.g., `a * b`) or with concrete constants (e.g., `a * 2`). The outcome is not a concrete number but rather a symbolic expression representing the com-

1 Using the same string literal with the same type refers to the identical symbolic constant.

putation. The following example shows declaring constants `a` and `b` and creating an expression `aTimesB` representing their product.

```
a, b :: SymInteger
a = "a"
b = "b"

aTimesB :: SymInteger
aTimesB = a * b

> aTimesB
(* a b) -- The symbolic expression, printed in S-expression form.
```

Here, the type `SymInteger` indicates that `a`, `b`, and `aTimesB` are symbolic integer values. `a` and `b`, being created directly from string literals, are specifically *symbolic constants*. Grisette provides various primitive symbolic types like `SymInteger`, `SymBool`, and `SymWordN` (symbolic unsigned bit-vectors), etc., corresponding to common solver theories.

Symbolic Boolean values (`SymBool`) represent symbolic logical conditions, and they can be used as *symbolic constraints*. Just as Grisette provides symbolic arithmetic operators, it offers operators to build these constraints. For example, the symbolic equality operator `.==` yields a `SymBool` representing the equality condition, unlike the standard `==` which yields a concrete `Bool`. The following example constructs a constraint asserting that the product `aTimesB` must equal `12`.

```
-- Symbolic equality is provided by the .== operator.
eqConstraint :: SymBool
eqConstraint = aTimesB .== 12

> eqConstraint
(== (* a b) 12)
```

Using an SMT solver, such as Z3 [43], we can attempt to solve this constraint. If a solution exists, the solver returns a *model* satisfying the constraint (a model is a mapping from symbolic constants to concrete values). Here, one possible model maps `a` to `3` and `b` to `4`, satisfying `aTimesB .== 12`.

```
> solve z3 eqConstraint
Right (Model {a → 3, b → 4}) -- One possible solution model
```

Not all constraints are satisfiable. For example, no two integer values greater than `1` can have a product of `11`. We can construct a constraint stating this and prove it unsatisfiable with an SMT solver:

```
-- .>, .&&, and .== are symbolic operators provided by Grisette.
> solve z3 $ a .> 1 .&& b .> 1 .&& aTimesB .== 11
```

```
Left Unsat
```

A key strength of the symbolic host language approach is that symbolic values are just data structures, allowing seamless integration of symbolic computation with the host language's programming model. In Grisette, defining a function that operates on symbolic values uses standard Haskell syntax, and no special declaration is needed to help the solver understand symbolic computation. What we need to do is simply use operators and types supported by Grisette (like `SymInteger`, `+`, $*$), and we can then use solvers to reason about the function's behavior:

```
-- Takes three symbolic integers and returns a symbolic integer.
axpy :: SymInteger → SymInteger → SymInteger → SymInteger
axpy a x y = a * x + y

-- Evaluate the function with symbolic inputs.
> axpy 2 "x" 4
(+ 4 (* 2 x))

-- Solve for 'x' satisfying axpy(2, x, 4) == 10
> solve z3 $ axpy 2 "x" 4 .== 10
Right (Model {x → 3 :: Integer})

-- Prove unsatisfiability of axpy(2, x, 4) == 11
> solve z3 $ axpy 2 "x" 4 .== 11
Left Unsat
```

Here, the solver tells us that $x = 3$ is a solution to the equation $2x + 4 = 10$, while there is no integer solution to the equation $2x + 4 = 11$.

In summary, Grisette allows writing standard Haskell code that operates on symbolic values. When this code executes along a single path, the result is a symbolic expression representing the computation, which captures the computation's dependence on symbolic inputs. The next section explores how Grisette handles scenarios involving multiple execution paths due to symbolic conditions.

2.2 MULTI-PATH SYMBOLIC EVALUATION

The `axpy` example involved only straight-line code, resulting in a single execution path regardless of whether inputs are concrete or symbolic. However, programs often involve conditional logic (`if` statements, etc.). When writing explicitly symbolic programs, or when analyzing concrete programs with symbolic inputs, it is possible that the path taken depends on a symbolic value, and cannot be determined at the symbolic evaluation time. There-

fore, to reason about all possible behaviors of the program, the symbolic evaluator must effectively explore and represent *all feasible execution paths*.

Traditionally, symbolic evaluators handle this by enumerating all possible paths [36]. For example, consider the C program in Listing 1. Can the assertion be violated for some integer inputs `a` and `b`?

Listing 1: C program with conditional branches

```
void f(int a, int b) {
  int x = 1, y = 0;
  if (a != 0) {           // Condition C1
    y = x + 1;
  }
  if (b == 0 && a != 1) { // Condition C2
    x = 2 * (a + b);
  }
  assert(x != y);
}
```

To analyze this program by treating `a` and `b` as symbolic integers, a symbolic evaluator must explore the combinations of paths resulting from the two `if` statements. There are four potential scenarios (paths) based on the conditions $C_1 : (a \neq 0)$ and $C_2 : (b = 0 \wedge a \neq 1)$:

1. **Path 1:** (C_1 true, C_2 true): Requires $a \neq 0 \wedge b = 0 \wedge a \neq 1$. State before assert: $x = 2a, y = 2$. Assertion $2a \neq 2$ holds since $a \neq 1$. No violation.

2. **Path 2:** (C_1 true, C_2 false): Requires $a \neq 0 \wedge (b \neq 0 \vee a = 1)$. State before assert: $x = 1, y = 2$. Assertion $1 \neq 2$ holds. No violation.

3. **Path 3:** (C_1 false, C_2 true): Requires $a = 0 \wedge (b = 0 \wedge a \neq 1)$, which simplifies to $a = 0 \wedge b = 0$. State before assert: $x = 0, y = 0$. Assertion $0 \neq 0$ is **false**. Violation found for input $a = 0, b = 0$.

4. **Path 4:** (C_1 false, C_2 false): Requires $a = 0 \wedge (b \neq 0 \vee a = 1)$, which simplifies to $a = 0 \wedge b \neq 0$. State before assert: $x = 1, y = 0$. Assertion $1 \neq 0$ holds. No violation.

This analysis reveals that the assertion is violated if and only if the input is $a = 0, b = 0$.

This example illustrates that even simple programs generate multiple execution paths under symbolic inputs. As the number of conditional branches increases, the total number of paths can grow exponentially (e.g., 2^n paths for n independent binary `if` statements). This leads to the well-known *path*

explosion [16] problem, a fundamental challenge that practical symbolic evaluation tools must address to remain feasible for analyzing non-trivial programs.

2.2.1 *Merging Paths with `ite` for Primitive Types*

One strategy, suitable for simple data types directly supported by SMT solvers like Booleans (`SymBool`) or integers (`SymInteger`), is to encode the branching logic into a single symbolic expression using the solver's native `ite` (if-then-else) construct. An SMT `(ite c t f)` expression symbolically represents the choice: it evaluates to `t` if condition `c` is true, and `f` otherwise. This strategy is employed effectively in tools like bounded model checkers, for example, CBMC [19].

GRISETTE provides the `symIte` function to access the native `ite` construct. It allows merging the outcomes of both branches into a single symbolic value, collapsing the two potential paths into one formula without explicitly exploring each path.

```
maxInt :: SymInteger → SymInteger → SymInteger
maxInt x y = symIte (x .> y) x y

> maxInt "a" "b"
-- A single symbolic integer representing the merged result
(ite (< b a) a b)
```

Using this technique, we can conceptually transform the original program into a symbolic Static Single Assignment (SSA)-like representation, where each variable update incorporates the conditional logic using `ite`:

```
-- Original program
void f_ssa(symbolic int a, symbolic int b) {
  symbolic int x0 = 1, y0 = 0;
  symbolic int y1 = ite(a != 0, x0 + 1, y0);
  symbolic int x1 = ite(b == 0 && a != 1, 2 * (a + b), x0);
  assert(x1 != y1);
}
```

Evaluating this SSA form does not involve exploring an exponential number of program paths. The assertion condition becomes a single, potentially complex, symbolic Boolean formula:

```
assert(ite(b == 0 && a != 1, 2 * (a + b), 1) != ite(a != 0, 1 + 1, 0));
```

It is worth noting that while printing such a monolithic formula might suggest exponential growth if sub-expressions are naively duplicated, symbolic evaluation engines typically represent these formulas internally as

Directed Acyclic Graphs (DAGs). Shared sub-expressions are represented only once in memory, with multiple parts of the formula pointing to them. This DAG representation ensures that the actual size of the symbolic formula does not explode exponentially, even if its printed string representation does.

Solving for inputs making this formula false using an SMT solver reveals the violation at $a = 0, b = 0$. This demonstrates how `ite` avoids path explosion for programs operating solely on primitive types.

2.2.2 *Handling Complex Types with Symbolic Unions*

The `ite` construct works well for primitive types directly supported by SMT solvers. By merging branches into a single symbolic expression, it effectively avoids the path explosion problem for those types. However, extending this approach to handle complex user-defined data structures (like algebraic data types, lists, trees) becomes significantly more challenging.

Supporting complex types via direct SMT encoding requires the ability to translate all operations and intermediate values associated with these types into corresponding SMT logic. This can involve "heroic engineering effort" [72] to develop the necessary translations for every data structure and function used. Furthermore, it often requires access to the source code of libraries to translate their operations symbolically. Tools like CBMC [19] adopt this approach. While powerful, building and maintaining such translators requires significant effort over years.

Consider evaluating the Haskell-like code snippet in Listing 2 symbolically using only the `ite` approach:

Listing 2: Merging lists under symbolic conditions

```
-- "a", "b", "x", "y", "z" are symbolic Booleans
list = if "a" then ["x"] else if "b" then ["y", "x"] else ["y", "z"]
result = head list
```

To handle this purely with SMT encoding, we would need SMT-level representations for lists, potentially with *symbolic lengths*, and *symbolic operations* like `head`, along with a way to express the conditional construction of lists within the SMT logic (an `ite` for lists). While SMT theories for algebraic data types exist and might handle simple list structures with its built-in `ite` support, this approach faces some significant limitations. First, support for ADT theories is not universal across all SMT solvers, limiting solver choices. Furthermore, the approach does not readily scale to more complex user-defined data structures or arbitrary Haskell library functions operating on them, as the engineering effort required to create the necessary

SMT encodings remains substantial. Critically, representing data structures purely as SMT terms means sacrificing the rich features provided by the host language; for example, Haskell's powerful pattern matching would no longer be directly applicable to these SMT-encoded lists. Consequently, from the user's perspective, interacting with complex data solely through their low-level SMT representation can become cumbersome, potentially offering little advantage over programming directly with raw SMT terms via a solver's API. This diminishes the abstraction and convenience offered by the host language.

In contrast, an alternative strategy is *explicit path enumeration*, as illustrated with the C programming language example at the beginning of Section 2.2, where we explicitly enumerated the four paths. This approach only needs to encode the *path conditions* (the sequence of constraints derived from branching decisions) as symbolic Boolean formulas. Crucially, it enables *partial evaluation*: operations on complex data structures within a specific path can be executed concretely by the host language's runtime during symbolic evaluation. In the list example (Listing 2), along the path where `a` is true, the list value is simply the concrete structure `[x]`. Along the path where `a` is false and `b` is true, the list value is `[y, x]`. These are concrete list structures, although their elements (`y` and `x`) might themselves be symbolic values. We would not need to encode lists with symbolic sizes or the `head` operation into the SMT formula itself, as `head` could be evaluated concretely by the host language for each path's specific list value. This significantly simplifies handling arbitrary data types but, as discussed, suffers directly from the *path explosion* problem.

This presents a dilemma: direct SMT encoding via `ite` avoids path explosion but requires heroic efforts to encode complex types, while explicit path enumeration handles complex types easily via partial evaluation but suffers from path explosion. Is there a middle ground?

Luckily, the answer is yes, as evidenced by systems like Rosette [71, 72] and MultiSE [56]. These systems introduced the concept of *symbolic unions* (also known as *value summaries* in MultiSE). A symbolic union represents multiple potential outcomes from different execution paths within a single data structure. Each outcome is paired with its corresponding *path condition*. Crucially, these outcomes can be complex data structures that are *partially concrete*, meaning their high-level structure (like being a list of a certain length) is concrete, allowing standard host-language operations, even if their elements might still be symbolic values.

This structure enables a powerful, data structure-aware merging strategy: merge outcomes with the same concrete structure as much as possible, while keeping structurally different outcomes separate. For example,

when dealing with lists resulting from different paths, a union-based system might group lists of the same length together, merging them element-wise (potentially using `ite` for the elements), but keep lists of different lengths as distinct entries in the union. This preserves the structural information needed for partial evaluation while still collapsing many redundant paths.

Let's revisit the list example in Listing 2 to see how a symbolic union might handle it:

There are 3 paths in this example:

- Path 1: Condition `a`, Result `[x]` (Length 1)
- Path 2: Condition `(&& (! a) b)`, Result `[y, x]` (Length 2)
- Path 3: Condition `(&& (! a) (! b))`, Result `[y, z]` (Length 2)

A symbolic union evaluating this could merge the two branches that produce lists of length 2 (Paths 2 and 3). For these two paths, the first element `y` is common; the second elements `x` and `z` differ and are merged using `ite` under their respective sub-conditions (`b` vs `(! b)`, given `(! a)`). The conceptual state of the union for `list` becomes:

```
listUnion = {
  Guard: a,       Outcome: [x],              -- Length 1 entry
  Guard: (! a),   Outcome: [y, (ite b x z)]  -- Merged Length 2 entry
}
```

Now, consider evaluating `head` on the merged list. Because the symbolic union preserves the concrete list structure within each entry, we can *distribute* the standard `head` operation to each outcome inside the union via partial evaluation:

```
-- distribute the head operation over the union elements
head listUnion = {
  Guard: a,     Outcome: head [x],             -- Length 1 entry
  Guard: (! a), Outcome: head [y, (ite b x z)] -- Merged Length 2 entry
}
⇒ -- partial evaluation
{
  Guard: a,     Outcome: x,
  Guard: (! a), Outcome: y,
}
⇒ -- merge the two entries using ite for primitive types
(ite a x y)
```

Note that no special symbolic encoding for `head` or lists was required; the standard list operation was directly applied thanks to the partially concrete values stored in the symbolic union.

Symbolic unions thus combine the benefits of both approaches:

- Like `ite`, they avoid explicitly exploring all paths by merging some outcomes.
- Like explicit enumeration, they store structurally concrete values from different branches, thereby supporting partial evaluation naturally.

Grisette implements the symbolic union with its generic `Union a` type constructor. A value of type `Union a` holds multiple potential outcomes of type `a`, each associated with its *path condition*. One of Grisette's key contributions, discussed later in Chapter 4, is its novel and efficient internal representation (Ordered Guards, ORG) for symbolic unions that facilitates efficient merging and aims to generate symbolic formulas that are easier for SMT solvers to handle.

The combination of path merging and partial evaluation makes symbolic unions a powerful technique for handling complex data structures and mitigating path explosion in symbolic evaluation. Now, let's examine a more concrete example to see how Grisette helps achieve this in the context of a user-defined algebraic data type.

2.2.3 *Example: Symbolic Access Control*

To illustrate the challenges of multi-path evaluation and demonstrate how symbolic unions can be applied in practice, let's consider an example access management system. In this system, we have *primitive resources*, each associated with one of three access levels, ordered from most restrictive to most permissive: `Denied`, `ReadOnly`, and `ReadWrite`. A *composite resource* aggregates several primitive resources, and its overall access level is determined by the minimum (i.e., most restrictive) permission granted for any of its constituent primitive resources.

Access decisions depend on the attributes of an incoming request. For our example, we consider four attributes:

- Clearance level (an integer)
- Number of currently authenticated sessions for the user (an integer)
- Multi-factor authentication (MFA) status (a Boolean)

- Network origin (a Boolean indicating if the request is from a public network)

Each attribute can affect the access granted by multiple primitive resources, creating non-trivial dependencies in the overall access logic.

First, we will show a concrete (non-symbolic) system implementation that computes the composite access level based on given request attributes in Section 2.2.3.1. Then, we will solve the inverse problem (Section 2.2.3.2) via symbolic reasoning: given a desired (or perhaps forbidden) composite access level, can we find request attributes that produce it? Finding such attributes could reveal potential security bugs in the access policy. We will demonstrate both symbolic solutions with explicit path enumeration (Section 2.2.3.3) and symbolic union (Section 2.2.3.4) and show that the symbolic union approach does not suffer from the path explosion problem. We will use Grisette code and its design of symbolic union as an example—the benefits of the specific ORG design will be discussed later, in Chapter 4.

2.2.3.1 *Concrete Implementation*

Let's model this system concretely in Haskell, as shown in Listing 3. We define the `Access` levels and a `RequestAttributes` structure holding the four attributes. Functions `primAccess1` through `primAccess3` represent the policies determining the access level for three primitive resources based on these attributes. The logic is intentionally contrived to demonstrate coupled dependencies.[2] The `composeAccessList` function calculates the overall access level by folding the `min` operation over the primitive resource accesses, using `ReadWrite` (the most permissive access level) as the identity element for the fold.

Listing 3: Concrete access control

```haskell
-- The automatically derived Ord instance orders
-- Denied < ReadOnly < ReadWrite.
-- Thus 'min access1 access2' computes the most restrictive level.
data Access = Denied | ReadOnly | ReadWrite
  deriving (Show, Ord, Eq)

-- Request attributes.
data RequestAttributes = RequestAttributes
  { clearanceLevel :: Integer,
    authorizedSessionCount :: Integer,
    mfaVerified :: Bool,
```

2 This example uses somewhat arbitrary logic to demonstrate dependencies; real systems might have more complex rules distributed across multiple functions or modules.

```
    onPublicNetwork :: Bool
  }

-- Primitive Resource 1:
-- * Deny if the clearance level is under 5.
-- * Restrict to ReadOnly if too many simultaneous sessions with
--   high clearance level to defend against credential sharing.
-- * Otherwise, grant full access.
primAccess1 :: RequestAttributes → Access
primAccess1 RequestAttributes {..} =
  if clearanceLevel < 5
    then Denied
    else
      if clearanceLevel + authorizedSessionCount > 9
        then ReadOnly
        else ReadWrite

-- Primitive Resource 2:
-- * Deny if the clearance level is under 7.
-- * Grant full access if verified Multi-Factor Authentication (MFA).
-- * Otherwise, restrict to ReadOnly.
primAccess2 :: RequestAttributes → Access
primAccess2 RequestAttributes {..} =
  if clearanceLevel < 7
    then Denied
    else if mfaVerified then ReadWrite else ReadOnly

-- Primitive Resource 3:
-- * Deny if no authorized sessions.
-- * Restrict to ReadOnly if on public network.
-- * Otherwise, grant full access.
primAccess3 :: RequestAttributes → Access
primAccess3 RequestAttributes {..} =
  if authorizedSessionCount < 1
    then Denied
    else if onPublicNetwork then ReadOnly else ReadWrite

-- Combine two access levels. 'min' returns the more restrictive one.
composeAccess :: Access → Access → Access
composeAccess = min

-- Compute composite access for a list of primitive accesses.
-- Start fold with ReadWrite (identity 'min' over Access levels).
composeAccessList :: [Access] → Access
composeAccessList = foldr composeAccess ReadWrite

```

```
-- Compute the overall access for a given request.
compositeAccess :: RequestAttributes → Access
compositeAccess attrs =
  composeAccessList
    [primAccess1 attrs, primAccess2 attrs, primAccess3 attrs]

-- Example evaluations for specific requests.
> compositeAccess $ RequestAttributes 0 0 True False
Denied
> compositeAccess $ RequestAttributes 9 1 True True
ReadOnly
> compositeAccess $ RequestAttributes 7 1 True False
ReadWrite
```

As shown by the examples, different concrete request attribute combinations can lead to any of the three overall access levels for the composite resource.

2.2.3.2 *Symbolic Reasoning Problem*

We have shown how the system determines the composite access level given concrete request attributes. Now, we consider the inverse problem, which is often more interesting for verification and security analysis: *Given a target composite access level, can we find a set of request attributes that result in exactly that level?* If the target level is desired, we can use this to construct a request to gain access. If the target level is undesirable and should be prohibited, finding such a request reveals a security bug in the access policy.

Manually finding such specific attribute combinations can be difficult due to the complex ways the attributes influence the access decisions. Symbolic programming provides an automated approach to solve this inverse problem by treating the request attributes as symbolic variables and using SMT solvers to find values satisfying the desired outcomes. The following sections will discuss how this can be implemented, first using explicit path enumeration and then using Grisette's symbolic unions.

2.2.3.3 *Explicit Path Enumeration*

To solve the symbolic reasoning problem defined above, one possible strategy, as mentioned before, is *explicit path enumeration*. This involves exploring all feasible execution paths determined by the symbolic inputs (the request attributes) and recording the outcome (`Access` level) along with the path condition for each path. We will first illustrate this strategy to provide a clear contrast with Grisette's built-in approach using symbolic unions with merging. We implement it here in Haskell using a custom

`Paths` monad, drawing inspiration from techniques seen in systems like ROSETTE or described by Wei et al. [76], but specifically *without* implementing path merging.

The `Paths` monad is very similar to a non-determinism monad, but each outcome is augmented with a path condition. We use it to represent the result of a multi-path computation, where each element in the monad's collection corresponds to a distinct execution path. The monadic formulation gives us a clean and intuitive interface, and let's see how it works through examples.

A computation with a single, unconditional outcome can be represented in `Paths` using the standard monadic `return`. The path condition in this case is simply `true`.

```
> return $ "a" + 1 :: Paths SymInteger
Paths {runPaths = [(true,(+ 1 a))]}
```

Conditional branching requires a way to combine paths from 'then' and 'else' branches. We introduce a function `symIf cond t e`. It executes the then-branch `t` under the assumption `cond` is true, executes the else-branch `e` under the assumption `cond` is false, and collects all resulting paths.

```
-- if "a" then 1 else 2. Create two paths.
> symIf "a" (return 1) (return 2) :: Paths Integer
Paths {runPaths = [(a,1),((! a),2)]}
```

If branches are nested, `symIf` correctly combines the conditions:

```
-- Nested branching. Creates 3 branches.
-- if "a" then 1 else (if "b" then 2 else 3)
> symIf "a" (return 1) (symIf "b" (return 2) (return 3))
Paths {runPaths = [
  (a,            1),
  ((&& (! a) b), 2),
  ((! (|| a b)), 3)
]}
```

The `symIf` function (and the programs built upon it) maintains the invariant that within a `Paths` value:

- The path conditions associated with the different outcomes are mutually exclusive, meaning the conjunction of each pair of path conditions is unsatisfiable, and thus only one path can be taken at a time.
- The disjunction of all path conditions is a tautology (always true), meaning all possibilities are covered.

This representation ensures that exactly one outcome is valid for any given assignment to the symbolic constants involved in the path conditions.

Sequencing computations is handled using the monadic bind operation (»= or `do`-notation with `<-`). If a step in the sequence involves multiple paths, typically resulting from `symIf`, subsequent steps are executed for *each* incoming path. This explores the cross product of all branching computations, multiplying the number of paths.

Consider the following example:

```
-- x = if "a" then 1 else 2          -- Step 1: Branching into 2 paths
-- y = if "b" then x + 1 else x - 1  -- Step 2: Branching into 2 paths
-- return (x + y)
result :: Paths Integer
result = do
  -- 2 paths
  x ← symIf "a" (return 1) (return 2)
  -- Each path will further be splitted into 2 paths
  y ← symIf "b" (return $ x + 1) (return $ x - 1)
  return (x + y)

-- Result has 2 * 2 = 4 paths, each with a mutually exclusive condition
> result
Paths {runPaths = [
  ((&& a b),      3), -- a is true,  b is true:  x=1, y=2, x+y=3
  ((&& a (! b)),  1), -- a is true,  b is false: x=1, y=0, x+y=1
  ((&& (! a) b),  5), -- a is false, b is true:  x=2, y=3, x+y=5
  ((! (|| a b)),  3)] -- a is false, b is false: x=2, y=1, x+y=3
}
```

In the example, the first `symIf` splits into two paths, and the second `symIf` splits each of the two paths further into two paths. This results in 4 paths, each with a mutually exclusive path condition.

For completeness, the implementation of this `Paths` monad and `symIf` combinator is shown below. It is not necessary to understand the internal implementation, but this implementation clarifies how path conditions are managed and combined to maintain the invariants mentioned above.

```
data Paths a = Paths {runPaths :: [(SymBool, a)]}
  deriving (Show)

instance Functor Paths where
  fmap f (Paths ps) = Paths $ map (\(p, a) → (p, f a)) ps

instance Applicative Paths where
  pure a = Paths [(true, a)]
  (<*>) = ap
```

```haskell

instance Monad Paths where
  return = pure
  (Paths ps) >>= f = Paths $ do
    -- For each current path under path condition p with value a,
    (p, a) ← ps
    -- run the next computation 'f a', getting new paths.

    -- For each new path under new condition p' with value a',
    (p', a') ← runPaths (f a)
    -- combine the path conditions and return new results
    return (p .&& p', a')

-- Conditional branching: combine the paths from 'then' and 'else'
-- branches and maintain their path conditions.
symIf :: SymBool → Paths a → Paths a → Paths a
symIf c (Paths ts) (Paths es) =
  Paths $ (first (c .&&) <$> ts) ++ (first (symNot c .&&) <$> es)
```

The do-notation is desugared by the Haskell compiler into a sequence of monadic binds (»=):

```haskell
result =
  (symIf "a" (return 1) (return 2)) >>=
    -- The following is executed 2 times
    \x → (symIf "b" (return $ x + 1) (return $ x - 1) >>=
      -- The following is executed 4 times
      \y → return (x + y)
```

Now, we adapt the concrete access control logic (Listing 3) to this `Paths` monad for symbolic reasoning (Listing 4). We replace concrete types with symbolic ones for the attributes, use `symIf` instead of Haskell's `if`, use symbolic operators (e.g., `.<`), and use do-notation to compose the accesses, which enumerates all path combinations.

Listing 4: Symbolic access control with explicit path enumeration

```haskell
-- Reuse the concrete Access definition
data Access = Denied | ReadOnly | ReadWrite
  deriving (Eq, Ord, Show)

-- Symbolic request attributes.
-- Use SymInteger/SymBool instead of Integer/Bool.
data RequestAttributes = RequestAttributes
  { clearanceLevel :: SymInteger,
    authorizedSessionCount :: SymInteger,
    mfaVerified :: SymBool,
    onPublicNetwork :: SymBool
```

```
  }

-- Symbolic access logic using Paths monad.

-- The following is the original concrete primAccess1
--
-- primAccess1 :: RequestAttributes → Access
-- primAccess1 RequestAttributes {..} =
--   if clearanceLevel < 5
--     then Denied
--     else
--       if clearanceLevel + authorizedSessionCount > 9
--         then ReadOnly
--         else ReadWrite
primAccess1 :: RequestAttributes → Paths Access
primAccess1 RequestAttributes {..} =
  symIf
    (clearanceLevel .< 5)
    (return Denied)
    ( symIf
        (clearanceLevel + authorizedSessionCount .> 9)
        (return ReadOnly)
        (return ReadWrite)
    )

primAccess2 :: RequestAttributes → Paths Access
primAccess2 RequestAttributes {..} =
  symIf
    (clearanceLevel .< 7)
    (return Denied)
    (symIf mfaVerified (return ReadWrite) (return ReadOnly))

primAccess3 :: RequestAttributes → Paths Access
primAccess3 RequestAttributes {..} = do
  symIf
    (authorizedSessionCount .< 1)
    (return Denied)
    (symIf onPublicNetwork (return ReadOnly) (return ReadWrite))

-- Combining two access levels by enumerating all path combinations.
composeAccess :: Paths Access → Paths Access → Paths Access
composeAccess l r = do
  l' ← l
  r' ← r
  return (min l' r')

```

```haskell
composeAccessList :: [Paths Access] → Paths Access
composeAccessList = foldr composeAccess (return ReadWrite)

-- Overall access with all paths enumerated.
compositeAccess :: RequestAttributes → Paths Access
compositeAccess attrs =
  composeAccessList
    [primAccess1 attrs, primAccess2 attrs, primAccess3 attrs]

-- Evaluate for symbolic attributes
attr = RequestAttributes "clr" "auth" "mfa" "pub"
allPaths = compositeAccess attr
-- Check the number of paths generated
> length $ runPaths allPaths
27
```

As expected, evaluating the access level for these 3 primitive resources symbolically results in $3^3 = 27$ distinct paths, showing the exponential nature ($O(3^n)$ for n primitive resources where each can return 3 access levels) of explicit path enumeration. While feasible for this small example, this quickly becomes intractable for realistic systems.

To find an attribute set for a specific access level (e.g., `ReadOnly`), we filter these 27 paths for those yielding that level, extract their path conditions, and pass them to an SMT solver, one at a time, until a satisfiable condition is found. For verification purposes, finding a single counterexample leading to the undesirable access level reveals a security bug.

```haskell
-- Filter paths for a specific access level.
getConditions :: Access → Paths Access → [SymBool]
getConditions targetAccess (Paths paths) =
  fst <$> filter ((== targetAccess) . snd) paths

-- Solve the conditions one by one.
-- Find the attributes for the target access.
findAttrsFor :: Access → Paths Access → IO ()
findAttrsFor targetAccess paths =
  go $ getConditions targetAccess paths
  where
    go :: [SymBool] → IO ()
    go [] = putStrLn "Failed to find a solution."
    go (c : cs) = do
      r ← solve z3 c
      case r of
        Right m → putStrLn "Found solution:" ≫ print m
        _ → go cs

```

```
-- Find attributes for each access level.
-- Results shown are illustrative and solver may find different models.
-- Models are manually formatted for readability.
> findAttrsFor Denied allPaths
Found solution:
Model {auth → 0 :: Integer, clr → 0 :: Integer}
> findAttrsFor ReadOnly allPaths
Found solution:
Model {
  auth → 1 :: Integer,
  clr → 9 :: Integer,
  mfa → true :: Bool,
  pub → true :: Bool
}
> findAttrsFor ReadWrite allPaths
Found solution:
Model {
  auth → 1 :: Integer,
  clr → 1 :: Integer,
  mfa → true :: Bool,
  pub → false :: Bool
}
```

We can find attributes that yield the desired access levels, but this approach requires generating a potentially exponential number of paths. This highlights the path explosion problem and poses a scalability challenge in real-world scenarios.

2.2.3.4 *Symbolic Union Approach in Grisette*

While conceptually simple and offering a clean monadic interface via our illustrative `Paths` monad, the explicit path enumeration approach suffers from exponential complexity. Grisette addresses this scalability challenge by adopting the symbolic union approach, with its built-in `Union` type, which represents multiple potential outcomes under different path conditions with path merging.

Conceptually, a `Union a` value, like our `Paths a` value, holds outcome values of type `a` with their associated path conditions. However, unlike `Paths` which keeps all paths distinct, `Union` automatically identifies and merges paths that lead to equivalent outcomes.[3] When multiple paths result

3 In fact, `Union` merges not only equivalent outcomes, but also structurally mergeable outcomes, such as lists of symbolic Booleans with the same length. This will first be demonstrated in Section 2.3.2.1 and will be detailed in Chapter 4.

in the same value, their path conditions are combined, reducing the number of distinct entries stored in the union.

A significant advantage of GRISETTE's design is that this powerful merging capability is provided behind a monadic interface that remains almost the same as the clean, intuitive interface of the non-merging `Paths` monad. GRISETTE provides `mrg` variants of standard Haskell monadic combinators (like `mrgReturn` instead of `return`) and branching constructs (`mrgIf` instead of a basic `if`). Users interact with `Union` using standard `do`-notation and these specialized combinators, which can usually be used as drop-in replacements for the standard operators. These `mrg` variants (detailed in Section 4.5.2) benefit from automatic path merging.

Internally, GRISETTE's `Union` employs a novel representation called Ordered Guards (ORG), based on normalized if-then-else trees, which differs from the flat list-of-guards approach in systems like ROSETTE or MULTISE. This ORG representation is key to efficient merging. The details of ORG and its advantages will be discussed in Chapter 4. For now, let's focus on the usage and general benefits of symbolic unions.

We now adapt the access control code from Listing 3 again, replacing the custom `Paths` monad and its `symIf` with GRISETTE's `Union` monad and `mrgIf`. We also use features like smart constructors provided by GRISETTE to further simplify the code.

Listing 5: Symbolic access control with symbolic union

```
-- Reuse the concrete Access definition.
data Access = Denied | ReadOnly | ReadWrite

-- Use Grisette's TH procedures to derive instances and generate
-- smart constructors:
-- denied, readOnly, readWrite :: Union Access
makeGriseteADT ''Access

-- Reuse symbolic RequestAttributes definition from the Paths
-- implementation.
data RequestAttributes = RequestAttributes
  { clearanceLevel :: SymInteger,
    authorizedSessionCount :: SymInteger,
    mfaVerified :: SymBool,
    onPublicNetwork :: SymBool
  }

-- Symbolic access logic using Union monad.

-- The following is the Paths implementation for primAccess1
-- primAccess1 :: RequestAttributes → Paths Access
```

```
-- primAccess1 RequestAttributes {..} =
--   symIf
--     (clearanceLevel .< 5)
--     (return Denied)
--     ( symIf
--         (clearanceLevel + authorizedSessionCount .> 9)
--         (return ReadOnly)
--         (return ReadWrite)
--     )
primAccess1 :: RequestAttributes → Union Access
primAccess1 RequestAttributes {..} =
  mrgIf
    (clearanceLevel .< 5)
    denied
    ( mrgIf
        (clearanceLevel + authorizedSessionCount .> 9)
        readOnly
        readWrite
    )

primAccess2 :: RequestAttributes → Union Access
primAccess2 RequestAttributes {..} =
  mrgIf
    (clearanceLevel .< 7)
    denied
    (mrgIf mfaVerified readWrite readOnly)

primAccess3 :: RequestAttributes → Union Access
primAccess3 RequestAttributes {..} =
  mrgIf
    (authorizedSessionCount .< 1)
    denied
    (mrgIf onPublicNetwork readOnly readWrite)

-- Combining two access levels using Union monad and mrgReturn.
composeAccess :: Union Access → Union Access → Union Access
composeAccess l r = do
  l' ← l
  r' ← r
  -- In Grisette, merging is guided by a merging strategy, and Grisette
  -- relies on the monadic bind (≫=) to perform the merge.
  -- As we cannot add extra constraints to the standard (≫=) to
  -- resolve the strategy, we rely on mrgReturn to resolve it and
  -- propagate to the bind operator.
  mrgReturn $ min l' r'

```

```
-- Desugared monadic do with conceptual merging points:
-- merge (l >>= \l' → merge (r >>= \r' → mrgReturn $ min l' r'))

composeAccessList :: [Union Access] → Union Access
composeAccessList = foldr composeAccess readWrite

compositeAccess :: RequestAttributes → Union Access
compositeAccess attrs =
  composeAccessList
    [primAccess1 attrs, primAccess2 attrs, primAccess3 attrs]

-- Evaluate for symbolic attributes.
attrs = RequestAttributes "clr" "auth" "mfa" "pub"
finalAccess = compositeAccess attrs

-- Display the final merged union (under Grisette's ORG tree).
-- The compact representation has 3 outcomes.
-- Conditions for each outcome are merged.
> finalAccess
{ If
    (|| (< -5 (- clr)) (|| (< -7 (- clr)) (< -1 (- auth))))
    Denied
    ( If
        (|| (< 9 (+ clr auth)) (|| (! mfa) pub))
        ReadOnly
        ReadWrite
    )
}
```

Crucially, although intermediate steps (`composeAccess`) still involve enumerating paths (up to $i \times j = 9$ paths when combining two unions of size i and j), the `Union` immediately merges paths that evaluate to the same result (in this case, the same `Access` level) at the end of the `do`-block. This ensures that the number of distinct results in the `Union` is bounded, in this case by the 3 possible distinct `Access` levels, making each call to `composeAccess` enumerate at most 9 paths and produce a `Union` with at most 3 distinct outcomes. The overall complexity to combine n symbolic unions of `Access` is thus $O(n \times 3^2)$, as opposed to the exponential path explosion ($O(3^n)$) shown in explicit path enumeration. The path conditions for the merged outcomes are also combined into very compact forms within the ORG tree representation. The path condition terms are represented as DAGs, and subterms are shared globally, leading to a compact internal representation.

Finding an attribute set for a desired access level now requires solving only a single query against the final union structure, using Grisette's symbolic equality operator:

```
findAttrsFor :: Access → Union Access → IO ()
findAttrsFor targetAccess evalResult = do
  r ← solve z3 (evalResult .== return targetAccess)
  case r of
    Right m → putStrLn "Found solution:" ≫ print m
    _ → putStrLn "Failed to find a solution."

> findAttrsFor Denied finalAccess
Found solution:
Model {auth → 0 :: Integer, clr → 0 :: Integer}

> findAttrsFor ReadOnly finalAccess
Found solution:
Model {
  auth → 1 :: Integer,
  clr → 9 :: Integer,
  mfa → false :: Bool,
  pub → false :: Bool
}

> findAttrsFor ReadWrite finalAccess
Found solution:
Model {
  auth → 1 :: Integer,
  clr → 7 :: Integer,
  mfa → true :: Bool,
  pub → false :: Bool
}
```

This example highlights how Grisette's symbolic union, combining path merging and partial evaluation technique, provides a practical and more scalable approach compared to explicit path enumeration or pure `ite`-based encoding, especially when dealing with user-defined or library data types like `Access`.

2.3 SYMBOLIC DOMAIN-SPECIFIC LANGUAGE

The ability to represent values symbolically (Section 2.1), handle multi-path symbolic evaluation with unions (Section 2.2), and ultimately generate SMT constraints provides a powerful foundation. Building upon these core capabilities, Grisette aims to help developers build sophisticated tools for program analysis, verification, and synthesis by facilitating the creation and evaluation of *Symbolic Domain-Specific Languages (SDSLs)*.

An SDSL defines the syntax and semantics of a specialized language relevant to their problem domain (e.g., access control policies, simplified programming languages, or hardware descriptions). By implementing an interpreter or evaluator for this SDSL within GRISETTE, using its symbolic types, data structures, and control flow constructs, the SDSL evaluator automatically becomes a *symbolic* evaluator. Executing an SDSL program (or program space for synthesis, shown later in Section 2.3.3.1) with symbolic inputs yields symbolic values or constraints representing its behavior across many possibilities, which can then be passed to an SMT solver for reasoning. GRISETTE provides a comprehensive toolkit to simplify defining SDSLs and build practical reasoning tools.

To illustrate this capability, we introduce a simple language supporting basic arithmetic and Boolean operations and demonstrate how to build two common reasoning tools for it: an **expression equivalence verifier** and a simple **rewrite rule synthesizer**.

- The **expression equivalence verifier** takes two expressions, e_1 and e_2, and determines if they produce identical results for all possible assignments to their shared symbolic *input variables* I. For instance, it should verify that the expression representing $x + x$ is equivalent to $x \times 2$ (where $I = \{x\}$), but not equivalent to $x \times 3$. Formally, the verifier checks the validity of the universally quantified formula:

  $$\forall I.[\![e_1]\!](I) = [\![e_2]\!](I)$$

  where $[\![e]\!](I)$ denotes the semantic evaluation of expression e given inputs I. This can usually be done by solving its negation, and the original formula is valid if the negation is unsatisfiable:

  $$\exists I.[\![e_1]\!](I) \neq [\![e_2]\!](I)$$

- The **rewrite rule synthesizer**, by contrast, takes a target expression e_{target} and an *expression space* e_{space}. The expression space is essentially a template containing not only input variables I but also symbolic *synthesizable variables* (or "holes" [61]) H, which control the structure or constants within the template (Section 2.3.3.1). The synthesizer attempts to find concrete values for the holes H such that the instantiated e_{space} becomes equivalent to e_{target} for all possible inputs I. For example, given target $x + x$ ($I = \{x\}$) and space $x \times c$ ($H = \{c\}$), it should find $c = 2$. Formally, the synthesizer attempts to find a satisfying assignment for the existentially quantified holes H in the formula:

  $$\exists H.\forall I.[\![e_{target}]\!](I) = [\![e_{space}]\!](I, H)$$

The grammar of the language is defined as follows:

```
 Expr → IExpr | BExpr
IExpr → IVal Integer
      | IAdd IntExpr IntExpr
      | IMul IntExpr IntExpr
BExpr → BVal Bool
      | BAnd BoolExpr BoolExpr
      | BOr BoolExpr BoolExpr
      | BEq IntExpr IntExpr
      | BEq BoolExpr BoolExpr
```

This language includes integer expressions (`IExpr`) and Boolean expressions (`BExpr`). In this initial definition, the leaf expressions `IVal` and `BVal` directly hold concrete `Integer` and `Bool` constants. As we move to symbolic implementation for verification (Section 2.3.2) and synthesis (Section 2.3.3), we will adapt these to handle symbolic values (`SymInteger` and `SymBool`). The `BEq` constructor checks equality between two expressions, constrained to operate only on operands of the same base type.

2.3.1 *Concrete Language Implementation in Haskell*

To define a domain-specific language, we need to specify its *syntax* (the structure of valid programs) and its *semantics* (what those programs mean or how they execute). We will first implement the concrete version of our example expression language in Haskell before extending it to handling symbolic computation in the subsequent sections.

It is worth noting that implementing a concrete version first is not necessary when building symbolic tools with Grisette; one could directly implement the symbolic version for reasoning, or use the unified interface (Section 2.3.4) to build the language with both concrete and symbolic semantics. However, we present the concrete implementation here for pedagogical purposes. It serves two main goals:

1. It establishes the basic approach for defining the syntax and semantics in a familiar, non-symbolic setting.
2. It highlights how easily this concrete implementation can be adapted to its symbolic counterpart in Grisette (see Section 2.3.2 and Section 2.3.3), often with minimal changes beyond updating types and using Grisette's corresponding symbolic operations and combinators.

This ease of transition is a key benefit of the symbolic host language approach.

The grammar is defined in Haskell using a Generalized Algebraic Data Type (GADT), `Expr a`, which enforces type correctness at compile time. Each instance of `Expr a` represents an expression derived from the grammar. Boolean expressions have the type `Expr Bool`, and integer expressions have the type `Expr Integer`. The `BEq` constructor's type signature ensures that both operands have the same type `a`, which must support equality (`Eq a`).

```haskell
data Expr a where
  IVal :: Integer → Expr Integer
  IAdd :: Expr Integer → Expr Integer → Expr Integer
  IMul :: Expr Integer → Expr Integer → Expr Integer
  BVal :: Bool → Expr Bool
  BAnd :: Expr Bool → Expr Bool → Expr Bool
  BOr :: Expr Bool → Expr Bool → Expr Bool
  -- Requires Eq instance for a
  BEq :: (Eq a) ⇒ Expr a → Expr a → Expr Bool
```

For the semantics, we can implement a simple recursive interpreter (`eval`) using standard Haskell pattern matching and concrete arithmetic/Boolean operations:

```haskell
eval :: Expr a → a
eval (IVal a) = a
eval (IAdd a b) = eval a + eval b
eval (IMul a b) = eval a * eval b
eval (BVal a) = a
eval (BAnd a b) = eval a && eval b
eval (BOr a b) = eval a || eval b
eval (BEq a b) = eval a == eval b

> eval (IAdd (IVal 1) (IVal 2))
3
> eval (BEq (IVal 1) (IVal 2))
False
```

2.3.2 *Symbolic Language for Verification*

Now, let's adapt the language for symbolic reasoning, focusing first on *verification*. We will see that only minimal changes from the concrete implementation are needed to adapt it for verification.

2.3.2.1 *Symbolic Syntax*

For tasks like checking equivalence ($\forall I.[\![e_1]\!](I) = [\![e_2]\!](I)$), we need to represent fixed expression structures e_1, e_2 where the inputs I can be symbolic. To achieve this, we can simply define a symbolic syntax mirroring the concrete grammar, but with leaf nodes (`IVal`, `BVal`) holding symbolic types (`SymInteger`, `SymBool`). We reuse the name `Expr` for this symbolic representation.

```
data Expr a where
  -- Leaf holds symbolic integer.
  IVal :: SymInteger → Expr SymInteger
  IAdd :: Expr SymInteger → Expr SymInteger → Expr SymInteger
  IMul :: Expr SymInteger → Expr SymInteger → Expr SymInteger
  BVal :: SymBool → Expr SymBool
  BAnd :: Expr SymBool → Expr SymBool → Expr SymBool
  BOr :: Expr SymBool → Expr SymBool → Expr SymBool
  -- BasicSymPrim is a collection of constraints satisfied by basic
  -- symbolic primitive types including SymBool and SymInteger,
  -- Symbolic equality (SymEq) is part of BasicSymPrim.
  Eq :: (BasicSymPrim a) ⇒ Expr a → Expr a → Expr SymBool
```

2.3.2.2 *Symbolic Semantics*

The semantics can also be defined in almost the same way as our concrete recursive evaluator, using the symbolic operators (`.&&`, `.||`, `.==`, etc.) instead of the original concrete counterparts.

```
eval :: Expr a → a
eval (IVal a) = a
eval (BVal a) = a
eval (IAdd a b) = eval a + eval b
eval (IMul a b) = eval a * eval b
eval (BAnd a b) = eval a .&& eval b
eval (BOr a b) = eval a .|| eval b
eval (Eq a b) = eval a .== eval b
```

2.3.2.3 *Implementing the Verifier*

Building the verifier is also straightforward: evaluate both expressions with `eval` and query the solver about their inequality, trying to obtain counterexamples. If the solver returns `Unsat`, meaning no counterexample is found, the equivalence is verified.

```
verifyEquivalent :: (BasicSymPrim a) ⇒ Expr a → Expr a → IO ()
```

```
verifyEquivalent e1 e2 = do
  res ← solve z3 $ eval e1 ./= eval e2
  case res of
    Left Unsat → putStrLn "Equivalent"
    Left err → putStrLn $ "Unexpected failure: " <> show err
    Right model → do
      putStrLn "Not equivalent. Counterexample:"
      print model
      putStrLn $
        "LHS evaluates to: " <> show (evalSym False model $ eval e1)
      putStrLn $
        "RHS evaluates to: " <> show (evalSym False model $ eval e2)

x :: Expr SymInteger
x = IVal "x"
> verifyEquivalent (IAdd x x) (IMul x (IVal 2))
Equivalent
> verifyEquivalent (IAdd x x) (IMul x x)
Not equivalent. Counterexample:
Model {x → -6 :: Integer}
LHS evaluates to: -12
RHS evaluates to: 36
```

2.3.3 *Symbolic Language for Synthesis*

While the previous definition works for verification, program synthesis requires representing *spaces* of possible programs where the structure itself might contain choices or holes. A single expression space value now needs to represent potentially many grammatical derivations, guarded by symbolic conditions.

2.3.3.1 *Syntax for Program Spaces*

To achieve this, we further adapt the `Expr` type to include choices. The key change is that the operands of operators now have type `UExpr a`, i.e., `Union (Expr a)`, allowing sub-expressions (like the arguments to `IAdd`) to represent multiple possibilities. This allows representing program spaces with symbolic "hole" variables controlling structure.

```
data Expr a where
  IVal :: SymInteger → Expr SymInteger
  -- Operands are now Unions, allowing choices within sub-expressions
  IAdd :: UExpr SymInteger → UExpr SymInteger → Expr SymInteger
  IMul :: UExpr SymInteger → UExpr SymInteger → Expr SymInteger
```

```
  BVal :: SymBool → Expr SymBool
  BAnd :: UExpr SymBool → UExpr SymBool → Expr SymBool
  BOr :: UExpr SymBool → UExpr SymBool → Expr SymBool
  BEq :: (BasicSymPrim a) ⇒ UExpr a → UExpr a → Expr SymBool

type UExpr a = Union (Expr a)

-- Use makeGrisetteBasicADT due to existential 'a' in 'BEq'.
-- makeGrisetteADT does not work.
makeGrisetteBasicADT ''Expr
```

Now we can define target expressions and program spaces. To synthesize a rewrite rule, we solve the constraint $\exists H.\forall I.[\![e_{target}]\!](I) = [\![e_{space}]\!](I, H)$. We distinguish between *inputs variables* (I, universally quantified) and *synthesizable variables*, or *holes* (H, existentially quantified). As a convention, we use `x`, `y`, `z` for inputs and `c`, `c0`, `c1`, `cond` for holes. The following code defines target $x \times 2$ and a space exploring $x + x$, $x \times x$, or $x + c$. Note the use of smart constructors (`iVal`, `iAdd`, `iMul`) for `UExprs`, which implicitly wrap values in `Union`. Also note that target expressions, although represented as `UExpr`, should not contain internal choices or holes intended for synthesis.

Listing 6: Defining the symbolic syntax

```
x :: UExpr SymInteger
x = iVal "x" -- Input variable x

-- Target expression: x * 2 (fixed structure, no holes)
target :: UExpr SymInteger
target = iMul x (iVal 2)
> target
{IMul {IVal x} {IVal 2}}

-- Program space exploring potential rewrites.
-- c0 and c1 (Boolean holes) control structure;
-- c is an integer hole for a synthesized constant.
space :: UExpr SymInteger
space =
  mrgIf "c0" (iAdd x x)    -- If c0, use x + x
    (mrgIf "c1" (iMul x x) -- Else if c1, use x * x
      (iAdd x (iVal "c"))) -- Else, use x + c
-- Represents multiple expressions compactly.
> space
{If
  (|| c0 (! c1))
  (IAdd {IVal x} {IVal (ite c0 x c)})
  (IMul {IVal x} {IVal x})
}
```

It is worth noting how GRISETTE's `Union` structure for `space` compactly represents the choices through structural merging. Observe that the two `IAdd` expressions from the different branches, i.e., `IAdd x x` (when `c0` is true) and `IAdd x c` (when `c0` and `c1` are both false), are merged into one `IAdd` expression in the resulting `Union`. The identical first argument `x` is kept in the merged `IAdd` expression, while the second arguments, `x` (from the first `IAdd`) and `c` (from the second) are merged into a single `IVal` node containing the conditional symbolic integer `(ite c0 x c)`. The alternative choice, `IMul x x`, remains distinct as it has a different top-level constructor `IMul`.

This idea of structural merging is introduced by ROSETTE [72]. It helps keep the symbolic unions compact and is crucial for efficient symbolic evaluation. GRISETTE adopts this idea, and makes it configurable in a principled way by the `Mergeable` type class (detailed in Chapter 4). For ADTs, we can leverage GRISETTE's Template Haskell utilities like `makeGrisetteBasicADT` (as was used for `Expr`), and a suitable `Mergeable` instance that enables structural merging is automatically derived. Partial evaluation works on structurally merged values: we can extract the values with `do`-notation, and standard Haskell pattern matching works on the extracted values. For non-ADT user-defined types, or to customize merging behavior, users can manually implement `Mergeable` instances. The details of these merging mechanisms and strategies are explored further in Chapter 4.

2.3.3.2 *Evaluator for Program Spaces*

The evaluator for this new syntax for expression spaces must handle the `Union` operands. GRISETTE provides the `(.#)` combinator for this.

```
eval :: Expr a → a
eval (IVal a) = a
eval (BVal a) = a
eval (IAdd a b) = eval .# a + eval .# b
eval (IMul a b) = eval .# a * eval .# b
eval (BAnd a b) = eval .# a .&& eval .# b
eval (BOr a b) = eval .# a .|| eval .# b
eval (BEq a b) = eval .# a .== eval .# b
```

The `eval .# uexpr` combinator acts like `fmap eval` over the `Union` monad but additionally merges the resulting `Union a` back into a single value `a` using `ite`, provided `a` is `SimpleMergeable` (a type class for types, like `SymInteger` or `SymBool` here, that can be merged using the `ite` operator without `Unions`, see Section 4.3.1). Conceptually, it evaluates each choice

extracted from `uexpr` by applying `eval` to the choice, collecting the results into a new `Union` with the path conditions extracted from `uexpr`, and then merging this new union into a single value of type `a`. For example, evaluating `eval .# s` where `s` is a union $\text{If}(\text{cond}, e_1, e_2)$ where $e_1 = x + x$ and $e_2 = x \times x$ is roughly equivalent to:

```
-- Step 3: at the end of the do-notation, collect all the paths in a
--         union {If cond (+ x x) (* x x)} :: Union SymInteger
--         Step 3 is conceptual. We directly construct step 4 result
--         without constructing this intermediate result.
-- Step 4: merge the union into a single value
--         {(ite cond (+ x x) (* x x))} :: Union SymInteger
-- Step 5: simpleMerge extracts the single value out of the union:
--         (ite cond (+ x x) (* x x)) :: SymInteger
simpleMerge $ do
  -- Step 1: extract expression e from the union s
  e ← s
  -- Step 2: For each path, apply eval to the extracted expression:
  --         Path 1: cond,   eval (x+x) ⇒ (+ x x) :: SymInteger
  --         Path 2: ¬cond, eval (x*x) ⇒ (* x x) :: SymInteger
  mrgReturn $ eval s
```

With this new `eval` evaluator, evaluating the program space `space` using `eval .#` yields a single symbolic term incorporating the holes:

```
> eval .# space
(ite (|| c0 (! c1)) (+ x (ite c0 x c)) (* x x))
```

This demonstrates how multiple potential expressions can be evaluated simultaneously, yielding a single symbolic result, which is crucial for program synthesis tasks.

2.3.3.3 *Implementing the Synthesizer with CEGIS*

To solve the synthesis problem $\exists H.\forall I.[\![e_{\text{target}}]\!](I) = [\![e_{\text{space}}]\!](I, H)$, we employ Counter-Example Guided Inductive Synthesis (CEGIS) [61]. CEGIS solves this problem with an iterative loop:

1. *Synthesize (Guess)*: Find values for the holes in the expression space such that the resulting program satisfies the specification for a known set of input counterexamples (initially empty). This gives us a candidate program.

2. *Verify (Challenge)*: Check if the candidate program is equivalent to the target expression for all possible inputs, using the verifier logic we discussed in the previous section (Section 2.3.2.3). If equivalent, synthesis

succeeds. If not, the verifier produces a new counterexample. Add the new counterexample to the set and loop back to the synthesis step.

When the verifier cannot find any more counterexamples, we know that the CEGIS loop has found a correct program. Grisette provides a high-level `cegisForAll` function that implements this CEGIS loop. The `cegisForAll` function takes the SMT solver, the target expression, and a formula for the synthesis condition. The target expression argument helps `cegisForAll` identify the input variables and the holes: the symbolic constants appearing within the target expression should be treated as universally quantified input variables; the remaining symbolic constants appearing in the formula but not in the target expression should be treated as existentially quantified holes.

We can now synthesize with a simple program space. In Section 2.3.3.4, we will discuss modular construction of program spaces.

```
-- Although target is UExpr, we expect no choices and all
-- symbolic constants are universally quantified inputs.
synthesizeRewrite :: (BasicSymPrim a) ⇒ UExpr a → UExpr a → IO ()
synthesizeRewrite target space = do
  -- Use z3 as the SMT solver
  -- Set all the symbolic constants in the target as inputs
  r ← cegisForAll z3 target $
    -- The post-condition is the formula to be proven:
    cegisPostCond $ eval .# target .== eval .# space
  case r of
    (_, CEGISSuccess model) → do
      putStrLn "Synthesis succeeded:"
      print $ evalSym False model space
    _ → putStrLn "Synthesis failed"

x :: UExpr SymInteger
x = iVal "x" -- Input variable x

-- Example 1: Synthesize constant c such that x * c = x + x
-- target: x + x
-- space:  x * c
> synthesizeRewrite (iAdd x x) (iMul x (iVal "c"))
Synthesis succeeded:
{IMul {IVal x} {IVal 2}}

-- Example 2: Synthesize cond to choose between x + x and x * x
-- target: x * 2
-- space: If(cond, x + x, x * x)
> synthesizeRewrite
>   (iMul x (iVal 2))
```

```
>   (mrgIf "cond" (iAdd x x) (iMul x x))
Synthesis succeeded:
{IAdd {IVal x} {IVal x}}
```

A subtle point here is that we used the `UExpr a` type (which supports choices via `Union` operands) for both the synthesis `space` and the `target` expression. However, the `target` expression in synthesis typically has a fixed structure without internal choices or holes. While one can ensure this by convention when constructing the `target` (e.g., by avoiding `mrgIf` and holes), the type system itself doesn't enforce this distinction. An alternative would be to use the verification-oriented `Expr a` type (from Section 2.3.2.1) and its evaluator specifically for the `target`, but this would necessitate maintaining two slightly different sets of syntax and semantics definitions—one for verification and one for synthesis.

To address this potential duplication and provide a more streamlined experience, Grisette offers a *unified API*. This allows defining an SDSL with a single syntax and semantics, parameterized by type-level tags. These tags control the evaluation mode (concrete vs symbolic), statically enabling or disabling choices as appropriate. Consequently, the same SDSL definition can be used for concrete evaluation (fixed structures and constants), verification (operating on fixed structures), and synthesis (exploring program spaces), ensuring type safety and reducing code duplication. This advanced feature will be detailed in Section 2.3.4.

2.3.3.4 *Constructing Program Spaces with Fresh Monad*

In the previous section, we demonstrated synthesis using manually constructed program spaces, where holes and structural choices were explicitly defined. However, manually defining complex program spaces with numerous alternatives can become tedious and error-prone. To address this, Grisette provides the `Fresh` monad and associated utilities to systematically generate these choices and construct complex program spaces.

The `Fresh` monad facilitates the generation of *fresh* (i.e., unique) symbolic constants. The `chooseFresh` function, operating within the `Fresh` monad, takes a list of alternative expressions (`[a]`) and returns a `Fresh (Union a)`. The `Union a` represents the choice among the provided alternatives, with `chooseFresh` automatically generating and using fresh symbolic Boolean constants as guards for each alternative. A key guarantee of the `Fresh` monad is that distinct calls to choice-generation functions like `chooseFresh` (or fresh constant generation like `simpleFresh`) will yield unique symbolic constants, preventing unintended sharing or collision of choice variables.

For example, let's construct a program space with the following structure:

- The top-level operation is chosen from either addition (`IAdd`) or multiplication (`IMul`).
- The first operand `e1` is chosen from the input variable `x` or `y`, or a freshly generated symbolic integer hole `c`.
- The second operand `e2` is chosen independently, also from `x`, `y`, or `c`.

We define this within the `Fresh` monad using `do`-notation and then use `runFresh` to instantiate the final program space.

```
freshSpace :: Fresh (UExpr SymInteger)
freshSpace = do
  -- Generate a fresh SymInteger to be used as a symbolic constant hole
  c ← simpleFresh ()
  -- Choose e1 from x, y, c, creating fresh symbolic Booleans as guards
  e1 ← chooseFresh [IVal "x", IVal "y", IVal c]
  -- Choose e2 from x, y, c, creating fresh symbolic Booleans as guards
  e2 ← chooseFresh [IVal "x", IVal "y", IVal c]
  -- Choose the top-level operation (Add or Mul)
  chooseFresh [IAdd e1 e2, IMul e1 e2]

-- Execute the Fresh computation to instantiate the program space.
-- "_space" is a prefix for generating fresh symbolic identifiers.
instantiatedSpace :: UExpr SymInteger
instantiatedSpace = runFresh freshSpace "_space"

-- Result (reformatted for readability):
-- _space@0 is the SymInteger generated by simpleFresh.
-- _space@1, _space@2, etc., are SymBools generated by chooseFresh.
-- Grisette ensures all the fresh variables are unique.
> instantiatedSpace
{ If
    _space@5 -- Chooses between IAdd and IMul
    ( IAdd
        {IVal (ite _space@1 x (ite _space@2 y _space@0))} -- e1
        {IVal (ite _space@3 x (ite _space@4 y _space@0))} -- e2
    )
    ( IMul
        {IVal (ite _space@1 x (ite _space@2 y _space@0))} -- e1
        {IVal (ite _space@3 x (ite _space@4 y _space@0))} -- e2
    )
}
```

The resulting `instantiatedSpace` represents all $3 \times 3 \times 2 = 18$ possible expressions derived from the choices, compactly encoded using `Unions` and `ite` expressions.

As is common in monadic programming, this construction is modular and composable. We can easily define functions that generate parts of a program space and combine them. For instance, we can recursively define a function `freshSketch` to generate all expressions of a given grammar up to a certain depth, with the input variables x and y:

```
freshSketch :: Int → Fresh (UExpr SymInteger)
freshSketch 0 = do
  -- Base case (depth 0): choose from inputs x, y, or a fresh constant.
  c ← simpleFresh ()
  chooseFresh [IVal "x", IVal "y", IVal c]
freshSketch depth = do
  -- Recursive case: generate subexpressions of smaller depth.
  e1 ← freshSketch (depth - 1)
  e2 ← freshSketch (depth - 1)
  -- Choose top-level operation for the depth.
  chooseFresh [IAdd e1 e2, IMul e1 e2]
```

We can then use this `freshSketch` function with our synthesizer. For example, let's attempt to find an expression of depth 2 (meaning two levels of nested operations) that is equivalent to the target expression $x \times x + x \times y + x + y$. The synthesizer, exploring the space generated by `freshSketch 2`, can find that the target is equivalent to $(y + x) \times (1 + x)$ within this space:

```
> synthesizeRewrite
.   ( iAdd
.       (iAdd (iMul x x) (iMul x y))
.       (iAdd x y)
.   )
.   (flip runFresh "_sketch" $ freshSketch 2)
Synthesis succeeded:
{IMul {IAdd {IVal y} {IVal x}} {IAdd {IVal 1} {IVal x}}}
```

This example demonstrates how Grisette's `Fresh` monad and related utilities simplify the construction of complex and parametrically defined search spaces, which is a crucial aspect of building effective program synthesizers.

2.3.4 *A Unified Interface to Concrete and Symbolic Evaluation*

In Section 2.3.1 and Section 2.3.3.1, we demonstrated separate definitions for concrete and symbolic versions of our expression language. While defining a symbolic version is essential for reasoning tasks, it is often desirable to execute the same language definition on concrete inputs for testing or debugging purposes. A naive approach might involve writing only the

symbolic evaluator and then converting concrete inputs to their symbolic counterparts (e.g., `Integer` to `SymInteger`) for evaluation, followed by converting results back. This approach leverages the fact that symbolic types can represent values that are fully concrete (i.e., no symbolic constants are present), and relies on the partial evaluation and SMT term simplification to ensure that the result, when all inputs are fully concrete, is also fully concrete. However, this requires boilerplate conversions and can be error-prone if symbolic values are accidentally introduced during supposedly concrete executions.

To address this and promote code reuse, Grisette provides a *unified interface*. This allows a single language definition to serve for both concrete and symbolic evaluation, with the specific evaluation mode controlled at compile time by a type-level tag. This is achieved through type families like `GetInteger mode`, `GetBool mode`, and `GetData mode`. The `mode` parameter is an `EvalModeTag`, which can be `C` for concrete evaluation or `S` for symbolic evaluation. These type families resolve to standard Haskell types when `mode` is `C` (e.g., `GetInteger C` is `Integer`, `GetBool C` is `Bool`, `GetData C` is `Identity`), and to Grisette's symbolic types when `mode` is `S` (e.g., `GetInteger S` is `SymInteger`, `GetBool S` is `SymBool`, `GetData S` is `Union`).

Our expression language can then be redefined to be polymorphic over this `mode` parameter:

```
data Expr mode a where
  IVal :: GetInteger mode → IExpr mode
  IAdd :: DIExpr mode → DIExpr mode → IExpr mode
  IMul :: DIExpr mode → DIExpr mode → IExpr mode
  BVal :: GetBool mode → BExpr mode
  BAnd :: DBExpr mode → DBExpr mode → BExpr mode
  BOr :: DBExpr mode → DBExpr mode → BExpr mode
  BEq :: DIExpr mode → DIExpr mode → BExpr mode
type IExpr mode = Expr mode (GetInteger mode)
type BExpr mode = Expr mode (GetBool mode)
-- D denotes data, wrapped by GetData
type DExpr mode a = GetData mode (Expr mode a)
type DIExpr mode = DExpr mode (GetInteger mode)
type DBExpr mode = DExpr mode (GetBool mode)
```

This definition unifies the concrete syntax with the symbolic syntax suitable for both verification and synthesis. If finer-grained control is desired (e.g., distinguishing verification from synthesis more explicitly), one might use multiple `mode` parameters, though this single `mode` parameter covers many common use cases.

The evaluator function remains structurally similar but is now also polymorphic in `mode`. It uses unified operators from `Grisette.Unified` that dispatch to either concrete or symbolic operations based on the `mode` parameter. This dispatch is enabled by constraints like `EvalModeAll mode`, which bundles all necessary capabilities for the given mode.

```
import Grisette.Unified ((.#), (.==)) -- Example unified operators

eval :: forall mode a. (EvalModeAll mode) ⇒ Expr mode a → a
eval (IVal v) = v
eval (IAdd a b) = eval .# a + eval .# b -- .# handles GetData wrapper
... -- omitted for brevity
```

For symbolic tasks like synthesis or verification, we instantiate the expressions and the evaluator with mode `S`. Smart constructors (like `iVal`, `iAdd`, etc.) for types with evaluation mode would now return `DIExpr mode` or `DBExpr mode`, using `GetData mode` to wrap the constructed `Expr` node. The core logic of tools like `synthesizeRewrite` remains largely the same, except that the type signatures are updated to use `DIExpr S` or `DBExpr S`.

```
-- The smart constructors now produce DIExpr S or DBExpr S.
x :: DIExpr S
x = iVal "x"

target :: DIExpr S
target = iMul x (iVal "2")

space :: DIExpr S
space =
  mrgIf
    -- The @S type application helps the type checker resolve 'mode'
    -- when it cannot be inferred from context. This is common for
    -- polymorphic functions involving symbolic Booleans, like the
    -- unified mrgIf function.
    @S
    "c0"
    (iAdd x x)
    (mrgIf @S "c1" (iMul x x) (iAdd x (iVal "c")))

> synthesizeRewrite target space
Synthesis succeeded:
{Add {IntVal x} {IntVal x}}
```

For concrete evaluation, we instantiate with mode `C`, and the same `Expr` and `eval` definitions can be reused. The smart constructors for `DExpr C a` would use concrete literals (e.g., `1 :: Integer`) and `GetData C` (which is

`Identity`) to wrap the results. The `eval` function would then perform fully concrete evaluation.

```
conProg :: DIExpr C
conProg = iAdd (iVal 1) (iVal 2)
result :: Integer
result = eval .# conProg
> result
3
```

The unified interface allows developers to write their SDSL syntax and semantics only once, and the type system ensures that the correct operations (concrete or symbolic) are used based on the chosen evaluation mode. This reduces code duplication, avoids errors, and minimizes boilerplate code for conversions between concrete and symbolic types.

2.3.5 *Concluding Remarks on SDSL Construction*

The SDSL example, covering verification of fixed expressions and synthesis over program spaces (including systematic space generation with the `Fresh` monad), illustrates how Grisette provides a comprehensive and flexible toolkit for defining SDSLs. This infrastructure, when combined with Grisette's SMT solver interaction capabilities (such as the built-in CEGIS loop), enables developers to construct sophisticated automated reasoning tools for diverse problem domains.

2.4 HANDLING EFFECTS WITH MONAD TRANSFORMERS

Programs often exhibit various effects, such as exceptions (e.g., division by zero or assertion violations) or state modifications. When evaluating programs symbolically, these effects also need to be handled. Consider a division operation where the divisor is a symbolic value: during the symbolic evaluation, it is unknown whether this divisor will be zero. A symbolic evaluator cannot simply terminate or crash. Instead, it must represent and manage the possibility that an exception *may* occur. This leads to a fundamental challenge: how can a symbolic evaluator account for multiple outcomes and effects along each execution path, depending on the symbolic conditions? These outcomes and effects must be correctly propagated and, to avoid path explosion, merged with those from other paths. Crucially, for a path where an exception occurs, no further computation along that specific path should happen, modeling the short-circuiting nature of exceptions.

GRISETTE addresses this challenge with Haskell's monadic programming paradigm, allowing effects to be systematically incorporated into the symbolic evaluation process. This section focuses on exception handling as a representative example. We will first illustrate how exceptions are typically handled in concrete Haskell programs using the `Either` type. Then, we will demonstrate how GRISETTE extends this pattern to symbolic evaluation by integrating exception-handling capabilities with its `Union` monad (which manages multiple symbolic paths), using `ExceptT` monad transformer. This approach allows for modular reasoning about program behaviors that include exceptions. Note that the principles discussed here for exceptions can be generalized to other effects, using a set of type classes (see Section 4.5.3).

2.4.1 *Concrete Exception Handling with `Either`*

Let's enhance our concrete expression language from Section 2.3.1 with integer division (`IDiv`) and show exception handling using `Either`. We will use Haskell's `ArithException`, specifically `DivideByZero`, as the error value.

First, we update the concrete `Expr` definition to include the new `IDiv` constructor:

```haskell
data Expr a where
  IVal :: Integer → Expr Integer
  IAdd :: Expr Integer → Expr Integer → Expr Integer
  IMul :: Expr Integer → Expr Integer → Expr Integer
  IDiv :: Expr Integer → Expr Integer → Expr Integer
  -- Other constructors (BVal, BAnd, ...) omitted for brevity
```

The concrete evaluator for `Expr a` now returns `Either ArithException a` instead of just `a`. This type indicates that the evaluation can either succeed with a value of type `a` (wrapped in `Right`) or fail with an `ArithException` (wrapped in `Left`). Before implementing division, let's examine how other cases must be adapted.

Using `Either ArithException a`, we leverage its monadic interface. The `return` function wraps a success value in `Right`, and the `>>=` operator (implicit via the `do`-notation) sequences monadic actions and propagates errors in `Left`.

```haskell
eval :: Expr a → Either ArithException a
eval (IVal n) = return n -- return n is equivalent to Right n
eval (IAdd a b) = do
  -- If eval a is 'Left err', evaluation terminates with 'Left err',
  -- and a' binds to 'v' if eval a is 'Right v'
  a' ← eval a
  -- If eval b is 'Left err', evaluation terminates with 'Left err'
```

```
  b' ← eval b
  return (a' + b') -- Only executed if both a' and b' are Right
-- Other binary operators like IMul follow the same pattern
```

For `IVal`, no exception can occur, so we simply wrap the value `n` using `return`. For `IAdd`, the `do`-notation sequences the monadic actions. If evaluation of a sub-expression yields a `Left` value, the computation short-circuits, and the overall result is that error. This pattern with error handling can also be expressed concisely using *applicative combinators* like `liftA2` or `<$>`/`<*>`, from Haskell standard library:

```
eval (IAdd a b) = (+) <$> eval a <*> eval b
eval (IMul a b) = liftA2 (*) (eval a) (eval b)
```

Now, let's implement the `IDiv` case, which can *introduce* the division-by-zero error:

```
eval (IDiv n d) = do
  n' ← eval n -- Evaluate the numerator
  d' ← eval d -- Evaluate the denominator
  if d' == 0
    then throwError DivideByZero -- Raise the exception
    else return (n' `div` d')    -- Perform division if safe
```

Here, `throwError DivideByZero` signals the error when the denominator is zero, yielding a `Left DivideByZero` result that can be propagated to the top level. This logic can be encapsulated in a helper:

```
safeDiv :: Integer → Integer → Either ArithException Integer
safeDiv _ 0 = throwError DivideByZero   -- Or Left DivideByZero
safeDiv n d = return (n `div` d)        -- Or Right (n `div` d)

-- Using the helper
eval (IDiv n d) = do
  n' ← eval n
  d' ← eval d
  safeDiv n' d'
-- Or more concisely using join and liftA2
-- liftA2 safeDiv (eval n) (eval d) has the type:
--   Either ArithException (Either ArithException Integer)
eval (IDiv n d) = join $ liftA2 safeDiv (eval n) (eval d)
```

The `liftA2 safeDiv (eval n) (eval d)` expression would result in a nested `Either ArithException (Either ArithException Integer)` value, thus we use `join` to flatten it to a single-level `Either` value, signaling the error if it occurs in either level.

Let's see some examples:

```
> eval (IDiv (IVal 6) (IVal 3))
Right 2
> eval (IDiv (IVal 3) (IVal 0))
Left divide by zero
> eval (IDiv (IVal 3) (IAdd (IVal 1) (IVal (-1))))
Left divide by zero -- The divisor evaluates to 0
> eval (IAdd (IVal 3) (IDiv (IVal 3) (IVal 0)))
Left divide by zero -- Error propagates from the sub-expression
```

2.4.2 *Symbolic Exception Handling with `Union` and `ExceptT`*

To adapt this monadic exception handling pattern for symbolic evaluation with GRISETTE, we make two primary changes:

1. The base monad in our monad stack transitions from `Identity` (for single path concrete evaluation, noting that `Either e a` is isomorphic to `ExceptT e Identity a`) to `Union` (for multi-path symbolic evaluation). The monad stack becomes `ExceptT ArithException Union a`. This type now represents computations that can explore multiple symbolic paths (via `Union`) and where each path might either succeed with a symbolic value of type `a` or fail with an `ArithException`.

2. We use GRISETTE-provided monadic combinators (often prefixed with `mrg`, e.g., `mrgReturn`, `mrgLiftA2`, or using a dot prefix for operators, e.g., `.#`, `.<$>`, `.<*>`) instead of the standard Haskell ones. These combinators are `Union`-aware and manage path merging.

First, we adapt the symbolic `Expr` definition (Section 2.3.3.1) to include the the `IDiv`:

```
data Expr a where
  IVal :: SymInteger → Expr SymInteger
  IAdd :: UExpr SymInteger → UExpr SymInteger → Expr SymInteger
  IMul :: UExpr SymInteger → UExpr SymInteger → Expr SymInteger
  IDiv :: UExpr SymInteger → UExpr SymInteger → Expr SymInteger
  -- Other constructors omitted for brevity
```

The symbolic evaluator for this new `Expr` should now return results in `ExceptT ArithException Union a`. GRISETTE provides a polymorphic version of `safeDiv` that operates on concrete or symbolic integers and integrates with this monad stack. When `safeDiv` is called, if the divisor `d` contains a symbolic value such that we cannot rule out the case of zero, it will explore both outcomes: one where `throwError DivideByZero` occurs under the condition `d .== 0`, and another where the division proceeds under

`d ./= 0`. Both outcomes are represented in the underlying `Union` monad, each associated with its respective path condition.

```
eval :: Expr a → ExceptT ArithException Union a
eval (IVal a) = mrgReturn a -- Use mrgReturn for merging.
-- Example using mrgLiftA2 for conciseness
-- The type of (eval .# a) is ExceptT ArithException Union a
eval (IAdd a b) = mrgLiftA2 (+) (eval .# a) (eval .# b)
-- Or using .<$> and .<*>
eval (IMul a b) = (*) .<$> eval .# a .<*> eval .# b
-- Grisette's polymorphic safeDiv handles symbolic logic
eval (IDiv n d) = do
  n' ← eval .# n
  d' ← eval .# d
  safeDiv n' d'
-- Alternatively, we can use mrgJoin and mrgLiftA2 for IDiv
eval (IDiv n d) = mrgJoin $ mrgLiftA2 safeDiv (eval .# n) (eval .# d)
```

We use `eval .# a` to evaluate the `UExpr a` operands, which effectively applies `eval` to each expression within the `Union` and then combines the results. The crucial change from the concrete case is the use of Grisette's merging combinators like `mrgReturn`, `mrgLiftA2`, and `mrgJoin`, which correctly handle the `Union` layer for path merging.

We can reuse the same verification and synthesis tool definitions from Section 2.3.2.3 and Section 2.3.3.3 with minimal changes and they can now reason about expressions that might raise exceptions. When comparing two expressions with type `ExceptT ArithException Union a`, e.g., `eval .# e1` and `eval .# e2`, their symbolic equality `eval .# e1 .== eval .# e2` holds if, for all inputs:

- they both succeed, and their resulting symbolic values are equal, or
- they both fail with the same exception.

If one succeeds and the other fails, or they fail with different exceptions, or they succeed with different values, they have different symbolic behaviors and are considered *not* equal.

```
x :: UExpr SymInteger
x = iVal "x"
-- Example: x / (x * x) is equivalent to 1 / x,
-- including DivideByZero for x = 0.
> verifyEquivalent (iDiv (iVal 1) x) (iDiv x (iMul x x))
Equivalent

-- Synthesize c such that c / x is equivalent to x / (x * x).
> synthesizeRewrite (iDiv x (iMul x x)) (iDiv (iVal "c") x)
```

```
Synthesis succeeded:
{IDiv {IVal 1} {IVal x}}
```

We can also build new tools to find inputs that trigger specific exceptions. `solveExcept` takes a function to specify the paths of interest—here we are interested in paths that lead to `Left DivideByZero`.

```
-- Find inputs that trigger DivideByZero
findDivideByZero :: (BasicSymPrim a) ⇒ UExpr a → IO ()
findDivideByZero expr = do
  -- Check if evaluating expr raises DivideByZero
  -- r ← solve z3 $ eval .# expr .== throwError DivideByZero

  -- Or, we may also use solveExcept to 'interpret' the results in a
  -- specific way
  let -- Setting the goal to true instructs the solver to find a path
      -- leading to a Left DivideByZero
      goal (Left DivideByZero) = true
      -- false means other paths are ignored
      goal _ = false
  r ← solveExcept z3 goal (eval .# expr)
  case r of
    Left Unsat → putStrLn "No inputs cause DivideByZero"
    Left err → putStrLn $ "Solver error: " <> show err
    Right model → do
      putStrLn "Found input causing DivideByZero:"
      print model

-- The expression x / (x + 1) will raise DivideByZero for x = -1.
> findDivideByZero (iDiv x (iAdd x (iVal 1)))
Found input causing DivideByZero:
Model {x → -1 :: Integer}
```

This demonstrates how Grisette's monadic approach allows standard effect-handling patterns from Haskell to be naturally extended to symbolic evaluation. By composing monad transformers like `ExceptT` with `Union` and using Grisette's specialized merging combinators, we can construct tools that reason about complex program behaviors, including effects like exceptions, in a structured and modular way.

2.5 CHAPTER SUMMARY

This chapter introduced Grisette, a Haskell library designed as a typed, functional foundation for building reusable automated reasoning tools via symbolic compilation. We first discussed how symbolic constraints are de-

fined and solved in Grisette for basic symbolic programming, and then focused on Grisette's key capability: multi-path symbolic evaluation. We contrasted path enumeration and direct SMT `ite` encoding with the symbolic union approach, which Grisette implements via its `Union` type. This type, with its path merging capabilities, mitigates path explosion while supporting partial evaluation.

The chapter then illustrated how Grisette's features facilitate the construction of symbolic domain-specific languages (SDSLs). We then demonstrated how to build verification and synthesis tools for a simple expression language, using the `Fresh` monad for programmatic generation of symbolic programs and Grisette's unified interface for sharing SDSL definitions between concrete and symbolic evaluation.

Finally, we explored monadic effect management, focusing on exceptions. By composing `ExceptT` with `Union`, Grisette allows for errors to be represented as typed, conditional outcomes within the symbolic result. These are automatically propagated and merged, providing a structured way to reason about behaviors like division-by-zero. This overview established Grisette's core programming model, paving the way for deeper discussions on its design principles and detailed internal design.

3

A PRINCIPLED APPROACH TO SYMBOLIC EVALUATION SYSTEM DESIGN

The previous chapter showcased *what* Grisette offers. It introduced basic symbolic programming, symbolic unions, path merging, and monadic programming with Grisette, demonstrating their use in small examples. This chapter discusses *why* Grisette is designed this way, and *how* these core design choices contribute to making Grisette a safe, predictable, modular, and extensible symbolic engine for building symbolic reasoning tools.

Grisette's design is guided by a philosophy that prioritizes developer experience, safety, and performance when extending a host language for symbolic reasoning. This philosophy is realized through several key design principles that systematically address challenges faced by earlier symbolic host languages [11, 45, 49, 67–72]:

- *Safe Integration with Familiar Host Language Semantics*: Grisette aims to allow users to leverage their existing knowledge of the host language. However, naively mixing symbolic and concrete computations can be problematic. Thus, a core principle is to enable this integration while *ensuring safety through static typing*. This catches misuse (e.g., symbolic integers passed to concrete-only APIs) at compile time, preventing opaque runtime errors or mysterious solver failures.
- *Efficient Reusing Host Language Libraries via Partial Evaluation and Configurable Path Merging*: To avoid reinventing the wheel and to benefit from optimized, well-tested, and feature-rich host language libraries, Grisette is designed as a library that encourages *leveraging existing host language operations (and the wider Haskell ecosystem) through partial evaluation*. Data structures from host language libraries can be seamlessly integrated into Grisette's symbolic unions through its *configurable path merging* mechanisms (via the `Mergeable` type class). This principle ensures that users can apply standard Haskell functions to symbolic structures where appropriate, promoting reusability, ensuring good performance, and maintaining sound symbolic behavior, all with the safety provided by static typing.
- *Transparency and Predictability in Effect Management*: Symbolic evaluation often involves computational effects like errors and state. Grisette's principle is to ensure *transparency and predictability* by making

these effects explicit. This is achieved through a purely functional and monadic design, where effects are managed as explicit values within symbolic unions (typically using monad transformers), eliminating hidden global states and simplifying composition.

This chapter will explore how these principles are realized in Grisette's design, focusing on two main aspects:

- *Monadic Effect Management* (Section 3.3): This section demonstrates how Grisette's monadic approach, utilizing monad transformers like `ExceptT` and `StateT` over its `Union` monad, provides explicit, transparent, and predictable management of computational effects across symbolic paths. We will contrast this with less explicit systems.
- *Design of Symbolic Data Structures* (Section 3.4): Using associative maps as a running example, we analyze how the principles of safe integration and reuse of host language data structures are applied. We will show how type-driven concretization and extensible merging mechanisms allow for the flexible and efficient handling of symbolic data, crucial for performance and usability.

Throughout this chapter, we will contrast Grisette's solutions with alternative designs found in systems like CBMC [19], Rosette [49, 71, 72], SymPro [11], and Serval [45]. This comparison will illustrate the difficulties these alternative designs can face regarding error provenance, missed concretization for performance, or the complexities of state management. This will provide further context not only on *what* Grisette offers, but more importantly, *why* its design choices are made, and how these principles can be extended to new use cases.

3.1 FOUNDATIONAL GRISETTE CONCEPTS REFLECTING DESIGN PRINCIPLES

In Chapter 2, we introduced Grisette's core capabilities. These foundational concepts are not just standalone features; they directly stem from the design principles articulated above and work together to create a robust and developer-friendly symbolic evaluation system.

Users can write ordinary Haskell code that works on unknown values, and they can use Grisette's symbolic types like `SymInteger` alongside concrete ones. This directly supports the principle of *safe integration with familiar host language semantics*. Operations on symbolic types build symbolic expressions solvable by SMT solvers. The "safe" aspect of this integration is

critically guaranteed by Haskell's type system, which detects misuse (e.g., applying a concrete-only map lookup to a symbolic key) at compile time, preventing silent, confusing solver failures.

When programs branch on symbolic conditions, GRISETTE's `Union` type, guided by the `Mergeable` type class, manages multi-path execution and path merging. This realizes the principle of *efficient reusing host language libraries via partial evaluation and configurable path merging*. The `Union` is designed to retain sufficient concrete structure, enabling *partial evaluation*: ordinary Haskell functions can often be applied directly by distributing over the union's branches for concrete evaluation before results are merged. This allows existing, optimized Haskell libraries (like lists, maps, etc.) to be used within symbolic computations. The *configurable path merging*, which will be showcased in this chapter and detailed further in Chapter 4, ensures that the integration and reuse of host language libraries can be tuned for performance and correct semantics for diverse data structures, preventing path explosion while promoting reusability.

To handle computational effects like errors or state, GRISETTE's purely functional monadic design ensures *transparency and predictability*. The `Union` type itself is a `Monad`, and standard monad transformers like `ExceptT` and `StateT` can be used to make effects explicit components of the symbolic values. Each symbolic path carries its own result, error, or state update without reliance on mutable global state. This makes the flow of effects transparent, their behavior predictable, and the overall symbolic program easier to compose and reason about.

As we've seen, GRISETTE's main features come directly from its design principles. Next, we'll see how these design and their underlying principles help GRISETTE deal with real-world problems in symbolic evaluation.

3.2 MOTIVATING SCENARIO: SYMBOLIC ASSOCIATIVE MAPS

Associative maps, which store key-value pairs, are fundamental data structures in programming. In symbolic evaluation, they are often used to model environments, symbol tables, or memory states. Effectively integrating such maps into a symbolic context, where keys, values, or the map itself can be concrete or symbolic, raises design questions about their behavior and implementation. This chapter uses an associative map as a running example to outline desired symbolic semantics and highlight the challenges a symbolic evaluation system must address.

3.2.1 *Scenario 1: Reuse Map Implementations from Standard Libraries*

The most basic scenario involves a map with concrete keys mapping to symbolic values, and performing lookups or updates with concrete query keys. Consider an initial map: $M = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$.

- A lookup $M[1]$ should yield the symbolic value a.
- A lookup for a nonexistent key, $M[4]$, should result in a "KeyNotFound" error.
- An update to an existing key, $M' = M \oplus (1 \mapsto d)$, should result in a new state: $M' = \{1 \mapsto d, 2 \mapsto b, 3 \mapsto c\}$.
- An update involving a new key, $M'' = M \oplus (4 \mapsto e)$, should create a new entry: $M'' = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c, 4 \mapsto e\}$.

This mirrors the standard associative map interface. A key design goal for a symbolic host language is to allow efficient reuse of standard host-language hash map implementations for this common case.

However, even this "simple" scenario introduces design considerations.

For error handling, if a library operation (like a map lookup) fails due to a nonexistent key, this failure needs to be handled gracefully when the operation occurs under a symbolic path condition. For example, if a program attempts to access $M[1]$ on path p and $M[5]$ (nonexistent) on path $\neg p$, the symbolic evaluator must not crash. It should represent that the $\neg p$ path results in KeyNotFound, and this outcome should be encoded appropriately when making solver queries. Crucially, such *operational failures* (like KeyNotFound) should be distinguishable from *programming errors* (e.g., type mismatches), which should ideally trigger immediate, clear diagnostics to the user and not be silently converted into constraints.

Furthermore, updates raise questions about state management. Should $M' = M \oplus (1 \mapsto d)$ modify M in-place or produce a new, distinct map M'? In-place updates are challenging in a multi-path symbolic context, often requiring complex mechanisms to snapshot and restore state, especially when reusing host-language libraries not designed for symbolic evaluation. Functional updates (creating a new map) align better with purely functional symbolic evaluation and avoid snapshotting issues. However, this approach requires efficient ways to merge potentially many distinct map states created across different paths, as further discussed in scenario 4 (Section 3.2.4).

3.2.2 *Scenario 2: Symbolic Key Lookup Requires Custom Implementation*

Querying the map $M = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$ with an unconstrained, opaque symbolic integer key k presents a challenge. Standard hash map lookups require concrete keys for hashing to select at most one value in the map, while the desired symbolic result for $M[k]$ must capture all possibilities: $\text{ite}(k = 1, a, \text{ite}(k = 2, b, \text{ite}(k = 3, c, \text{KeyNotFound})))$. This implies that the lookup operation must effectively perform symbolic comparisons of k against all known concrete keys ($\{1, 2, 3\}$) in M. Additionally, an implicit condition ($k \in \{1, 2, 3\}$) will need to be added as a constraint to rule out KeyNotFound error. This typically requires a custom lookup logic, and one should consider how to prevent standard hash map lookups from the libraries (which would behave unexpectedly) from being applied with such opaque symbolic keys.

3.2.3 *Scenario 3: Efficient Lookup with Concretizable Symbolic Keys*

A common variation involves a symbolic query key that represents a choice among a known set of concrete values. Let $k = \text{ite}(p, 1, 4)$ be the query for $M = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$. Although k is symbolic, its structure reveals that it can only be 1 or 4. This process of identifying these feasible concrete values is called concretization [11]. Ideally, instead of performing the full symbolic comparison as in the previous scenario, the system should concretize k, perform concrete lookups for $M[1]$ (yielding a) and $M[4]$ (yielding KeyNotFound), and then combine these into the final symbolic result. This approach is significantly more efficient if the underlying map contains many concrete keys. The challenge lies in providing a robust and predictable mechanism for such concretization.

3.2.4 *Scenario 4: Merging Maps from Different Paths*

The value of a map can often diverge based on symbolic execution paths. Suppose we start with an initial map $M = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$. Under path condition p, an update $M_p = M \oplus (4 \mapsto d)$ occurs, while under path $\neg p$, a different update $M_{\neg p} = M \oplus (1 \mapsto e)$ occurs. The resulting map M' is effectively a conditional choice: $M' = \text{ite}(p, M_p, M_{\neg p})$.

Subsequent operations on M' must reflect this conditional nature. For example:

- $M'[1]$ should yield $\text{ite}(p, M_p[1], M_{\neg p}[1]) = \text{ite}(p, a, e)$,
- $M'[4]$ should yield $\text{ite}(p, M_p[4], M_{\neg p}[4]) = \text{ite}(p, d, \text{KeyNotFound})$.

As discussed in the recap (Section 3.1), representing such conditional choices is the primary motivation to introduce a symbolic union type. To manage complexity and avoid path explosion when many such conditional choices of maps are created, the system should ideally be able to *structurally merge* these maps when their structures are compatible. For instance, if we have two maps sharing the same keys, $M_0 = \{1 \mapsto a, 2 \mapsto b\}$ and $M_1 = \{1 \mapsto c, 2 \mapsto d\}$, then a choice $\text{ite}(p, M_0, M_1)$ should ideally resolve to a single map $\{1 \mapsto \text{ite}(p, a, c), 2 \mapsto \text{ite}(p, b, d)\}$. This merged form can be more efficient as subsequent operations would act on this single structure rather than needing to branch on p again.

However, as seen with M_p and $M_{\neg p}$ in our initial example, maps from different paths often have different keys sets (for example, M_p contains key 4, which $M_{\neg p}$ lacks). In such cases, they cannot be directly merged into a standard map structure without loss of information. The system must then represent the choice between these structurally distinct maps, for example, as distinct outcomes within a symbolic union.

While merging only maps with identical key sets is one approach, it might still lead to a large number of distinct map objects in the union if key sets frequently differ across paths. Alternative merging strategies might be desirable. For example, instead of representing a map strictly as (key, value) pairs, one might model it as mapping keys to an optional value. Missing a key or the key being mapped to an absent value have the same semantics. Under such a model, two maps could be merged into a single structure even if some keys are absent in one but present in the other, by making the "value-or-absent" itself conditional for each key. This could lead to a more compact symbolic representation of the choices, reducing the number of distinct maps and paths the symbolic evaluator needs to track, albeit potentially at the cost of more complex values within the single merged map.

The core challenge for a symbolic evaluation system, therefore, is to provide mechanisms that can:

1. Efficiently identify when data structures (like maps) from different paths are structurally compatible for merging.
2. Offer flexible, and ideally user-configurable, merging strategies to allow developers to choose the most effective way to merge conditional choices based on their specific use case and performance considerations.

This framework should enable different trade-offs between the number of distinct data structure instances stored in a union and the complexity of the individual (merged) instances.

3.2.5 *Scenario 5: Fully Symbolic Associative Maps*

For completeness, we also introduce the most general case, where the key itself is an opaque symbolic value. Consider an initial map $M = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$. If we perform an update $M' = M \oplus (k \mapsto d)$, where k is an unconstrained symbolic integer, the new map M' must symbolically reflect that d is now at key k, potentially overwriting a previous value if k coincides with an existing key. For example, a subsequent lookup $M'[1]$ must capture this conditional overwriting, yielding $\text{ite}(k = 1, d, a)$. This behavior requires symbolic comparison during the update to determine potential shadowing and cannot be directly implemented with standard host-language hash maps. Such scenarios often require modeling the map as an association list where updates prepend new (symbolic key, value) bindings and lookups scan from newest to oldest, naturally handling shadowing. Similar to the previous scenario, we need to carefully determine how these new data structures should be merged to keep compact in the states from multiple paths and avoid path explosion.

3.2.6 *Summary*

These scenarios show the challenges and desired properties that a robust symbolic evaluation system must address. Effectively supporting such data structures requires capabilities from the system itself to support a careful design of the symbolic map abstraction that leverages these capabilities.

First, the symbolic evaluation system should provide:

- **Multi-Path Value Representation**: A mechanism to represent values that are chosen conditionally based on symbolic path conditions (e.g., Grisette's `Union`).
- **Efficient Path Merging**: The system should provide means to control the merging of these symbolic data structures to help efficient use of them without causing path explosions.
- **Error Handling and Clear Diagnostics**: Operational failures should be handled gracefully under symbolic conditions; programming errors should be reported immediately with clear diagnostics.
- **Efficient State Management**: State management should be efficient, with no surprising behaviors caused by any unmanaged state modifications.

Given such system-level support, a user or library designer would then desire the following properties from a **symbolic associate map abstraction** built upon it:

- **Efficient Reuse with Concrete Keys**: When keys are concrete, operations should leverage efficient underlying host language implementations.
- **Clear Diagnostics on Misuse**: The data type should prevent undefined behaviors or other misuses with clear diagnostics, like when using a symbolic value where a concrete value is expected.
- **Support for Concretizable Keys in a Principled Way**: The map abstraction should handle operations where the key is a symbolic choice among concrete values efficiently. Also, we want to have a way to ensure that a key is concretizable at compile time.
- **Well-Defined Semantics for Opaque Keys**: With opaque symbolic keys, we should be able to define a type that captures the shadowing semantics.
- **Seamless Integration with Symbolic Evaluation**: The user-provided types should integrate seamlessly with the whole Grisette ecosystem, especially for the control of path merging, no matter using concrete keys or symbolic keys.

These scenarios highlight the challenges and desired properties that a robust symbolic evaluation system must address when supporting symbolic data structures and interacting with the host language. Effectively supporting such data structures requires careful design at both the system level and in the specific map abstraction. Before diving into data structure design, we first study how effects are managed in Grisette as a prerequisite. In Section 3.4, we will explore how Grisette's principled design provides solutions to these challenges, aiming to fulfill the properties outlined above.

3.3 MANAGING COMPUTATIONAL EFFECTS IN GRISETTE

Symbolic evaluation often involves operations that extend beyond pure value computation to include exceptional outcomes or state interactions. As a purely functional library, Grisette adopts Haskell's monadic paradigm to manage these effects explicitly within the symbolic values themselves, typically by composing monad transformers like `ExceptT` (for errors) or `StateT` (for states) with its base `Union` monad for multi-path evaluation.

This section details how this principled, functional approach leads to safer and more predictable symbolic evaluation behavior.

3.3.1 *Transparent and Predictable Error Handling*

Effective symbolic evaluation must robustly handle diverse failures. A principled error handling mechanism should clearly distinguish *programming errors* (bugs within the symbolic program) from *operational failures* (defined exceptional outcomes), especially when failures are conditional on symbolic path conditions. Consider the following pseudo code where `a`, `b`, `c` are symbolic Booleans:

Listing 7: Pseudo code for symbolic evaluation with exceptions

```
if a:
    x = 1
elif b:
    # Operational Failure: User-defined assertion
    raise AssertionError("Some assertion failure")
elif c:
    # Operational Failure: Arithmetic exception
    x = x / x # This should give ZeroDivisionError
else:
    # Programming Error: Type misuse (using integer as a map/list)
    x[0] = 1  # x is an integer, not indexable
```

An ideal symbolic evaluation system processing this code should:

- *Immediately diagnose programming errors:* The misuse `x[0] = 1` should trigger an immediate, clear error report without further evaluation.
- *Represent operational failures as conditional outcomes:* `AssertionError` (path $\neg a \wedge b$) and potential `ZeroDivisionError` (path $\neg a \wedge \neg b \wedge c \wedge x = 0$) should not crash the evaluation. Instead, these failures should be recorded with their path conditions for subsequent SMT queries to avoid or target these paths.
- *Support flexible querying and provenance:* The system should be able to distinguish different operational failures (e.g., `AssertionError` vs. `ZeroDivisionError`) and retain their provenance information for debugging or fine-grained, targeted queries.

Conflating programming errors with operational failures or failing to differentiate operational failures makes debugging harder and limits the construction of sophisticated verification tasks beyond treating all failures as

generic "bad states". This section discusses GRISETTE's approach to meet the requirements and contrast it with common alternative mechanisms.

3.3.1.1 *Grisette's Approach: Distinguishing and Managing Errors with Monads*

GRISETTE leverages Haskell's static type system and purely functional error handling to clearly separate programming errors from operational failures, aiming for early bug detection and a clear representation of conditional outcomes.

Many programming errors, especially type mismatches, are caught by the Haskell compiler. For other misuses not caught by types, GRISETTE favors explicit runtime diagnostics: e.g., its `Eq` instance for `SymInteger` throws a clear exception if used, because it does not have the symbolic equality semantics, which is likely the intended behavior.[1] The exception message then guides the user to the correct symbolic operator (`.==`) rather than yielding a misleading concrete `Bool` or silently converting this to an SMT constraint stating that the path is infeasible. This ensures immediate feedback for such programming errors.

Operational failures are managed using Haskell's standard error handling types like `Maybe` or `Either`, embedded within `Union` monad, sometimes via the `ExceptT` monad transformer (first shown in Section 2.4). For example, safe list indexing `(!?)` returns a `Maybe ValueType`. When used in GRISETTE with `Union`, it correctly captures the conditional `Nothing` (out-of-bounds for a path) versus `Just value` outcomes:

```
list = mrgIf p (return []) (return [a]) :: Union [SymInteger]
res = do l ← list; mrgReturn (l !? 0) :: Union (Maybe SymInteger)
> res
{If a Nothing (Just a)}
-- Querying for a specific successful outcome:
> solve z3 $ res .== (return (Just 1))
Right (Model {a → 1 :: Integer, p → false :: Bool})
```

For richer error information, the error type in `ExceptT` can be a custom ADT, enabling detailed error diagnostics and provenance, with minimal implementation effort, as illustrated with `ErrorWithOrigin`:

```
data ErrorWithOrigin e =
  ErrorWithOrigin {err :: e, origin :: Maybe T.Text}

```

1 GRISETTE provides an `Eq` instance for symbolic types like `SymInteger`, which throws runtime exceptions on misuse (e.g., `x == y` instead of `x .== y`), rather than omitting the instance. This is because `Eq` is fundamental and a superclass for many other important type classes like `Integral`.

```haskell
-- Captures the Haskell call stack and extract the function name where
-- throwErrorWithOrigin is called.
throwErrorWithOrigin ::
  (HasCallStack, TryMerge m, MonadError (ErrorWithOrigin err) m) ⇒
  err → m ()
throwErrorWithOrigin e = mrgThrowError . ErrorWithOrigin e $
  case IsList.toList callStack of
    (_ : (o, _) : _) → Just $ T.pack o
    _ → Nothing

-- Find if an error can happen in the specific function.
findErrorWithOrigin ::
  (SymEq e) ⇒
  ExceptT (ErrorWithOrigin e) Union () → T.Text → e → IO ()
findErrorWithOrigin evalResult funcName exc = do
  let goal (Left e) = e .== ErrorWithOrigin exc (Just funcName)
      goal _ = false
  solverResult ← solveExcept z3 goal evalResult
  print solverResult

data Exc = Exc1 | Exc2
inner :: (HasCallStack) ⇒ ExceptT (ErrorWithOrigin Exc) Union ()
inner = mrgIf "b" (return ()) (throwErrorWithOrigin Exc1)

outer :: (HasCallStack) ⇒ ExceptT (ErrorWithOrigin Exc) Union ()
outer = do
  mrgIf "a" (return ()) (throwErrorWithOrigin Exc2)
  inner

> findErrorWithOrigin outer "outer" Exc1
Left Unsat
> findErrorWithOrigin outer "outer" Exc2
Right (Model {a → false :: Bool})
> findErrorWithOrigin inner "inner" Exc1
Right (Model {a → true :: Bool, b → false :: Bool})
```

This approach allows users to define rich error types easily and flexibly query them using `solveExcept` with custom interpretation of the errors. Notably, detailed error tracking does not necessarily add SMT overhead if specific provenance details are not part of a query (a benefit of Grisette's ORG representation, see Section 4.5.4).

In summary, Grisette ensures programming errors are caught early (either compile-time or immediate runtime exception), while operational failures become explicit, typed, conditional values within `Union`, supporting rich diagnostics and flexible interpretation without a global error state.

3.3.1.2 *Context: Alternative Error Handling Mechanisms*

To better understand the advantages of GRISETTE's monadic approach, consider an alternative design common in some symbolic systems, particularly those integrated with languages using exceptions as a primary error handling mechanism. This alternative may expose the language's native exception model directly, allowing external libraries that rely on exceptions to be used within symbolic evaluation. In such systems, if an operation under a symbolic path condition p raises any exception, the symbolic engine typically adds $\neg p$ to a global constraint store (effectively asserting path p is infeasible) and continues execution along other paths. Subsequent SMT queries then implicitly incorporate all such accumulated global constraints.

This global constraint mechanism, while potentially offering easier integration with existing libraries and a familiar exception model for beginners, poses significant challenges when applied to scenarios like our example in Listing 7:

- *Masking of Programming Errors*: The type misuse `x[0] = 1` in our example, if it raises a runtime exception (as is common in dynamically-typed languages), would likely be converted into a global path infeasibility constraint, indistinguishable from operational failures. This prevents the user from receiving an immediate diagnostic for their bug, leaving them with potentially inexplicable SMT results (e.g., an unexpected `Unsat`). This issue is a recognized concern in systems like ROSETTE [67, 70].

- *Loss of Error Provenance and Inflexible Interpretation*: The distinct failures in our example (`AssertionError`, `ZeroDivisionError`, and type misuse `x[0] = 1`) would all be uniformly translated into path infeasibility constraints. This conflation makes it difficult to distinguish their origins or to perform targeted queries, such as "find inputs causing `AssertionError` specifically". As we discuss further in the context of TENSORRIGHT's verification tasks (Chapter 5), this fixed interpretation of all errors as path infeasibility is not flexible enough for complex symbolic reasoning tasks.

- *Global Coupling and Debugging Challenges*: A global constraint store inherently creates coupling between different parts of the program. Constraints arising from unrelated code can accumulate in this store and non-locally affect SMT queries in unexpected ways. This makes debugging surprising solver outcomes challenging, as the true cause might be obscured by a large set of human-unreadable SMT constraints.

GRISETTE's design, by contrast, aims to avoid these pitfalls. It ensures programming errors are either compile-time type errors or result in direct runtime exceptions with clear diagnostics. Operational failures, on the other hand, are explicitly represented as typed, conditional outcomes within the symbolic value itself (typically using `ExceptT` over `Union`). This approach keeps error semantics localized and allows them to be flexibly interpreted with custom logic—much like transforming any other symbolic value—at the SMT query construction stage.

3.3.2 *Functional State Management with Monads*

Beyond error handling, symbolic evaluation often needs to model computations with state, such as an interpreter's symbol table or a simulated memory state. GRISETTE manages state through its core purely functional and monadic approach. Instead of relying on mutable data structures and side effects, states are represented by immutable data structures, and updates produce new state values. This state is threaded through computations using monad transformers, like `StateT StateType Union ResultType`, to simulate an "imperative" style of programming. An update to the state, perhaps via `mrgPut newState` or `mrgModify updateFunction` causes the subsequent monadic computations to see and operate on the new state for that particular path.

While perhaps having a steeper initial learning curve for those accustomed to imperative programming, this functional approach offers significant advantages in the context of symbolic execution, primarily due to its simplicity and uniformity. It provides an extensible way to manage any kind of state that can be represented as an immutable data structure, provided that it implements the `Mergeable` type class. The complexities of handling state changes across multiple symbolic paths are managed by GRISETTE's existing `Union` merging mechanisms rather than requiring complex state management logic in the symbolic engine for specific mutable types.

To appreciate the benefits, consider alternative approaches. Handling mutations in arbitrary external libraries or host language mutable types safely and symbolically is a difficult problem. One approach, seen in whole-program analysis tools like CBMC [19], is for the symbolic engine to take full control of all memory and state, effectively reimplementing the language's mutable semantics. While comprehensive for the target language, this typically does not extend to easy integration with pre-compiled external libraries. They may require source code or stub implementations of the libraries [58].

Symbolic host languages like Rosette [72] offer another model by providing sophisticated built-in support for managing symbolic mutations to a *select set* of lifted mutable host types, such as mutable vectors (`vector`) and boxes. When an operation mutates such a lifted type under a symbolic condition, the system internally records its original state performs the mutation for one path, records the new state, rolls back the mutation, evaluate other paths from the original state, and finally merges the resulting states (changes). This mechanism is powerful for the types it supports. However, its key limitation is also extensibility. This internal state management is typically not generalizable to arbitrary mutable types from the host language or external libraries for similar reasons encountered by systems like CBMC: lack of information about arbitrary code's effects. If a user uses an unmanaged mutable type (e.g., a standard mutable hash map) under symbolic conditions, mutations might occur directly on the underlying object without the symbolic engine's snapshot-rollback-merge process. This can lead to incorrect symbolic results, as mutations from one symbolic path may "leak" into and affect the evaluation of other paths.

Grisette's purely functional approach to state, using immutable data structures within the `StateT` monad and relying on the `Mergeable` type class for state merging, avoids these challenges of selective lifting or complex internal rollback mechanisms for specific mutable objects. Any immutable Haskell data structure can serve as the state. Its mutation is handled by creating new instances, and its merging is just as the merging mechanism for any ordinary values, governed by the `Mergeable` instance. This provides a uniform, extensible, and conceptually simpler model for state management, yielding fewer surprising behaviors.

3.4 BUILDING SYMBOLIC DATA STRUCTURES

Having explored how Grisette manages computational effects like errors and state within its purely functional, monadic framework (Section 3.3), we now shift our focus to another important aspect of practical symbolic reasoning tools: the design and use of symbolic data structures. Building effective reasoning tools demands robust ways to represent and operate on symbolic data with various data structures. This section explores Grisette's approach to these challenges, using the associative map scenarios and desired properties introduced in Section 3.2 as our primary case study. We will show how Grisette's design principles—safe integration with host language semantics, efficient reuse through partial evaluation and configurable merging, and transparent effect management—are applied to ensure safe and performant use of symbolic data structures.

3.4.1 *Type Safety and Diagnostics for Concrete-Keyed Maps*

As shown in our Scenario 1 of our associative map discussion (Section 3.2.1), a common and efficient use case in symbolic evaluation involves maps from concrete keys to symbolic values. GRISETTE directly supports this by allowing the use of standard Haskell libraries, such as `HashMap` from the `unordered-containers` package [65], for this purpose. Operations with concrete keys are straightforward and benefit from Haskell's static type checking:

```
import qualified Data.HashMap.Strict as HM

w, x, y, z :: SymInteger
(w, x, y, z) = ("w", "x", "y", "z")
-- A symbol table mapping concrete keys to symbolic integers
hashMap :: HM.HashMap Integer SymInteger
hashMap = HM.fromList [(0, x), (1, y)]

> HM.lookup 0 hashMap
Just x
> HM.lookup 2 hashMap
Nothing
```

The primary benefit of static typing is immediate: if a map expects concrete `Integer` keys, attempting to query it with a symbolic key, such as `SymInteger`, results in a compile-time type error:

```
> HM.lookup ("a" :: SymInteger) symbolTable
-- This yields a compile time error:
Couldn't match type 'Integer' with 'SymInteger'
Expected: HashMap SymInteger SymInteger
  Actual: HashMap Integer SymInteger
```

This compile-time checking, a direct benefit of Haskell's static typing and GRISETTE's design principle of safe integration, directly addresses the desired property of preventing the misuse of symbolic keys with standard concrete-keyed hash maps and providing clear diagnostics. As discussed in our Scenario 2 (Section 3.2.2), the standard `HM.lookup` operation does not have the intended behavior with symbolic keys to compare against all stored concrete keys. If such a misuse occurred in a dynamically-typed system, it may only be caught at runtime. As discussed in Section 3.3.1.1, if that system then uniformly translates runtime errors into constraints, the original programming error could be silently masked.

Achieving the desired semantics requires the programmer to explicitly implement custom logic:

```
-- Grisette provides all operations via type classes.
-- SymEq allows symbolic equality.
-- ToSym allows converting a concrete value to an equivalent symbolic
-- type.
lookupOpaqueSymKey ::
  (SymEq sk, ToSym ck sk, Mergeable v) ⇒
  sk → HM.HashMap ck v → Union (Maybe v)
lookupOpaqueSymKey k m =
  HM.foldlWithKey
    (\acc ck v → mrgIf (k .== toSym ck) (mrgJust v) acc)
    mrgNothing
    m

> lookupOpaqueSymKey ("x" :: SymInteger) hashMap
{If (! (|| (= x 1) (= x 0))) Nothing (Just (ite (= x 1) y x))}
```

This custom `lookupOpaqueSymKey` explicitly constructs the desired conditional logic. Grisette's type system ensures that standard `HM.lookup` cannot be misused for this purpose, forcing the programmer to explicitly state the intention of using a symbolic key for a concrete-keyed map.

Furthermore, to prevent the misleading use of standard `HM.HashMap` with symbolic key types like `SymInteger`, Grisette deliberately does not provide a `Hashable` instance for `SymInteger`. This leads to a compile-time error if one attempts to create such a map:

```
symKeyHashMap :: HM.HashMap SymInteger SymInteger
symKeyHashMap = HM.fromList [(0, "x"), (1, "y")]
-- This will give a compile time error:
No instance for ‘Hashable SymInteger’
  arising from a use of ‘HM.fromList’
```

This design choice ensures that users are guided towards an explicit and principled way to clearly represent their intentions via the type system, without accidentally triggering undefined and misleading behaviors. This compile-time safeguards contribute significantly to the safety and predictability of symbolic programming in Grisette.

3.4.2 *Type-Enforced Concretization of Keys*

As highlighted in Scenario 3 (Section 3.2.3), when operating on data structures like concrete-keyed hash maps or arrays, the key or index used for querying might itself be a symbolic choice, e.g., `k = ite c 1 2`. To perform such operation efficiently while leveraging the underlying concrete data structure, symbolic evaluation systems must concretize the symbolic

values [11, 45]. This involves resolving the symbolic key/index into its feasible concrete values each associated with its path condition (here, 1 if c, 2 if ¬c), so that operations can then be applied to each concrete alternative.

GRISETTE integrates concretization directly into its static type system. A symbolic choice among known concrete values is explicitly typed as `Union ConcreteType` (e.g., `Union Integer`), which is distinct from opaque symbolic types like `SymInteger` (representing an arbitrary integer expression). This type-level distinction is fundamental to statically preventing the *missed concretization* problem [11], which can lead to excessive constraints or path explosion. This is discussed later in this section.

Consider a lookup in a standard `HM.HashMap Integer SymInteger` using a `Union Integer` key:

```
hashMap :: HM.HashMap Integer SymInteger
hashMap = HM.fromList [(0, x), (1, y), (2, z)]

key :: Union Integer
key = mrgIf a 0 (mrgIf b 1 3) -- Key is a choice: 0, 1, or 3

lookupResult :: Union (Maybe SymInteger)
lookupResult = do
  k ← key -- k becomes concrete (0, 1, or 3) per path
  mrgReturn $ HM.lookup k hashMap
```

When this `do`-block is evaluated, the monadic bind for `Union` explores each path defined by `key`. On each path, `k` takes on a specific concrete integer value. The standard `HM.lookup` function is then called with this *concrete* integer, allowing the efficient underlying hash map operation to proceed. The results are then merged:

```
> lookupResult
{If a (Just x) (If b (Just y) Nothing)}
```

The type `Union Integer` itself guarantees this concretizability by representing a collection of path-condition-guarded concrete integer values, and no complex runtime analysis of opaque terms is needed to discover these alternatives. This principle extends to other structures, such as using `Union Int` to index into a list.

Most other systems do not distinguish between opaque symbolic values and concretizable choices at the type level. While some systems (like ROSETTE [69] and SERVAL [45], which is based on ROSETTE) employ runtime heuristics to deconstruct simple `ite` terms like `idx = (ite a 1 3)`, these heuristics are fragile. For example, an arithmetic operation like `idx + 1` transforms the term into `(+ (ite a 1 3) 1)`. These new terms can easily

become truly opaque to simple structural analysis, as robustly determining its concrete possibilities generally requires solver-level reasoning. In contrast, if we use `idx :: Union Integer = mrgIf a 1 3`, Grisette's lifted arithmetic `idx + 1` yields `{If a 2 4}`, which is also a `Union Integer`, with its structure and concretizability guaranteed by the type system.

Failure to effectively concretize leads to the "missed concretization" problem [11]. This results in large symbolic constraints, exploration of infeasible paths, increased SMT query complexity, and consequently, severe performance degradation. The issues are discussed in work on symbolic profiling tools like SymPro [11] (integrated into Rosette [69]) and observed in systems like Serval [45], which requires the programmer carefully apply structural analysis to perform concretization. While profiling can help identify these bottlenecks, fixing them often requires significant manual developer effort, without the support of a type system.

Grisette's use of types like `Union Integer` provides a more principled and robust solution. By distinguishing between general opaque symbolic values and concretizable choices explicitly via the type system, Grisette guides programmers to clearly represent their intentions. This enables precise and efficient operations when symbolic choices are involved, thereby avoiding the common pitfalls of missed concretization and reliance on fragile runtime heuristics.

3.4.3 *Extensible and Configurable Structural Merging with Mergeable*

Our discussion of Scenario 4 (Section 3.2.4) highlighted a critical requirement: when symbolic execution produces different values of a data structure (like a hash map) on different paths, the system must efficiently merge these values to avoid path explosion within the `Union` structure. Grisette addresses this through its `Mergeable` type class, which provides a flexible mechanism for defining type-directed merging strategies. This allows users to specify how values of a given type, including library types like `HashMap`, should be combined when they appear in different branches of a `Union`. This extensibility also enables users to fine-tune merging behavior for performance. We present how the two merging strategies discussed in Scenario 4 can be implemented using the `Mergeable` type class in this sub-section.

3.4.3.1 *Strategy 1: Merging Hash Maps with Identical Key Sets*

A standard Haskell `HM.HashMap k v` has a single, fixed set of concrete keys. Therefore, merging two such maps from different symbolic paths into a single structure is only possible if both maps share the same set of concrete

keys. When this condition holds, and if the value type `v` is `SimpleMergeable` (meaning its values can be merged without a union, with an `ite`-like operation `mrgIte`), we can construct a merged map by merging the values element-wise.

This suggests a two-level merging approach: first, group maps by their key sets; then, for maps within each group (those with identical key sets), merge their values element-wise. GRISETTE's `Mergeable` type class allows implementing this using a `SortedStrategy`. The key set (`HM.keysSet`) serves as the index for this strategy, enabling GRISETTE's ORG merging algorithm (detailed in Chapter 4) to group maps with identical key sets. For each group, a nested `SimpleStrategy` then performs the element-wise merging using `mrgIte`.

```
instance
  -- MergingIndex is an alias for (Ord k, Typeable k).
  (MergingIndex k, SimpleMergeable v) ⇒
  Mergeable (HM.HashMap k v)
  where
  rootStrategy =
    SortedStrategy
      -- 1. Index by the key set.
      HM.keysSet
      -- 2. For each index, we use a SimpleStrategy to merge values.
      -- The SimpleStrategy provides an ite-like operation that is
      -- defined for the set of values with the key set.
      ( \keysSet →
          SimpleStrategy $
            \cond t f → HM.unionWith (\tv fv → mrgIte cond tv fv) t f
      )
```

With this instance, the `Unions` of `HM.HashMap Integer SymInteger` merge maps sharing keys, while keeping those with different key sets as distinct conditional outcomes:

```
-- Assume x, y, z :: SymInteger; a :: SymBool.
h1, h2, h3 :: HM.HashMap Integer SymInteger
h1 = HM.fromList [(0, x), (1, y)]
h2 = HM.fromList [(0, y), (1, z)] -- Same keys as h1
h3 = HM.fromList [(0, z), (2, x)] -- Different keys

-- Merging h1 and h2 (same keys) yields a single map with conditional
-- values:
> mrgIf a (return h1) (return h2)
{HM.fromList [(0, ite a x y), (1, ite a y z)]}
-- Merging h1 and h3 (different keys) results in an If node:
> mrgIf a (return h3) (return h1)
```

```
{ If
    (! a)
    (HM.fromList [(0, x), (1, y)])
    (HM.fromList [(0, z), (2, x)])
}
```

If the map's value type `v` is `Mergeable` but not `SimpleMergeable` (e.g., an ADT like `Either`), we can still use the map with `Unions` being the value type, e.g., `HM.HashMap Integer (Union (Either Error SymInteger))`. The `Either` results under the same key but from different paths are then merged with `mrgIf`.

3.4.3.2 *Strategy 2: Merging Hash Maps with Different Key Sets*

The preceding strategy, which merges hash maps only if their key sets are identical, is less effective if operations like conditional inserts frequently alter key sets across symbolic paths. This can lead to many distinct hash map instances within a `Union` and may cause path explosion.

To address this, Grisette provides the flexibility to define alternative representation and merging strategies. One such strategy ensures that two maps *always* merge into a single map structure, even if their key sets differ. This involves changing the representation of values within the map: instead of directly storing a value of type `v`, the map stores a `Union (Maybe v)`. The `Union (Maybe v)` explicitly encodes whether a value `v` is present (`Just v`) or absent (`Nothing`) under different path conditions.

The `Mergeable` instance would then leverage this new representation: a `MergedMap` (internally an `HM.HashMap k (Union (Maybe v))`) would use a `SimpleStrategy` to merge any two such maps into one. The merging function `mrgIte` would then use the union set of the key sets of the two maps as the new key set, and the value for each key would use `Just` to represent a present value and `Nothing` to represent an absent value.

```
data MergedMap k v = MergedMap (HM.HashMap k (Union (Maybe v)))

instance
  (Hashable k, MergingIndex k, Mergeable v) ⇒
  Mergeable (MergedMap k v)
  where
  rootStrategy = SimpleStrategy mrgIte

instance
  (Hashable k, MergingIndex k, Mergeable v) ⇒
  SimpleMergeable (MergedMap k v)
  where
```

```
  mrgIte c (MergedMap t) (MergedMap f) =
    -- onlyT, onlyF, both are maps that contains the keys that are
    -- only present in t, only in f, and in both t and f, respectively.
    -- mrgNothing returns a Nothing wrapped in Union.
    let onlyT = fmap (\v → mrgIf c v mrgNothing) $ HM.difference t f
        onlyF = fmap (\v → mrgIf c mrgNothing v) $ HM.difference f t
        both = HM.intersectionWith (\tv fv → mrgIf c tv fv) t f
     in MergedMap $ HM.unions [onlyT, onlyF, both]
```

The merge is shown in the following example:

```
data Color = Red | Blue | Green
makeGrisetteADT ''Color

map1, map2 :: MergedMap Integer Color
map1 = MergedMap $ HM.fromList [(0, mrgJust Red), (1, mrgJust Blue)]
map2 = MergedMap $ HM.fromList [(0, mrgJust Blue), (2, mrgJust Green)]

> mrgIte c (return map1) (return map2)
fromList [
  (0, If c (Just Red) (Just Blue)), -- Key 0 present in both maps
  (1, If c (Just Blue) Nothing),    -- Key 1 present only in map1
  (2, If c Nothing (Just Green))    -- Key 2 present only in map2
]
```

The strategy ensures that a `Union (MergedMap k v)` always contains a single `MergedMap` instance, and the user may choose not to wrap it in a `Union`—just using `mrgIte` on `MergedMaps` is sufficient. The trade-off is that the values stored in the map are more complex, and custom `lookupKey/insertKey` functions are needed for the `MergedMap` type. Grisette's `Mergeable` type class allows users to define the type-directed merging logic and make such trade-offs, choosing the best representation and merging strategy that is most suitable for their use case.

3.4.4 *A Fully Symbolic Associative Map*

To conclude our discussion on associative maps, we briefly present how a map compatible with fully symbolic keys and values (as described in Scenario 5, Section 3.2.5) can be implemented and integrated within Grisette, leveraging the concepts already discussed. Such maps are typically represented as an association list—an ordered list of key-value pairs representing the update history, where newer entries effectively shadow older ones for the same key.

We can define a simple symbolic map and derive its necessary instances (including `Mergeable`) using `makeGrisetteADT`. Insertion prepends to the list, and lookup scans the list sequentially to build a conditional result:

```
data SymMap k v = SymMap [(k, v)]
makeGrisetteADT ''SymMap

symMap :: SymMap SymInteger SymInteger
symMap = SymMap [("a", "x"), ("b", "y"), ("c", "z")]

-- Inserting a key-value pair is prepending to the list.
insertKey :: (SymEq k) ⇒ k → v → SymMap k v → SymMap k v
insertKey k v (SymMap updates) = SymMap ((k, v) : updates)
> insertKeySym "d" "w" symMap
{SymMap [(d,w),(a,x),(b,y),(c,z)]}

-- Looking up a key in the map requires a sequential search.
lookupKey :: (SymEq k) ⇒ k → SymMap k v → Union (Maybe v)
lookupKey k (SymMap updates) =
  foldr
    (\(k', v) acc → mrgIf (k .== k') (mrgJust v) acc)
    (return Nothing)
    updates
> lookupKeySym "d" symMap
{ If
    (! (|| (= d a) (|| (= d b) (= d c))))
    Nothing
    (Just (ite (= d a) x (ite (= d b) y z)))
}
```

This `lookupKey` function performs a linear scan, using symbolic equality (`.==`, provided by `SymEq`) and `mrgIf` to construct a conditional result that correctly reflects the shadowing semantics.

The default `Mergeable` instance derived by `makeGrisetteADT` for `SymMap` performs a structural merge on the underlying list of pairs. This means if two `SymMap` instances from different symbolic paths have structurally different update histories, they will generally be kept as distinct entries in a `Union (SymMap k v)`. For example, here we use concrete keys to illustrate this structural list difference:

```
symMap0 :: SymMap Integer SymInteger
symMap0 = SymMap [(0, "x"), (1, "y")]
symMap1 :: SymMap Integer SymInteger
symMap1 = SymMap [(0, "x"), (2, "z")]
unionMap :: Union (SymMap Integer SymInteger)
unionMap = mrgIf a (return symMap0) (return symMap1)
```

```
7
8  -- Kept distinct because we cannot merge distinct concrete integers
9  -- with Integer type.
10 > unionMap
11 {If a (SymMap [(0,x),(1,y)]) (SymMap [(0,x),(2,z)])}
```

While this default merging is structurally correct, it may not always be the most efficient for preventing path explosion, as many difference in the update lists leads to distinct `SymMap` in the `Union`. Grisette's flexibility allows for alternative representations and custom `Mergeable` instances, also as shown in the previous section. For example:

- `SymMap [(Union k, Union v)]`: The individual keys and values in the association list are themselves merged in `Union`s, and the derived `Mergeable` instance for such a list of pairs of `Union`s would merge lists of the same length element-wise, by merging the corresponding `Union k` and `Union v` pairs. This can reduce the number of distinct `SymMap` objects in the outer `Union`, especially if each path updates the maps for similar times.

- `SymMap [Union (Maybe (Union k, Union v))]`: Here, each entry in the association list can be conditionally present (as specified by a `Maybe` type). With a custom `Mergeable` instance, which is similar to the merging strategy for concrete-keyed maps with different keys, two such lists could always be merged into a single list of a common length by padding with `Nothing`s and then merging entries element-wise.

Choosing the most effective representation and merging strategy is usually application-specific, involving trade-offs between the compactness of the overall `Union` structure, the complexity of the internal representation (e.g., keys and values as `Union`s), the cost of merging operations, and the efficiency of operations like `lookupKey`. Grisette's typed framework and extensible merging mechanism provide the necessary tools for users and library designers to explore these trade-offs and implement strategies that are most suitable for their use case.

This concludes our discussion on associative maps as a case study. These examples illustrate how Grisette's design principles—particularly static typing for enhanced safety and clear diagnostics, type-enforced concretization for efficiently handling symbolic choices over concrete elements, and an extensible merging framework via the `Mergeable` type class—contribute to building robust, predictable, and flexible symbolic evaluation tools.

3.5 CHAPTER SUMMARY

In this chapter, we have moved from "what" GRISETTE provides to "why" it is designed in this way, and "how" its key design decisions address the challenges of symbolic evaluation. We began by revisiting the foundational concepts of GRISETTE's symbolic programming model, then described scenarios involving associative maps that pose challenges in error handling, state management, safe concretization, and efficient merging to prevent path explosion.

In the first part on managing effects we saw how standard monad transformers applied over the `Union` make errors and states explicit per-path. This design cleanly separates programming mistakes that should halt and provide diagnostics from operational failure that is part of the symbolic evaluation result. It also avoids hidden global state and complex rollback logic to handle mutable states.

In the second part on symbolic data structures, we analyzed how static typing identifies library misuse and clearly shows programmers' intent regarding the semantics including concretization. We also showed that type-directed merging lets developers choose or customize merging strategies for maps across different branches. We saw that the same principles extend to general structures such as our list of update logs to represent symbolic maps.

Together, these design principles make GRISETTE a solid foundation for building symbolic reasoning tools. Static typing prevents accidental misuse and clarifies developer intent. Purely functional, monadic effects remove hidden state, decouple the system, provide better diagnostics, and simplify extensibility. Type-directed merging enables partial evaluation of ordinary library code without path explosion. In the next chapter, we will explore the details of our merging mechanism, particularly the Ordered Guards (ORG) representation.

4

REPRESENTATION OF SYMBOLIC VALUES

The `Union` type, introduced in Chapter 2 (specifically Section 2.2.2), is central to Grisette's ability to manage multiple execution paths under symbolic path conditions. It also enables partial evaluation of operations on complex data structures. While Chapter 2 demonstrated its usage and benefits from a user's perspective, this chapter explains in detail the novel internal representation of Grisette's `Union`: the *Ordered Guards (ORG)* representation.

The development of ORG is motivated by Grisette's core design principles. Grisette is built as a purely functional, monadic library to achieve safety and modularity, while also aiming for efficient and extensible framework for symbolic evaluation. As shown in Chapter 3, these principles, particularly the purely functional programming formulation, allow Grisette to provide support for rich, self-contained, and composable symbolic values. However, this design choice also means that these values often encapsulate more structural complexity than is typically found in systems that rely on global mutable states for managing computational effects. The ORG representation is specifically designed to manage this inherent complexity effectively, aiming to produce more compact symbolic formulas and support configurable merging strategies for diverse data types.

This chapter begins by motivating the need for a new symbolic union representation like ORG. We will discuss how Grisette's functional approach influences the structure of symbolic values and then examine the limitations of traditional list-based union representations, which typically rely on mutually exclusive path conditions. Following this, we will formally define the ORG data structure and its crucial *Hierarchical Merging Invariant*. This invariant is key to ORG's efficiency and relies on maintaining a canonical sorted order for elements within a union. We will then detail how the *type-directed merging strategies* are integral to ORG. This concept was first introduced in Section 2.3.3.1 and further discussed in Section 3.4.3 for customizing merging behavior for external types. These strategies define the canonical sorted order for values within an ORG union and maintain the representation invariant across diverse Haskell types. A formal semantics for the ORG merging algorithm will then be presented. Finally, we will discuss its application to features like flexible error handling, illustrating how ORG enables rich error provenance without sacrificing solver performance.

Ultimately, this chapter aims to demonstrate how the ORG representation, with its hierarchical merging invariant and type-directed merging strategies, serves as a cornerstone of Grisette's efficient and extensible foundation for automated reasoning.

4.1 MOTIVATION

An effective symbolic value representation is critical for the performance and expressiveness of any symbolic evaluation system. As highlighted in Section 2.2.2, relying solely on an SMT solver's native `ite` (if-then-else) construct can be problematic, especially when representing values that arise from multiple execution paths involving complex, user-defined data types. Such an approach often requires extensive engineering effort to encode all operations into SMT logic and may also lead to sacrificing the rich features and abstractions available in the host language.

Symbolic unions were introduced to address this complexity by storing multiple, partially concrete host language structures, each under its corresponding path condition. This enables the direct use of host language operations via partial evaluation. The design of the symbolic union's internal representation, as we will demonstrate, is particularly important for Grisette.

In Grisette, the adoption of a purely functional programming model means that all computational effects, such as error propagation or other program states, are explicitly managed and embedded within the symbolic values themselves. This design choice offers substantial benefits in terms of safety, modularity, and debuggability, as discussed in Chapter 3. However, it inherently leads to symbolic values that are structurally more complex. Consequently, these values can be more challenging to merge efficiently compared to systems where such effects might be handled by external, mutable states. This places unique demands on the union representation to manage the resultant value complexity without compromising symbolic evaluation efficiency or the conciseness of the logical formulas generated for SMT solvers.

4.1.1 *The Challenge: Increased Value Complexity in Functional Symbolic Execution*

In Chapter 3, we contrasted Grisette's purely functional approach with systems that rely on global mutable state. This contrast applied to handling operational failures (Section 3.3.1) and program state (Section 3.3.2). A key

consequence of GRISETTE's design is that all information related to computational effects becomes an integral part of the symbolic values manipulated by the program. This embedding of effects ensures referential transparency and facilitates modular reasoning, but it also naturally leads to symbolic values that are often more structurally complex than their counterparts in stateful systems, where such information is managed separately.

For instance, when an assertion is encountered in GRISETTE, its potential violation is not recorded in a separate, global verification condition. Instead, the symbolic value resulting from an operation that includes assertions typically takes the form of an `ExceptT ErrorType Union ResultType`. This structure explicitly represents all possible outcomes within a single, self-contained value: error states (e.g., `Left AssertionViolation`) if the assertion fails and error occurs under certain path conditions, and success states (e.g., `Right SuccessValue`) if the assertions hold. Consider the simple function:

```
f :: SymInteger → ExceptT AssertionViolation Union SymInteger
f a = do
  symAssert (a .< 0) -- Symbolic assertion
  mrgReturn (a - 2)  -- Some computation
```

Evaluating `f "a"` in GRISETTE yields a symbolic value that clearly shows these two distinct conditional outcomes:

```
> f "a"
{ExceptT {If (>= a 0) (Left AssertionViolation) (Right (- a 2))}}
```

In this GRISETTE example, the two outcomes are explicitly present in the resulting symbolic value. In contrast, a system like ROSETTE might evaluate a similar function to a simpler symbolic expression (e.g., `(+ -2 a)`). This simpler expression only corresponds to the successful outcomes, and information about the assertion does not directly alter the structure of this result value. Instead, the assertion condition `(< a 0)` is managed in a global verification condition, separate from the primary outcomes representing successful execution.

Similarly, managing program state functionally in GRISETTE, perhaps using the `StateT StateType Union ResultType` monad, often results in computations that yield pairs of the form `(StateType, ResultType)`. When merging outcomes from different paths, these state-result pairs present a specific merging challenge. Two such pairs, say `(s1, r1)` from one path and `(s2, r2)` from another, can only be fully merged into a single pair if both their respective state components (`s1` and `s2`) and their result components (`r1` and `r2`) can be fully merged. For example, if the state and result types are both integers, and the result components (`r1` and `r2`) are identical

(e.g., both are the concrete integer `2`), but the state components are distinct (e.g., `1` and `2`, respectively, representing different environments), the pairs `(1, 2)` and `(2, 2)` cannot be collapsed into a single pair whose result is simply `2` without appropriately representing the conditional nature of the accompanying state. This interdependence means that a difference in either part of the pair can prevent a complete merge of the pair structure itself, potentially leading to a greater number of distinct branches in the overall symbolic union.

Furthermore, features like rich error provenance, where distinct error types and associated contextual data are tracked to provide detailed diagnostic feedback (as discussed in Section 3.3.1), naturally increase the number of distinct, yet semantically related, outcomes that a single logical `Union` value might need to represent.

This inherent structural complexity of symbolic values in a purely functional system places significant demands on the underlying symbolic union representation. To effectively support GRISETTE's goals, such a representation must possess several key capabilities:

- It must efficiently merge a potentially large number of distinct conditional branches that arise from nested effects and symbolic choices.
- It must keep the representation of these merged values, and particularly their path conditions, as compact as possible to avoid generating excessively large or logically convoluted formulas that could overwhelm SMT solvers.
- It must flexibly and uniformly handle a diverse range of Haskell value types within unions. This includes primitive symbolic types like `SymInteger`, complex user-defined algebraic data types (ADTs), and even complex types from external libraries such as hash maps. Importantly, GRISETTE's extensible design should allow advanced users to define specialized merging logic for these types to accurately preserve their intended symbolic semantics or to optimize performance.

These demanding requirements highlight the need for a union representation more sophisticated than traditional flat list-based approaches, which can struggle under these combined pressures.

4.1.2 *Limitations of Traditional List-Based Union Representations*

A common approach to representing symbolic unions is to use a collection (effectively a list or set) of pairs, where each pair consists of a value and its path condition. This approach is found in influential systems like

ROSETTE [72] and MULTISE [56]. To ensure that the meaning of the union is unambiguous, the path conditions are usually made explicitly *mutually exclusive*. We refer to this as the *Mutually Exclusive Guards (MEG)* representation.

Consider the symbolic evaluation of the program snippet in Listing 8 using a MEG-style union, as illustrated in Figure 1.

Listing 8: Program for symbolic evaluation under MEG and ORG.

```
-- c0, c1 are symbolic booleans.

-- Values 1, 2, 3, 4 are treated as distinct concrete integers here,
-- meaning we will not merge them using an SMT ite in this example.
lst = if c0 then [1] else if c1 then [2,3] else [1,3,4]
res = head lst
```

When representing `lst` after its conditional assignment (see `lst` (merged) in the MEG column of Figure 1), the MEG union stores three distinct lists. Each list is associated with a path condition constructed to be mutually exclusive with the others: $[1]$ is guarded by c_0, $[2,3]$ by $\neg c_0 \wedge c_1$, and $[1,3,4]$ by $\neg c_0 \wedge \neg c_1$.

Subsequently, when `res = head lst` is evaluated, the `head` operation is distributed over each concrete list within this MEG union. This yields three guarded values (`res` (pareval) in Figure 1). The normalization and merging process for MEG then involves grouping identical values (e.g., the two instances of the integer 1) and then replacing each group with a single instance of the value, guarded by the disjunction of the original guards. Thus, in our example, the two occurrences of the value 1 are grouped and merged. The value 1 is now guarded by the new condition $c_0 \vee (\neg c_0 \wedge \neg c_1)$ (see `res` (merged)).

While conceptually straightforward, the MEG representation is not well-suited to deal with the structurally complex symbolic values common in GRISETTE's purely functional setting. The primary overhead is associated with creating and maintaining these mutually exclusive guards. This often involves introducing new conditions with negations of conditions for other outcomes (e.g., introducing $\neg c_0$ to guard the branches for $[2,3]$ and $[1,3,4]$ in the MEG column of Figure 1). When values are subsequently merged, their already potentially complex guards are connected with disjunctions. As shown in Figure 1, this tends to increase the size and logical depth of path conditions. Such growth in complexity can make the symbolic terms harder to simplify and increase the cost of solving the constraints by downstream SMT solvers. For example, while the expression $c_0 \vee (\neg c_0 \wedge \neg c_1)$ is logically equivalent to $c_0 \vee \neg c_1$, performing such simplifications reliably

Variable	Mutually exclusive guards (MEG)	Ordered guards (ORG)
`lst` (merged)	$\begin{cases} [1] & \text{if } c_0 \\ [2,3] & \text{if } \underline{\neg c_0} \wedge c_1 \\ [1,3,4] & \text{if } \neg c_0 \wedge \neg c_1 \end{cases}$	$\begin{cases} [1] & \text{if } c_0 \\ [2,3] & \text{else if } c_1 \\ [1,3,4] & \text{else} \end{cases}$
`res` (pareval)	$\begin{cases} 1 & \text{if } c_0 \\ 2 & \text{if } \neg c_0 \wedge c_1 \\ 1 & \text{if } \neg c_0 \wedge \neg c_1 \end{cases}$	$\begin{cases} 1 & \text{if } c_0 \\ 2 & \text{else if } c_1 \\ 1 & \text{else} \end{cases}$
`res` (grouped)	$\begin{cases} 1 & \text{if } c_0 \\ 1 & \text{if } \neg c_0 \wedge \neg c_1 \\ 2 & \text{if } \neg c_0 \wedge c_1 \end{cases}$	$\begin{cases} 1 & \text{if } c_0 \\ 1 & \text{else if } \underline{\neg c_1} \\ 2 & \text{else} \end{cases}$
`res` (merged)	$\begin{cases} 1 & \text{if } c_0 \underline{\vee} (\neg c_0 \wedge \neg c_1) \\ 2 & \text{if } \neg c_0 \wedge c_1 \end{cases}$	$\begin{cases} 1 & \text{if } c_0 \underline{\vee} \neg c_1 \\ 2 & \text{else} \end{cases}$

Figure 1: Two union representations: mutually exclusive guards (MEG) versus ordered guards (ORG), for the program in listing 8. For each column, we start with the merged representation of the `lst` value, and then show the partial evaluation (pareval) of `res` under the respective representation. Then, we show conceptual reordering (grouped) to group identical values before final merging (merged). Underlined subterms indicate where guards are expanded or transformed during normalization to maintain the respective representation's invariants. In the ORG column, the guard for the second branch is c_1, which is evaluated only when c_0 is false. For ORG, the guard $\neg c_1$ for the second 1 in `res` (grouped) arises because, within the context where c_0 is false, this 1 was originally the final "else" case after the "else if c_1" branch. The final merged guard $c_0 \vee \neg c_1$ for 1 in ORG's `res (merged)` is logically equivalent to $c_0 \vee (\neg c_0 \wedge \neg c_1)$.

and efficiently across arbitrarily complex guards is a difficult task. It often requires computational effort comparable to an SMT query itself. Although heuristic simplification techniques exist [29, 51], they may prove insufficient for the deeply nested or highly convoluted conditions that can easily arise, particularly since general-purpose symbolic evaluators often lack the domain-specific knowledge to enable more aggressive simplifications. Consequently, a union representation should ideally minimize the growth of path conditions during its normalization and merging process.

These issues are exacerbated in a system like Grisette, where, due to the embedding of effects, the number of distinct values within a union can be much larger than in systems relying on global mutable states. Each distinct outcome, even if it represents a subtle variation of a thrown error, must be distinctly represented in a MEG union with its own mutually exclusive guard. This naturally leads to more complex guards and greater normalization overhead as the number of branches increases.

This limitation becomes particularly apparent when considering features like rich error provenance, as supported by Grisette (discussed in Section 3.3.1). Suppose we have a sequence of operations that can throw different errors, represented conceptually as:

```
-- Throw Err1 if 'a' is false, Err2 if 'b' is false (given 'a'), etc.
do
  condThrow Err1 a
  condThrow Err2 b
  condThrow Err1 c
```

In a MEG representation, to track each specific error type distinctly, each would require its own mutually exclusive path condition. Maintaining this can lead to MEG values like the following (showing the *conceptual* intermediate states after each step):

```
-- Intermediate steps showing MEG construction:
do
  -- {true → Right ()}
  condThrow Err1 a
  -- {¬a ↦ Err1, a ↦ Right ()}
  condThrow Err2 b
  -- {¬a ↦ Err1, a∧¬b ↦ Err2, a∧b ↦ Right ()}
  condThrow Err1 c
  -- Before grouping and merging:
  -- { ¬a ↦ Err1,
  --   a∧¬b ↦ Err2,
  --   a∧b∧¬c ↦ Err1,
  --   a∧b∧c ↦ Right ()
  -- }
  -- After grouping and merging:
  -- {¬a∨(a∧b∧¬c) ↦ Err1, a∧¬b ↦ Err2, a∧b∧c ↦ Right ()}
```

If one wishes to query for the conditions under which *any* error occurs, it requires forming the disjunction of the guards for all error states. In the final merged state above, this would be $\neg a \vee (a \wedge b \wedge \neg c) \vee (a \wedge \neg b)$. While this is logically equivalent to $\neg a \vee \neg b \vee \neg c$, it is syntactically more complex. As the number of potential error types and successful outcomes grows, constructing queries about specific error categories or overall failure conditions by disjoining these complex, mutually exclusive guards becomes inefficient.

Furthermore, the MEG representation, by treating each distinct guard-value pair in a flat list, does not readily facilitate querying or merging based on shared structural properties of the values themselves. For example, different error values might share common structural elements (e.g., belonging to the same error category) while differing in details (e.g., source location).

MEG's flat structure makes it difficult to leverage this shared structure. Performing a query like "did any file-related error occur?" might involve iterating through all error entries and checking their type, rather than exploiting a potentially hierarchical representation where file-related errors could be grouped. This "flat structure" characteristic hinders the ability to use the internal structure of ADTs or other complex values for more intelligent, hierarchical merging or for efficiently querying conditions related to partial properties of the stored values.

4.1.3 *The ORG Representation: Ordered Guards with Normalized Structure*

Given the limitations of the MEG representation, particularly concerning guard complexity and inflexibility with structured data, Grisette adopts an alternative approach: *ordered guards*. This representation avoids normalizing guards into an explicitly mutually exclusive form upfront. Instead, the different conditional values within a symbolic union are organized in an ordered sequence, similar to an `if-elif-...-else` structure. The selection semantics are determined by this order: a value is chosen if its associated guard holds *and* all preceding guards in the sequence are false. We call this representation ORG, for *Ordered Guards*.

Consider again the example from listing 8 and its representation in the ORG column of Figure 1. The merged representation of `lst` directly mirrors the nested conditional structure of its definition:

- if c_0: $[1]$
- else if c_1: $[2, 3]$
- else: $[1, 3, 4]$

Here, the guards c_0 and c_1 are allowed to be logically overlapping. The ordering dictates that the $[2, 3]$ branch is only taken if c_1 is true *and* c_0 is false, and the mutual exclusivity is provided implicitly by the order. This structure avoids the need to immediately introduce negated terms into the guards stored within the representation solely to express mutual exclusivity, potentially keeping the initial representation more compact than MEG's.

However, representing unions in this ordered manner does not, by itself, eliminate the challenges associated with achieving a fully merged, canonical form. Although ORG avoids making guards mutually exclusive explicitly in the structure, transformations to the guards can still occur during the necessary normalization of the union to merge outcomes. Normalization requires grouping values that should be merged, which are typically

identical values or structurally compatible values. If these values are not already adjacent in the ordered sequence, they must be reordered first. This reordering process requires adjusting path conditions associated with the values to maintain logical correctness. For example, looking at the evaluation of `res` in the ORG column of Figure 1, the initial partial evaluation result (`res` (pareval)) has the value 1 appearing under the condition c_0 and also as the final `else` case. To merge these two instances, the second 1 must be grouped with the first. The step `res` (grouped) illustrates the result of this conceptual reordering: the second 1 is moved up and now associated with the guard $\neg c_1$ (within the implicit context where c_0 is false). In general, the further a value needs to move during reordering to group it with compatible values, the more its guard may grow. If arbitrary reordering were frequently necessary, the potential benefit of ORG's initially simpler guard representation could be diminished by the complexity introduced during these normalization steps.

This reveals a potential trade-off: MEG grows guards substantially to enforce mutual exclusivity upfront, while a naive ORG might grow guards during the reordering required for merging. The key insight behind Grisette's ORG design is that the cost and complexity of reordering can be largely avoided by enforcing and maintaining a *canonical sorted order* for all elements within every ORG union.

Our intuition to make ORG efficient is to reduce the need for reordering by ensuring that elements within an ORG union are always kept sorted according to a consistent, type-specific criterion. This canonical order is defined by the *type-directed merging strategy* associated with the Haskell type `a` stored within the `Union a`, This strategy is provided by its `Mergeable` instance, a concept introduced in Section 2.3.3.1 and detailed in Section 3.4.3. Since all ORG unions of a given type adhere to the same sorting convention defined by their strategy, merging two such unions (e.g., the `then` and `else` branches from an `mrgIf` operation) can be performed efficiently using a *mergesort-style algorithm*. This is illustrated conceptually in Figure 2d.

This algorithm traverses the two already-sorted input unions and constructs a new sorted union. It combines the guards for elements deemed compatible by the type's merging strategy, and the results are kept sorted according to the canonical order. This merge operation typically runs in linear time with respect to the number of branches in the input unions. Crucially, because it operates on pre-sorted inputs and avoids arbitrary reordering, it tends to generate more compact resulting path conditions compared to MEG's disjunction-based approach, which must operate on explicitly mutually exclusive (and thus potentially more complex) guards. The ORG algorithm ensures that the resulting union is itself fully normal-

ized (i.e., all compatible elements according to the strategy are merged) and sorted, upholding the crucial *Hierarchical Merging Invariant* (detailed in Section 4.3). Figures 2a to 2c illustrate the properties of a valid, normalized ORG union versus invalid states.

$$\begin{cases} 1 & \text{if} & c_0 \\ 2 & \text{else if} & c_1 \\ 3 & \text{else} & \end{cases} \qquad \begin{cases} 1 & \text{if} & c_0 \\ 1 & \text{else if} & c_1 \\ 3 & \text{else} & \end{cases} \qquad \begin{cases} 1 & \text{if} & c_0 \\ 3 & \text{else if} & c_1 \\ 2 & \text{else} & \end{cases}$$

(a) Valid ORG (sorted, fully merged)

(b) Invalid ORG (contains adjacent identical values)

(c) Invalid ORG (values 3 and 2 are not sorted)

$$\text{mrgIf}\left(c, \begin{cases} 1 & \text{if} & c_0 \\ 2 & \text{else if} & c_1 \\ 4 & \text{else} & \end{cases}, \begin{cases} 1 & \text{if} & c_2 \\ 3 & \text{else if} & c_3 \\ 4 & \text{else} & \end{cases}\right) = \begin{cases} 1 & \text{if} & \text{ite}(c, c_0, c_2) \\ 2 & \text{else if} & c \wedge c_1 \\ 3 & \text{else if} & \neg c \wedge c_3 \\ 4 & \text{else} & \end{cases}$$

(d) Mergesort-style merge of two sorted ORG unions via `mrgIf`.

Figure 2: Properties of ORG unions (a-c) and an illustration of the mergesort-style merge procedure (d). Values (1, 2, 3, 4) are assumed sorted by a type-directed strategy. The merge procedure (d) efficiently combines already sorted unions, preserving sortedness and normalized form.

Furthermore, GRISETTE's ORG representation is not merely a flat, sorted list. It is inherently *hierarchical*, allowing ORG unions to be nested within each other. This structure is particularly advantageous when dealing with complex data types like ADTs, tuples, or records. Instead of treating a complex value as an opaque unit, the merging process can leverage the type's structure. If two complex values share the same top-level structure (e.g., they are pairs, records, or the same ADT constructor), the merging strategy can recursively merge their corresponding sub-components. Consider merging two unions containing pairs, represented abstractly:

$$\text{mrgIf}\left[s, c, \begin{cases} t_1 & \text{if} & c_1 \\ v_3 & \text{else if} & c_2 \\ v_4 & \text{else} & \end{cases}, \begin{cases} t_1' & \text{if} & c_3 \\ t_2' & \text{else if} & c_4 \\ v_4' & \text{else} & \end{cases}\right]$$

$$= \begin{cases} \text{mrgIf}(s', c, t_1, t_1') & \text{if} & \text{ite}(c, c_1, c_3) \\ t_2' & \text{else if} & \neg c \wedge c_4 \\ v_3 & \text{else if} & c \wedge c_2 \\ f(c, v_4, v_4') & \text{else} & \end{cases}$$

Here, s represents the overall merging strategy. It identifies that t_1 and t_1' are sub-trees of structurally compatible values and generates a sub-strategy

s' to recursively merge them. Similarly, it identifies v_4 and v_4' as mergeable base cases and applies a merging function `f` (derived from their `Mergeable` instance). The distinct elements v_3 and t_2' are interleaved according to the canonical order. This hierarchical merging, guided by the type-specific strategies defined in `Mergeable` instances, allows ORG to handle structured data much more effectively than flat representations. It naturally supports sharing of common sub-structures and their guards, leading to more compact representations and potentially faster merging operations. We describe the hierarchical merging algorithm and the invariant it maintains in detail in Section 4.3 and Section 4.4.

While the idea of using ordered, nested if-elif-else structures for representing merged paths has appeared in prior symbolic execution work (e.g., VERITESTING [5], JAVA RANGER [57], and Sinha [59]), these systems typically focused on generic term optimizations (like simplifying unreachable branches) or employed limited, heuristic rewriting rules for merging. They generally did not enforce a rigorous, globally consistent normal form based on a canonical sorted order, nor did they provide a mechanism for achieving complete merging of all structurally compatible values according to extensible, type-directed strategies. GRISETTE's ORG representation, by incorporating these principles—enforced sortedness, the Hierarchical Merging Invariant, type-directed strategies, and hierarchical structure—aims to provide a more systematic and complete approach to normalization and merging within symbolic unions.

The combination of the ordered nature and the hierarchical structure, enabled by type-directed merging strategies, contributes significantly to ORG's effectiveness. Our empirical study suggests that this approach can lead to substantially smaller symbolic formulas compared to prior systems. For example, ORG can generate formulas up to 91.2% smaller than ROSETTE 4 [49] and 79.2% smaller than an internal GRISETTE baseline using MEG on certain benchmarks (see Section 4.6). These improvements pave the way for more efficient SMT solving and enhance the scalability of symbolic evaluation, particularly for the complex value structures inherent in GRISETTE's functional design.

4.2 THE ORG DATA STRUCTURE

Having motivated the need for a novel symbolic union representation suitable for GRISETTE's functional setting, we now introduce its core data structure: the ORG (Ordered Guards) representation. In GRISETTE, the ORG representation is concretely implemented by the `UnionBase` Haskell data type, which forms a nested if-then-else tree:

```haskell
data UnionBase a = Single a | If SymBool (UnionBase a) (UnionBase a)
```

While Section 4.5.2 will describe the user-facing `Union` type, which extends `UnionBase` to integrate with Haskell's `do`-notation for monadic programming with merging, this chapter primarily focuses on the internal `UnionBase` structure. For conciseness within this chapter, we will not distinguish between `UnionBase` and `Union` when context allows, and often refer to `UnionBase` as `Union`.

A `Union` α value is either:

- A `Single` leaf node, containing a single value v of type α. This value v can itself be a complex symbolic term (like a `SymInteger` expression), a data structure, or even another union (if α is, for example, `Union` β).
- An `If` internal branching node. It contains a symbolic Boolean guard (`SymBool`), a then branch (itself a `Union` α), and an else branch (also a `Union` α).

The ORG tree structure directly mirrors the ordered, if-elif-else semantics described previously. The right spine of this tree (formed by repeatedly taking the else branch of `If` nodes) represents the ordered sequence of outcomes. For instance, the mathematical notation for an ORG union corresponds to the Haskell `Union` structure as follows:

```
If     c0  (Single [a])
   (If c1  (Single [b,c])
           (Single [a,c,d]))
```

versus

$$\begin{cases} [a] & \text{if} & c_0 \\ [b,c] & \text{else if} & c_1 \\ [a,c,d] & \text{else} & \end{cases}$$

The recursive nature of the branches (both being `Union` α, not just α) is key to ORG's ability to represent nested conditional structures and hierarchical data. This is essential for merging complex types effectively, as motivated in Section 4.1. An ORG union can contain other ORG unions (sub-trees) as outcomes within its branches. For example, consider the following structure:

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

In this structure, v_2, v_3, v_{11}, v_{12} and v_{13} are leaf values in `Single` nodes, while t_1 represents a nested ORG sub-tree (itself an `If` node). If the top-level condition c_1 is true, the outcome is further determined by the conditions c_{11} and c_{12} within the sub-tree t_1.

The effectiveness of the ORG representation critically depends on the invariants maintained by GRISETTE's merging algorithm. The next subsection (Section 4.3) introduces the *Hierarchical Merging Invariant*. This invariant governs the arrangement and merging of the values and sub-trees within the ORG structure, ensuring they are kept in a canonical sorted order according to type-directed merging strategies. While the construction and management of the symbolic Boolean guards are integral to the merging algorithm, the discussion of the invariant will primarily focus on the hierarchical ordering and merging of the contained values and sub-trees. The details of guard construction will be covered in Section 4.4.

To simplify the subsequent discussion, especially when discussing the representation invariant where the specific guards are not the primary focus, we introduce a compact list notation. This notation represents the sequence of outcomes (sub-trees or leaf values) found along the *right spine* of an ORG union. The example structure above would be represented as $[t_1, v_2, v_3]$ along its top-level right spine, where the sub-tree t_1 itself corresponds to the sequence $[v_{11}, v_{12}, v_{13}]$ along its own right spine. We denote the set of all leaf values inside the `Single` constructors as the *leaf values* of an ORG union u, denoted as $\text{leaves}(u)$. For the example above, $\text{leaves}(u) = \{v_{11}, v_{12}, v_{13}, v_2, v_3\}$.

4.3 HIERARCHICAL MERGING INVARIANT

To achieve complete and efficient merging with ORG semantics, GRISETTE's merging algorithm normalizes unions to maintain a *Hierarchical Merging Invariant*. This invariant is the key to ORG's efficiency. It ensures that values within an ORG union are kept hierarchically sorted according to type-specific criteria. This consistent ordering enables efficient mergesort-style algorithms when merging ORG unions, as introduced conceptually in Section 4.1. This section builds the intuition for this invariant by examining its application to various Haskell types, from simple ones like symbolic primitives and concrete integers, to more complex structures like product and sum types (algebraic data types, ADTs). The formal merging algorithm that maintains this invariant is detailed in Section 4.4.

The primary operator that performs merging is mrgIf. It takes a symbolic Boolean condition, a `then` branch `Union`, and an `else` branch `Union`. It merges the two branches according to the condition. The type signature of mrgIf is:

```
mrgIf :: SymBool → Union α → Union α → Union α
```

The two branches must be of the same type `Union` α, and should satisfy the Hierarchical Merging Invariant with respect to α (in Section 4.5.1 we will show how we ensure correctness even when the invariant is not strictly enforced). The mrgIf operation then produces a new `Union` α that also satisfies the invariant.

Table 1: Typical ORG union representations under the Hierarchical Merging Invariant.

Category	Typical Type	Typical Representation (Conceptual)
Symbolic primitive	`SymBool`	A single symbolic value v (e.g., $\text{ite}(c, b_1, b_2)$). The union is always a single node.
Concrete primitive	`Int`	An ordered list of distinct values along the right spine of the ORG tree: $[v_1, v_2, \ldots, v_n]$ where $v_1 < v_2 < \cdots < v_n$.
Product type	`(Int, Int, SymBool)`	A hierarchically sorted list (the right spine of the ORG tree) of sub-trees: $[t_1, t_2, \ldots, t_n]$. Each t_i contains tuples that share the same first element f_i, with $f_1 < f_2 < \ldots < f_n$. Within each t_i, tuples are further sorted by their second element, and symbolic third elements are merged using ite.
Sum type	`Either SymBool Int`	A hierarchically sorted list (the right spine of the ORG tree) of sub-trees. Each sub-tree corresponds to a data constructor (e.g., [`Left`-subtree, `Right`-subtree], sorted by constructor definition order). Values within each sub-tree are then merged according to the types of that constructor's fields.
Complex type	`HashMap` (Section 3.4.3)	A hierarchically sorted list based on user-defined sorting criteria and merging logic.

Table 1 provides an overview of how this invariant typically applies to various categories of Haskell types when they are stored in an ORG union. The key idea is that the ORG structure is not a flat list but a tree, where the "list" of sub-trees is formed by following the right (`else`) branches of `If` nodes. This right-spine is kept sorted according to a type-specific criteria.

4.3.1 *Merging Symbolic Primitive Types: The Simple Mergeable Types*

For types that are directly representable in SMT logic and can be merged using the SMT ite (if-then-else) operator, such as `SymBool` or `SymInteger`, Grisette's merging strategy always collapses an ORG union into a single value. For example, merging two `Union SymBool` values, `Single b` and `Single c`, under a condition `a` results in:

```
mrgIf a (Single b) (Single c) ⇒ Single (symIte a b c)
```

Here, symIte is GRISETTE's function for creating the SMT-native ite term. The invariant for such types is that a `Union` of these types must always be a `Single` node.

This principle extends to any type α for which a user defines a *simple merging function* that behaves like `symIte`. We call this function mrgIte, which has the following type:

$$\text{mrgIte} :: \texttt{SymBool} \to \alpha \to \alpha \to \alpha$$

Types that support such a merging function are termed *simple mergeable*. Their `Union` representation always satisfies the invariant by being a `Single` value, containing the result of applying mrgIte to combine all conditional outcomes.

GRISETTE provides a type class `SimpleMergeable` for such types:

```
class SimpleMergeable a where
  mrgIte :: SymBool → a → a → a
```

Notable simple mergeable types include tuples of simple mergeable types and unions of mergeable types (the `Mergeable` class will be introduced later in this section):

```
instance
  (SimpleMergeable a, SimpleMergeable b) ⇒ SimpleMergeable (a, b)
  where
  mrgIte c (a1, b1) (a2, b2) = (mrgIte c a1 a2, mrgIte c b1 b2)

-- We will introduce the Mergeable class later.
-- Nested Union types will get merged using mrgIf
instance Mergeable a ⇒ SimpleMergeable (Union a) where
  mrgIte = mrgIf -- Merging two Union results in a new Union
```

4.3.2 *Merging Concrete Types: The Sorted Invariant*

GRISETTE distinguishes between symbolic primitive types (like `SymInteger`) and their concrete Haskell counterparts (like `Integer`). A `Union Integer` represents a set of concrete `Integer` values, each guarded by a path condition. This distinction is useful for controlling concretization, as discussed in Section 3.4.2.

While `SymInteger` values merge into `ite` expressions, a `Union Integer` merges by combining identical concrete integers under combined path conditions. A naive approach might involve pairwise comparisons of all N

possible integer values in two unions, which is inefficient ($O(N^2)$). Sorting the integers first ($O(N \log N)$) is better, but dynamically reordering values within an ORG tree during each merge can be costly due to guard reconstruction Section 4.1. Instead, Grisette's ORG algorithm maintains the invariant that all `Union Integer` inputs to a merge operation are already sorted. This allows merging with a linear-time mergesort-style algorithm.

Definition 4.3.1 (Sorted Invariant (for concrete primitive types))**.** A union u for type α, where α is a concrete primitive type with a total order, meets the *Sorted Invariant* if its leaf values are organized along the right spine of the ORG tree as an ordered sequence $[v_1, v_2, \ldots, v_n]$ such that $v_1 < v_2 < \ldots < v_n$. Each v_i is a distinct concrete value of type α.

$$t_l = \begin{cases} 1 & \text{if} & c_1 \\ 2 & \text{else if} & c_2 \\ 5 & \text{else} \end{cases} \quad t_r = \begin{cases} 2 & \text{if} & c_3 \\ 3 & \text{else if} & c_4 \\ 4 & \text{else} \end{cases}$$

(a) Two ORG unions t_l and t_r to merge under condition c (i.e., to compute $\text{mrgIf}(c, t_l, t_r)$).

$$t_l \equiv \begin{cases} 1 & \text{if} & c_1 \\ 2 & \text{else if} & c_2 \\ \text{undef}_3 & \text{else if} & \bot \\ \text{undef}_4 & \text{else if} & \bot \\ 5 & \text{else} \end{cases} \quad t_r \equiv \begin{cases} \text{undef}_1 & \text{if} & \bot \\ 2 & \text{else if} & c_3 \\ 3 & \text{else if} & c_4 \\ 4 & \text{else if} & \top \\ \text{undef}_5 & \text{else} \end{cases}$$

(b) Conceptual alignment of t_l and t_r for merging. undef_i indicates value i is not present in that path, guarded by $\bot$ (false). For t_r, value 4 is in the else case, so its effective guard for this alignment is $\top$ (true) once 2 and 3 are not chosen.

$$\text{mrgIf}(c, t_l, t_r) = \begin{cases} 1 & \text{if} & \text{ite}(c, c_1, \bot) \equiv c \wedge c_1 \\ 2 & \text{else if} & \text{ite}(c, c_2, c_3) \\ 3 & \text{else if} & \text{ite}(c, \bot, c_4) \equiv \neg c \wedge c_4 \\ 4 & \text{else if} & \text{ite}(c, \bot, \top) \equiv \neg c \\ 5 & \text{else} \end{cases}$$

(c) Merging result.

Figure 3: Example of merging two sorted ORG unions of concrete integers.

Figure 3 illustrates merging two ORG unions of concrete integers, t_l and t_r, under a condition c, to produce $\text{mrgIf}(c, t_l, t_r)$. Both t_l and t_r are already sorted. The merging process conceptually aligns these sorted lists (Figure 3b), padding with conceptually undefined (undef) entries where

necessary, and then combines corresponding entries. For example, the value 1 only appears in t_l (when c_1 holds), so in the merged result, 1 appears if $c \wedge c_1$. The value 2 appears in both t_l (if c_2) and t_r (if c_3), so in the merged result, 2 appears if $\text{ite}(c, c_2, c_3)$ (assuming 1 is not chosen by its condition). The resulting union (Figure 3c) is also sorted, maintaining the invariant. The conditions are constructed using ite based on c and the original conditions from t_l and t_r. This process will be detailed in Section 4.4.

4.3.3 *Hierarchical Merging Invariant for ADTs*

The Hierarchical Merging Invariant enables efficient handling of algebraic data types (ADTs), including product types (tuples, records) and sum types (like `Either` or `Maybe`).

Consider a product type like `(Integer, Integer, SymBool)`. A naive "flat list" approach would sort tuples lexicographically based on their concrete fields, then merge tuples with identical concrete fields by applying mrgIte to their symbolic fields. In this scheme, all leaf values (tuples) are kept as a flat list along the right spine of the ORG tree. This can be inefficient for several reasons. If the list of tuples becomes very long, traversing it repeatedly for merging can be costly. Furthermore, if there are structural differences based on initial components of the tuples—for instance, if one union being merged contains many tuples starting with the integer 2 (e.g., $(2, 1, a), (2, 2, b)$) while the other union contains no tuples starting with 2—a flat approach would need to modify the guard for each of the $(2, *, *)$ tuples from the first union. This modification reflects that they are chosen only if that union is selected, potentially introducing many terms to the guards.

Grisette's ORG representation uses its hierarchical tree structure to handle such cases more effectively. Values are grouped into sub-trees based on a primary sorting key (e.g., the first element of a tuple). These sub-trees are then recursively merged and sorted based on secondary keys, and so on. For `(Integer, Integer, SymBool)`, the ORG union would first group tuples by their first `Integer` field. Each group forms a sub-tree along the right spine of the ORG tree. Within each such sub-tree, tuples are then internally sorted by the second `Integer` field. Finally, tuples within these sub-sub-groups (sharing the same first two concrete fields) would have their `SymBool` fields merged using ite (as `SymBool` is simple mergeable).

The hierarchical structure is particularly beneficial for what we call *sparse merges*. This occurs when a sub-tree corresponding to a particular primary key (e.g., all tuples starting with 2) is present in one union being merged but entirely absent in the other. In such cases, the ORG merge algorithm can incorporate the entire sub-tree from the first union into the result un-

der a single conditional guard derived from the top-level merge condition. It avoids iterating through the internal structure of this sub-tree merely to introduce the top-level condition to all its internal guards. This ability to treat entire sub-trees as atomic units when appropriate can lead to performance gains (potentially sub-linear complexity for merging) and results in more compact SMT formulas (a single more complex guard instead of many simpler guards with repeated but not shared structures).

Figure 4 illustrates this hierarchical merging for ORG unions of the tuple type `(Integer, Integer, SymBool)`. The unions t_l and t_r are merged under condition c. The algorithm first compares the primary sorting key (the first `Integer` field).

- Both t_l and t_r have sub-trees corresponding to the first field being 1 ($t_{l,1}$ and $t_{r,1}$). These are recursively merged.
- t_l has a sub-tree for the first field being 2 ($t_{l,2}$), while t_r does not. The hierarchical merge incorporates $t_{l,2}$ into the result, guarded by $c \wedge c_{l,2}$ (the condition for choosing t_l and then $t_{l,2}$ within t_l). The internal structure of $t_{l,2}$ is preserved under this single new guard.
- Both t_l and t_r have sub-trees for the first field being 9 ($t_{l,9}$ and $t_{r,9}$). These are recursively merged.

The recursive merge for $t_{l,1}$ and $t_{r,1}$ (both with first field 1) proceeds by comparing their second `Integer` fields. Both $(1, 1, v_{l,1,1})$ from $t_{l,1}$ and $(1, 1, v_{r,1,1})$ from $t_{r,1}$ have 1 as the second field, so their third `SymBool` components are merged with ite. The tuple $(1, 2, v_{l,1,2})$ from $t_{l,1}$ has no counterpart in $t_{r,1}$ and is included accordingly, guarded by the condition c and then the specific path within $t_{l,1}$. The merge for $t_{l,9}$ and $t_{r,9}$ follows a similar logic. The conditions for each resulting tuple in the final merged ORG are constructed based on c and the original conditions from t_l and t_r.

To formalize this hierarchical merging, we define *merging strategies*.

Definition 4.3.2 (Merging Strategy)**.** For a given type α, a *merging strategy* s :: `MergingStrategy` α specifies how unions of α should be merged. It is one of the following:

1. A *simple strategy* is defined by an ite-like merging function for a subset of values $S \subseteq \alpha$: partialIte :: `SymBool` $\to S \to S \to S$ (there isn't a subset type in Haskell, but for notational convenience, we use S to denote that the merging function is only defined on S). It applies when values in S can always be combined into a single value in S under a symbolic condition using partialIte.

$$t_l = \begin{cases} t_{l,1} & \text{if} \quad c_{l,1} \\ t_{l,2} & \text{else if} \quad c_{l,2} \\ t_{l,9} & \text{else} \end{cases} \qquad t_r = \begin{cases} t_{r,1} & \text{if} \quad c_{r,1} \\ t_{r,9} & \text{else} \end{cases}$$

where

$$t_{l,1} = \begin{cases} (1,1,v_{l,1,1}) & \text{if} \quad c_{l,1,1} \\ (1,2,v_{l,1,2}) & \text{else} \end{cases} \qquad t_{r,1} = (1,1,v_{r,1,1})$$

$$t_{l,2} = \begin{cases} (2,3,v_{l,2,3}) & \text{if} \quad c_{l,2,3} \\ (2,4,v_{l,2,4}) & \text{else if} \quad c_{l,2,4} \\ (2,5,v_{l,2,5}) & \text{else} \end{cases} \qquad t_{r,2} \text{ does not exist}$$

$$t_{l,9} = (9,2,v_{l,9,2}) \qquad t_{r,9} = (9,1,v_{r,9,1})$$

(a) Two ORG unions, t_l and t_r, with type `(Integer, Integer, SymBool)`, to be merged under condition c, i.e., $\text{mrgIf}(c, t_l, t_r)$. Each $t_{l/r,i}$ is a sub-tree where the first element of all contained tuples is i. $v_{l/r,i,j}$ are `SymBool` values.

$$\text{mrgIf}(c, t_l, t_r) = \begin{cases} \text{mrgIf}(c, t_{l,1}, t_{r,1}) & \text{if} \quad \text{ite}(c, c_{l,1}, c_{r,1}) \\ t_{l,2} & \text{if} \quad \text{ite}(c, c_{l,2}, \bot) = c \wedge c_{l,2} \\ \text{mrgIf}(c, t_{l,9}, t_{r,9}) & \text{else} \end{cases}$$

$$\text{mrgIf}(c, t_{l,1}, t_{r,1}) = \begin{cases} (1,1,\text{ite}(c, v_{l,1,1}, v_{r,1,1})) & \text{if} \quad \text{ite}(c, c_{l,1,1}, c_{r,1,1}) \\ (1,2,v_{l,1,2}) & \text{else} \end{cases}$$

$$\text{mrgIf}(c, t_{l,9}, t_{r,9}) = \begin{cases} (9,2,v_{l,9,2}) & \text{if} \quad \text{ite}(c, \bot, \top) = \neg c \\ (9,1,v_{r,9,1}) & \text{else} \end{cases}$$

(b) Result of merging $\text{mrgIf}(c, t_l, t_r)$. The merge proceeds hierarchically. For example, sub-trees for key 1 ($t_{l,1}$, $t_{r,1}$) are themselves merged. The sub-tree for key 2 ($t_{l,2}$) is included as is, as it only exists in t_l.

Figure 4: Example of hierarchically merging two ORG unions of tuples.

2. A *sorted strategy* is defined by a totally ordered index type ι, an indexing function $\text{index} :: \alpha \to \iota$, and a sub-strategy function $\text{sub} :: \iota \to \texttt{MergingStrategy}\ \alpha$. It partitions values v of type α based on their index $\text{index}(v)$, orders these partitions by index along the right spine of the ORG tree, and then applies the appropriate sub-strategy $\text{sub}(\text{index}(v))$ to merge values within each partition.

3. A *no strategy* indicates that values of type α (or values within a certain subset of α) should not be structurally merged. They will be kept separate in the ORG union, each under its own path condition.

For notational convenience, we may write $\text{partialIte}(s)(c, v_t, v_f)$ when applying functions like partialIte, index, or sub that are part of a merging strategy s, to mean applying the specific function extracted from strategy s to arguments c, v_t, and v_f. If the strategy is clear from context, we may omit s and write $\text{partialIte}(c, v_t, v_f)$.

The corresponding Haskell GADT for `MergingStrategy` is:

```haskell
data MergingStrategy a where
  SimpleStrategy ::
    -- The merging function.
    (SymBool → a → a → a) →
    MergingStrategy a
  SortedStrategy ::
    -- MergingIndex is a constraint requiring 'idx' to have
    -- * Ord (for total ordering), and
    -- * Typeable (for runtime type information, an internal
    --   implementation detail).
    -- Most concrete types satisfy MergingIndex.
    (MergingIndex idx) ⇒
    -- The indexing function.
    (a → idx) →
    -- The sub-strategy function.
    (idx → MergingStrategy a) →
    MergingStrategy a
  NoStrategy :: MergingStrategy a
```

A sorted strategy thus enables hierarchical merging. The process repeats, applying sub-strategies, until a simple strategy (for actual value merging) or no strategy (to stop merging at that level) is reached.

For the tuple type `(Integer, Integer, SymBool)`, the strategy is:

```haskell
SortedStrategy (\(x,_,_) → x) (\_ →   -- Index by first Integer
  SortedStrategy (\(_,y,_) → y) (\_ → -- Index by second Integer
    -- Merge third SymBool field using its SimpleStrategy
    -- (mrgIte for SymBool)
    SimpleStrategy (\c (a,b,u) (_,_,v) → (a,b,mrgIte c u v))))
```

For a merging strategy to be well-behaved, it must be *proper* for the set of values $S \subseteq \alpha$ it operates on (where α is the Haskell type). This means it must terminate (have finite depth) and correctly handle all values in S. The depth of a strategy is defined as follows:

Definition 4.3.3 (Strategy Depth)**.** The depth $\text{depth}(s)$ of a merging strategy s is a natural number:

- A simple or no strategy has depth 0.
- A sorted strategy has depth $1 + \max_{i \in \{\text{index}(s)(v) \mid v \in S\}} \text{depth}(\text{sub}(s)(i))$.

Definition 4.3.4 (Proper Merging Strategy)**.** A merging strategy s for type α is a *proper* regarding a set $S \subseteq \alpha$ if it has a *finite depth* (ensuring termination) and one of the following holds:

- s is a simple strategy and meets the *closure requirement* on S.
- s is a sorted strategy and meets the *partitioning requirement* on S.
- s is a no strategy (which is trivially proper).

We use $\text{proper}(s, S)$ to denote that s is proper regarding S.

The closure and partitioning requirements ensure that the strategy can correctly process the values in S:

- **Closure Requirement (for simple strategies)**: A simple strategy s meets the closure requirement on S if S is closed under $\text{partialIte}(s)$; i.e., for any condition b :: `SymBool` and any values $v_1, v_2 \in S$, $v_{\text{merged}} = \text{partialIte}(b, v_1, v_2)$ is defined and $v_{\text{merged}} \in S$.
- **Partitioning Requirement (for sorted strategies)**: A sorted strategy s meets the partitioning requirement on S if its indexing function $\text{index}(s)$ is defined on all elements of S, and for every index $i \in \{\text{index}(s)(v) \mid v \in S\}$, the sub-strategy $\text{sub}(s)(i)$ is itself proper regarding the subset $\{v \mid v \in S \wedge \text{index}(s)(v) = i\}$.

Strategies with cyclic sub-strategy relations (e.g., a strategy being a sub-strategy of itself directly or indirectly) have infinite depth and are not proper.

With these concepts, we can now formally state the *Hierarchical Merging Invariant*:

Definition 4.3.5 (Hierarchical Merging Invariant)**.** A union u containing values of type α meets the *Hierarchical Merging Invariant* regarding a subset $S \subseteq \alpha$ and a merging strategy s (where $\text{proper}(s, S)$ holds) if:

- If s is a simple strategy, then u must be a single value $[v]$ where $v \in S$.
- If s is a no strategy, then u can be any ORG tree whose leaf values are in S. (No further structural merging is enforced at this level.)

- If s is a sorted strategy, then u (viewed as an ordered list of its top-level branches $[u_1, u_2, \cdots, u_n]$ in the right spine) must satisfy:
  1. All leaf values of u are in S: $\text{leaves}(u) \subseteq S$.
  2. For each branch u_k, all its leaf values map to the same index i_k under $\text{index}(s)$: $\forall v \in \text{leaves}(u_k), \text{index}(s)(v) = i_k$.
  3. The indices of the top-level branches are strictly increasing: $i_1 < i_2 < \cdots < i_n$. This ensures sortedness and that no adjacent, unmerged items (with the same index) exist at this level.
  4. Each branch u_k recursively satisfies the Hierarchical Merging Invariant with respect to the sub-strategy $\text{sub}(s)(i_k)$ and the subset $\{v \mid v \in S \wedge \text{index}(s)(v) = i_k\}$. This ensures that sub-trees are also correctly merged and sorted.

We denote the Hierarchical Merging Invariant for the union u with respect to the strategy s and the subset S as $\text{merged}(s, u, S)$.

This invariant ensures that ORG unions are always in a canonical, hierarchically sorted, and fully merged form according to their type's strategy. A type that has such a merging strategy is thus called *mergeable*.

Definition 4.3.6 (Mergeable and simple mergeable types)**.** A type α is *mergeable* if it has a unique *root merging strategy* that is proper for all values of α. A type α is *simple mergeable* if it is mergeable and its root merging strategy is a simple strategy.

Grisette provides Haskell type classes corresponding to these concepts:

```haskell
class Mergeable a where
  rootStrategy :: MergingStrategy a

-- Detailed in next section.
mrgIfWithStrategy ::
  MergingStrategy a → SymBool → Union a → Union a → Union a

-- User-facing mrgIf uses the root strategy.
mrgIf :: (Mergeable a) ⇒ SymBool → Union a → Union a → Union a
mrgIf = mrgIfWithStrategy rootStrategy

-- SimpleMergeable inherits from Mergeable. Its rootStrategy must be
-- equivalent to SimpleStrategy mrgIte.
class (Mergeable a) ⇒ SimpleMergeable a where
  mrgIte :: SymBool → a → a → a
```

For instances, the merging strategies for `SymInteger` and `Integer` are:

```haskell
instance SimpleMergeable SymInteger where
  mrgIte = symIte -- symIte is the Grisette's SMT ite operator

instance Mergeable SymInteger where
  rootStrategy = SimpleStrategy mrgIte

-- Sort by the integer value itself (using 'id' as index function).
-- For each specific integer i, the sub-strategy is simple:
-- if merging 'i' with 'i' under condition 'c', the result is 'i'.
instance Mergeable Integer where
  rootStrategy =
    SortedStrategy id (\_i → SimpleStrategy (\_c t _e → t))
```

Sum types like ADTs are handled by using a tag derived from the data constructor (e.g., an integer or Boolean tag based on declaration order) as the first-level sorting key. Each sub-tree in the ORG union then corresponds to one constructor, and its contents are merged recursively according to the types of that constructor's fields. For example, for the type `Either Integer SymInteger`, we can define the merging strategy (simplified here; in practice, Grisette defines it for the generic `Either a b` type) as:

```haskell
instance Mergeable (Either Integer SymInteger) where
  rootStrategy =
    SortedStrategy
      (\case
          Left _ → False -- Tag for Left constructor
          Right _ → True -- Tag for Right constructor
      )
      (\case
          False →
            -- Sub-strategy for Left (containing Integer).
            -- Derived from the root strategy for Integer.
            SortedStrategy id (\_ → SimpleStrategy (\_ t _ → t))
          True →
            -- Sub-strategy for Right (containing SymInteger).
            -- Derived from the root strategy for SymInteger.
            SimpleStrategy
              (\cond (Right x) (Right y) → Right (mrgIte cond x y))
      )
```

For the type `Union (Either Integer SymInteger)`, a conceptual merged ORG structure might look like:

$$\begin{cases} t_{\texttt{Left}} & \text{if} \quad c_{\texttt{Left}} \\ \texttt{Right}\ x & \text{else} \end{cases} \quad \text{where} \quad t_{\texttt{Left}} = \begin{cases} \texttt{Left}\ 0 & \text{if} \quad c_0 \\ \texttt{Left}\ 2 & \text{else if} \quad c_2 \\ \texttt{Left}\ 5 & \text{else} \end{cases}$$

Here, $c_{\texttt{Left}}$ is a symbolic Boolean that is true if the overall result is any `Left` value. The $t_{\texttt{Left}}$ sub-tree contains all `Left` values, internally sorted by their integer value. The `Right` branches, containing `SymInteger` (which is simple mergeable), are merged into a single `Right` node.

For user-defined ADTs, Grisette can automatically derive `Mergeable` instances using Template Haskell (the preferred method) or via `Generic`-based deriving. For complex non-algebraic types (like the hash maps from Section 3.4.3), users can manually implement `Mergeable` to define custom hierarchical partitioning and merging logic for performance tuning.

4.4 MERGING ALGORITHM SEMANTICS

This section details the ORG merging algorithm. The core of this algorithm is implemented by a function we refer to as mrgIfWithStrategy. As established in Section 4.3, this function takes a merging strategy s for a type α, a symbolic condition c, a then branch ORG union t, and an else branch ORG union f. It produces a new ORG union that satisfies the Hierarchical Merging Invariant (Definition 4.3.5) with respect to s, assuming that the input unions t and f also satisfy the invariant with respect to s. The user-facing mrgIf function (without an explicit strategy argument) is a specialized version that calls mrgIfWithStrategy using the root strategy of the type. For brevity in the following semantic rules, we will just use $\text{mrgIf}(s, c, t, f)$ with an explicit merging strategy s.

In the following sub-sections, we define the semantics using a set of reduction rules, where $\Downarrow$ denotes big-step evaluation to a normalized ORG union.

4.4.1 *Basic Merging Rules*

The basic cases for mrgIf are presented in Figure 5. These rules handle scenarios where the condition c is effectively concrete (i.e., is the symbolic term $\top$ or $\bot$), or where the merging strategy s is either a simple or no strategy.

- Rule mrgIf-True and Rule mrgIf-False: If the condition c is effectively concrete (is the symbolic term $\top$ or $\bot$), mrgIf simply evaluates

$$\frac{}{\mathrm{mrgIf}(s, \top, t, f) \Downarrow t}\ (\textsc{mrgIf-True}) \qquad \frac{}{\mathrm{mrgIf}(s, \bot, t, f) \Downarrow f}\ (\textsc{mrgIf-False})$$

$$\frac{\mathrm{notConc}(c) \quad \mathrm{isSimpleStrategy}(s)}{\mathrm{mrgIf}(s, c, \mathtt{Single}(v_t), \mathtt{Single}(v_f)) \Downarrow \mathtt{Single}(\mathrm{partialIte}(s)(c, v_t, v_f))}\ (\textsc{mrgIf-Simple})$$

$$\frac{\mathrm{notConc}(c) \quad \mathrm{isNoStrategy}(s)}{\mathrm{mrgIf}(s, c, t, f) \Downarrow \mathrm{If}(c, t, f)}\ (\textsc{mrgIf-No})$$

Figure 5: Basic rules for mrgIf. notConc(c) means c is a symbolic Boolean term that is not $\top$ or $\bot$. These rules apply when the strategy s is simple or no strategy, or the condition c is effectively concrete.

to the then branch t or else branch f, respectively. Since t and f are assumed to already satisfy the Hierarchical Merging Invariant with respect to s, the result also satisfies the invariant.

- Rule MRGIF-SIMPLE: If the strategy s is a simple strategy, the input unions t and f must be $\mathtt{Single}(v_t)$ and $\mathtt{Single}(v_f)$ respectively. This is because the Hierarchical Merging Invariant for types whose root strategy is simple requires their ORG unions to be `Single` nodes. The result is thus a `Single` node containing the value obtained by applying the partialIte function to merge v_t and v_f under condition c.
- Rule MRGIF-NO: If c is symbolic and the strategy s is no strategy, no structural merging is performed. The result is a new `If` node. The invariant is trivially satisfied with the no strategy.

4.4.2 *Helper Functions for Sorted Strategy*

When the merging strategy s is a sorted strategy and the condition c is symbolic, the mrgIf algorithm proceeds recursively. This process resembles a mergesort algorithm on a linked list. It deconstructs the input ORG unions t and f along their right spines based on the current strategy s. Viewing the right spine as a linked list of sub-trees, the algorithm identifies the maximal sub-trees where all leaf values share the same index according to s. It then recursively processes these sub-trees by comparing their indices, merges them (recursively with sub-strategies) and reconstructs a new, merged ORG union.

We define several helper functions for this deconstruction and reconstruction. We also define a conceptual marker for the exhausted end of a right spine as SpineTail; it is not part of the `Union` implementation but is used in the semantics to simplify the rules for recursion termination. The first set of helpers, leftMost, allSameIndex, and leftMostIndex (Figure 6), are used to analyze the structure of a union u with respect to a given sorted strategy s.

$$\frac{}{\text{leftMost}(\texttt{Single}(v)) \Downarrow v}\ (\textsc{LeftMostSingle}) \qquad \frac{}{\text{leftMost}(\texttt{If}(c, u_t, u_f)) \Downarrow \text{leftMost}(u_t)}\ (\textsc{LeftMostIf})$$

$$\frac{u \neq \text{SpineTail}}{\text{leftMostIndex}(s, u) \Downarrow \text{index}(s)(\text{leftMost}(u))}\ (\textsc{LeftMostIndex})$$

$$\frac{}{\text{leftMostIndex}(s, \text{SpineTail}) \Downarrow +\infty}\ (\textsc{LeftMostIndexTail})$$

$$\frac{}{\text{allSameIndex}(s, \texttt{Single}(v)) \Downarrow \top}\ (\textsc{AllSameIndexSingle})$$

$$\frac{i_t := \text{leftMostIndex}(s, u_t) \qquad i_f := \text{leftMostIndex}(s, u_f)}{\text{allSameIndex}(s, \texttt{If}(c, u_t, u_f)) \Downarrow i_t = i_f}\ (\textsc{AllSameIndexIf})$$

Figure 6: Rules for analysis helpers: leftMost, allSameIndex, and leftMostIndex. These functions analyze an ORG union u with respect to a sorted strategy s. leftMost finds a representative value for indexing. leftMostIndex retrieves the index of the leftmost leaf value in u. SpineTail is a conceptual marker for an exhausted right spine. allSameIndex determines if u forms a tree of values with the same index.

- leftMost(u): Recursively traverses the then (left) branches of u until it reaches a leaf value, returning that leaf value. This provides a representative value from u that is used by the indexing function index(s) of the strategy s.
- leftMostIndex(s, u): Returns the index (under strategy s) of the leftmost leaf value in the union u. For the sentinel SpineTail, we define leftMostIndex to be $+\infty$. This value is greater than any valid index, ensuring that SpineTail is correctly treated as the end of any sorted sequence of sub-trees.

- allSameIndex(s, u): Determines if all leaf values within the union u belong to the same index group under the current sorted strategy s. It is trivially true if u is a `Single` node (as it contains only one leaf value, hence one index group). For an $\texttt{If}(c, u_t, u_f)$ node, it evaluates to true if and only if the index of leftMost(u_t) is the same as the index of leftMost(u_f) according to s. This check is sufficient due to the Hierarchical Merging Invariant: if u_t and u_f are already normalized and their leftmost representative elements share an index under s, then all leaf values in both u_t and u_f must also belong to that same index group with respect to s. Thus, allSameIndex(s, u) being true implies that the entire union u forms a single index group according to the current level of indexing defined by s.

The next set of helpers, head, tail, and headGuard (Figure 7), deconstruct an ORG union u along its right spine. They use allSameIndex to identify the first maximal sub-tree (the "head") of the spine (treated as a linked list of sub-trees) that forms a single index group. The "tail" is the remainder of the spine after extracting the head, and SpineTail is used here to indicate that a spine has been fully processed. "headGuard" is the path condition for choosing the head.

$$\frac{\text{allSameIndex}(s, u)}{\text{head}(s, u) \Downarrow u}\ (\textsc{HeadSame}) \qquad \frac{\neg\text{allSameIndex}(s, \text{If}(c, u_t, u_f))}{\text{head}(s, \text{If}(c, u_t, u_f)) \Downarrow u_t}\ (\textsc{HeadDiff})$$

$$\frac{\text{allSameIndex}(s, u)}{\text{tail}(s, u) \Downarrow \text{SpineTail}}\ (\textsc{TailSame}) \qquad \frac{\neg\text{allSameIndex}(s, \text{If}(c, u_t, u_f))}{\text{tail}(s, \text{If}(c, u_t, u_f)) \Downarrow u_f}\ (\textsc{TailDiff})$$

$$\frac{\text{allSameIndex}(s, u)}{\text{headGuard}(s, u) \Downarrow \top}\ (\textsc{HeadGuardSame}) \qquad \frac{\neg\text{allSameIndex}(s, \text{If}(c, u_t, u_f))}{\text{headGuard}(s, \text{If}(c, u_t, u_f)) \Downarrow c}\ (\textsc{HeadGuardDiff})$$

Figure 7: Rules for deconstruction helpers: head, tail, and headGuard. These functions deconstruct an ORG union u along its right spine, based on a sorted strategy s. They identify the first maximal sub-tree that forms a single indexed group under s (the head), its effective path condition (the headGuard), and the remainder of the spine (the tail).

- head(s, u): If allSameIndex(s, u) is true, then u itself constitutes a sub-tree where all leaf values share the same index under s and is the head. Otherwise, the head is the then branch of the `If` node.

- tail(s, u): If allSameIndex(s, u) is true, there is no further tail within u (as u is a single indexed sub-tree). Otherwise, the tail is the else branch of the `If` node.
- headGuard(s, u): If allSameIndex(s, u) is true, the headGuard is $\top$, meaning u is taken unconditionally. Otherwise, the headGuard is the condition c of the `If` node.

Finally, the buildIf helper (Figure 8) reconstructs an ORG union. It takes care of the special cases the tails has been exhausted. In this case, the result is the remaining head sub-tree.

$$\frac{u_f = \text{SpineTail}}{\text{buildIf}(c, u_t, u_f) \Downarrow u_t} \quad \text{(BuildIfFalse)}$$

$$\frac{\text{notConc}(c) \qquad u_t \neq \text{SpineTail} \qquad u_f \neq \text{SpineTail}}{\text{buildIf}(c, u_t, u_f) \Downarrow \text{If}(c, u_t, u_f)} \quad \text{(BuildIf)}$$

Figure 8: Rules for reconstruction helper: buildIf. It simplifies when the condition is $\top$ or $\bot$ to avoid introducing the sentinel SpineTail into the final result.

4.4.3 *Merging Rules for Sorted Strategy*

The core recursive merging rules for a sorted strategy is defined in Figure 9. These rules apply when the condition c of mrgIf(s, c, u_t, u_f) is symbolic (notConc(c)) and s is a sorted strategy. The input ORG unions, u_t (the then branch) and u_f (the else branch), are deconstructed using the helper functions from Figure 7. The rules operate by comparing the indices of the "head" sub-trees of u_t and u_f to determine whether they should be merged, or kept as is.

For the rules in Figure 9, we use the following notations:

- $h_t \coloneqq \text{head}(s, u_t)$ and $h_f \coloneqq \text{head}(s, u_f)$ are the head sub-trees of u_t and u_f.
- $t_t \coloneqq \text{tail}(s, u_t)$ and $t_f \coloneqq \text{tail}(s, u_f)$ are their respective tails.
- $i_t \coloneqq \text{leftMostIndex}(s, u_t)$ and $i_f \coloneqq \text{leftMostIndex}(s, u_f)$ are the indices of the leftmost leaf values in u_t and u_f, which is the index of the head sub-trees assuming the trees meet the invariant. (Recall leftMostIndex$(s, \text{SpineTail}) = +\infty$).

- $c_t \coloneqq \text{headGuard}(s, u_t)$ and $c_f \coloneqq \text{headGuard}(s, u_f)$ are the path conditions for these head sub-trees to be chosen from within their original unions u_t and u_f.

$$\frac{\text{notConc}(c) \quad i_t < i_f \quad c_{\text{head}} \coloneqq c \wedge c_t \quad u_{\text{tail}} \coloneqq \text{mrgIf}(s, c, t_t, u_f)}{\text{mrgIf}(s, c, u_t, u_f) \Downarrow \text{buildIf}(c_{\text{head}}, h_t, u_{\text{tail}})} \text{(MrgIfSortedLt)}$$

$$\frac{\text{notConc}(c) \quad i_t > i_f \quad c_{\text{head}} \coloneqq \neg c \wedge c_f \quad u_{\text{tail}} \coloneqq \text{mrgIf}(s, c, u_t, t_f)}{\text{mrgIf}(s, c, u_t, u_f) \Downarrow \text{buildIf}(c_{\text{head}}, h_f, u_{\text{tail}})} \text{(MrgIfSortedGt)}$$

$$\frac{\begin{array}{c}\text{notConc}(c) \qquad i_t = i_f \neq +\infty \qquad c_{\text{head}} \coloneqq \text{ite}(c, c_t, c_f) \\ u_{\text{head}} \coloneqq \text{mrgIf}(\text{sub}(s)(i_t), c, h_t, h_f) \qquad u_{\text{tail}} \coloneqq \text{mrgIf}(s, c, t_t, t_f)\end{array}}{\text{mrgIf}(s, c, u_t, u_f) \Downarrow \text{buildIf}(c_{\text{head}}, u_{\text{head}}, u_{\text{tail}})} \text{(MrgIfSortedEq)}$$

$$\frac{\text{notConc}(c) \quad i_t = i_f = +\infty}{\text{mrgIf}(s, c, t, f) \Downarrow \text{SpineTail}} \text{(MrgIfSortedInf)}$$

Figure 9: Recursive merging rules for $\text{mrgIf}(s, c, u_t, u_f)$ with a sorted strategy s. The notations h_t, t_t, c_t, i_t (and h_f, t_f, c_f, i_f) are defined in the text using the helper functions.

The merging proceeds as follows, very similar to a mergesort algorithm:

- Rule MrgIfSortedLt: If the index of u_t's head (i_t) is less than the index of u_f's head (i_f), then h_t (the head sub-tree of u_t) comes first in the merged sequence, without the need to recurse into it. This is called the sparse merge optimization and may contribute a lot to the merging (by reducing the number of visited nodes) and solving (by reducing the size of SMT terms) performance. Its guard in the resulting ORG tree, c_{head}, becomes $c \wedge c_t$, which is equivalent to $\text{ite}(c, c_t, \bot)$. The rest of the merged union, u_{tail}, is formed by recursively calling mrgIf on the tail of u_t (t_t) and the original u_f, under the same condition c. This rule correctly handles cases where u_f might be SpineTail (making $i_f = +\infty$, so $i_t < i_f$ holds). The u_t cannot be SpineTail in this case.
- Rule MrgIfSortedGt: This rule is symmetric to Rule MrgIfSortedLt.

- Rule MrgIfSortedEq: If the indices are equal and finite, it means the head sub-trees h_t and h_f belong to the same primary index group under strategy s. These two heads are then recursively merged using mrgIf with the sub-strategy $\text{sub}(s)(i_t)$ and the condition c. The remainder of the result, u_{tail}, is formed by recursively merging the tails t_t and t_f with the original strategy s.
- Rule MrgIfSortedInf: A special case occurs when the indices are both $+\infty$. This means that both u_t and u_f are SpineTail. We do not merge and simply return the sentinel SpineTail and let the upper level buildIf handle the case.

4.4.4 *Properties of the Merging Algorithm*

The ORG merging algorithm, as defined by the semantic rules for mrgIf (shorthand for mrgIfWithStrategy) in Section 4.4, have several properties that ensure its correctness and efficiency. We outline these properties in this section. These claims are supported by a mechanized verification in Coq 8.16.0[1].

First, we define the notion of a well-formed invocation of the mrgIf function, which is the preconditions for the algorithm's guarantees.

Definition 4.4.1 (Well-Formed mrgIf Invocation)**.** $\text{mrgIf}(s, c, u_t, u_f)$ for values of type α is *well-formed* if:

1. s is a proper merging strategy (Definition 4.3.4) for type α with respect to the set of all possible values of α ($\text{proper}(s, \alpha)$).
2. $u_t :: \texttt{Union}\ \alpha$ and $u_f :: \texttt{Union}\ \alpha$ are ORG unions that both satisfy the Hierarchical Merging Invariant (Definition 4.3.5) with respect to the strategy s and the set of all possible values of α ($\text{merged}(s, u_t, \alpha)$ and $\text{merged}(s, u_f, \alpha)$).

Given a well-formed invocation, the algorithm is guaranteed to terminate, produce a unique result, and that result will also satisfy the Hierarchical Merging Invariant. Furthermore, the algorithm exhibits efficient performance characteristics.

Before stating these the properties, we first define the size of an ORG union, which is used to prove the termination and complexity of the algorithm.

Definition 4.4.2 (ORG Union Size)**.** The size of a union u, denoted $\text{usize}(u)$, is the number of leaf values in u. Formally:

1 https://github.com/lsrcz/grisette-coq

- $\text{usize}(\text{Single}(x)) = 1$
- $\text{usize}(\text{If}(c, t, f)) = \text{usize}(t) + \text{usize}(f)$

We also define $\text{usize}(\text{SpineTail})$ to be 0. A non-SpineTail union u always has $\text{usize}(u) \geqslant 1$.

Lemma 4.4.1 (Termination). *For any well-formed invocation* $\text{mrgIf}(s, c, u_t, u_f)$*, the evaluation defined by the rules in Section 4.4 terminates.*

Proof Sketch. Termination is proven by induction on a lexicographical measure $(\text{depth}(s), \text{usize}(u_t) + \text{usize}(u_f))$, where $\text{depth}(s)$ is the depth of the strategy s (Definition 4.3.3).

The basic rules (Figure 5) terminate directly. For the sorted strategy rules (Figure 9):

- Rules MrgIfSortedLt and MrgIfSortedGt recurse on a tail of one of the inputs (e.g., t_t in Rule MrgIfSortedLt), strictly reducing the sum of union sizes while strategy depth remains the same.
- Rule MrgIfSortedEq makes two recursive calls. The recursive call $\text{mrgIf}(\text{sub}(s)(i_t), c, h_t, h_f)$ uses a sub-strategy $\text{sub}(s)(i_t)$, which has a strictly smaller depth than s. The other recursive call $\text{mrgIf}(s, c, t_t, t_f)$ uses the same strategy s but operates on tails t_t, t_f, which are structurally smaller than the original u_t, u_f.
- Rule MrgIfSortedInf terminates without recursive calls.

Since proper merging strategies have finite depth and union sizes are non-negative integers, this lexicographical measure is well-founded, implying termination. □

Lemma 4.4.2 (Determinism). *For any well-formed invocation* x *defined by* $x = \text{mrgIf}(s, c, u_t, u_f)$*, if* $x \Downarrow u_1$ *and* $x \Downarrow u_2$*, then* $u_1 = u_2$*.*

Proof Sketch. The semantics rules are syntax-directed. For any given invocation $\text{mrgIf}(s, c, u_t, u_f)$, exactly one rule from Figure 5 or Figure 9 can apply based on s, c, u_t, and u_f. □

Lemma 4.4.3 (Correctness: Preservation of Hierarchical Merging Invariant). *If* $\text{mrgIf}(s, c, u_t, u_f)$ *is a well-formed invocation and* $\text{mrgIf}(s, c, u_t, u_f) \Downarrow u'$*, then* u' *satisfies the Hierarchical Merging Invariant (Definition 4.3.5) with respect to strategy* s *and the set of all possible payload values of type* α*.*

Proof Sketch. This is proven by induction on the structure of the evaluation derivation $\Downarrow$. Each rule is constructed such that if its inputs (operands u_t, u_f and the results of any recursive mrgIf calls) satisfy the invariant, the resulting ORG union u' also satisfies it. For example, in Rule MrgIfSortedEq, u_{head} satisfies the invariant with respect to $\text{sub}(s)(i_t)$ by the inductive hypothesis on the first recursive call. Similarly, u_{tail} satisfies the invariant with respect to s by the inductive hypothesis on the second recursive call. The buildIf rule then combines u_{head} and u_{tail}. Since u_{head} corresponds to index i_t, and all elements in u_{tail} will have indices strictly greater than i_t, the resulting structure maintains the sorted property at the current level of the hierarchy. Other rules similarly establish the invariant. □

These lemmas imply that mrgIfWithStrategy can be implemented as a terminating function that deterministically produces an ORG union that satisfies the Hierarchical Merging Invariant, provided its inputs satisfy the invariant.

For the computational complexity:

Lemma 4.4.4 (Bilinear Time Complexity)**.** *The number of reduction steps for a well-formed* mrgIf *invocation* $\text{mrgIf}(s, c, u_t, u_f)$ *is bounded by* $O(\text{depth}(s) \cdot (\text{usize}(u_t) + \text{usize}(u_f)))$.

Proof Sketch. The merging process can be viewed as a series of linked list merge operations, one for each level of the strategy depth. At a given level of a sorted strategy, the rules in Figure 9 effectively traverse the right spines of the input u_t and u_f once. Each step consumes the head of at least one union (h_t or h_f) or terminates. The number of such head blocks along the spines is bounded by $\text{usize}(t) + \text{usize}(f)$. When Rule MrgIfSortedEq makes a recursive call with a sub-strategy $\text{sub}(s)(i_t)$, the strategy depth decreases. The total work is thus bounded by $O(\text{depth}(s) \cdot (\text{usize}(t) + \text{usize}(f)))$. Note that this result assumes that the helpers introduced in Section 4.4.2 can be computed in constant time. □

Since the depth of merging strategies is typically a small constant for most Haskell types, the algorithm's runtime is often effectively linear in the sum of the sizes of the inputs ORG unions. However, due to the hierarchical nature and sparse merge optimization (where entire sub-trees can be incorporated without recursion into them), the actual number of comparisons and recursive calls can sometimes be significantly less than this worst-case bound, offering sub-linear performance in practice.

4.5 PRACTICAL IMPLEMENTATION FOR ORG REPRESENTATION

After discussing the ORG representation and the merging algorithm, we now discuss the practical issues arising in the implementation of the ORG representation in the Haskell programming language.

4.5.1 *Caching Information in UnionBase*

To achieve the desired complexity and ensure the robustness of the merging algorithm in a practical library implementation, the `UnionBase` structure in GRISETTE incorporates additional information.

First, the $O(1)$ assumptions for helper functions like leftMostIndex(s, u) (which relies on leftMost(u)) is made practical by caching. The leftmost value of a sub-tree can be computed once, in constant time, at its construction and stored within each `If` node. When a new `If` node `If`(c, u_t, u_f) is constructed, its cached leftmost value is simply the cached leftmost value of u_t. This makes retrieving the representative leftmost payload an $O(1)$ operation.

Second, while the core merging rules assume that input union u_t and u_f already satisfy the Hierarchical Merging Invariant with respect to s, this might not always hold if users construct unions improperly (e.g., forget to use `mrgReturn` instead of `return`). To handle this, GRISETTE's `UnionBase` data type includes a Boolean flag indicating whether a give union node is known to be normalized (i.e., satisfy the Hierarchical Merging Invariant with respect to its root strategy). The system would reconstruct the union bottom-up if the flag is not set. The effect of this is often localized and won't have a huge impact on the performance of the system.

```
data UnionBase a
  = Single a
  | If {
      _cachedLeftMost :: a
      _isNormlized    :: Bool
      _pathCond       :: SymBool
      _thenBranch     :: UnionBase a
      _elseBranch     :: UnionBase a
    }

leftMost (Single x) = x
leftMost (If lm _ _ _ _) = lm

isNormalized (Single _) = True
isNormalized (If _ n _ _ _) = n
```

```
16
17 -- Public API ensures normalization and works with unnormalized unions
18 mrgIf c t f | not (isNormalized t) = mrgIf c (normalize t) f
19 mrgIf c t f | not (isNormalized f) = mrgIf c t (normalize f)
20 -- mrgIfWithStrategy-constructed unions are always normalized
21 mrgIf c t f = mrgIfWithStrategy rootStrategy t f
22
23 -- Internal function for normalizing unions
24 normalize u@(If _ _ c t f) | isNormalized u = u
25 normalize u@(If _ _ c t f) =
26   mrgIf c (normalize t) (normalize f)
```

The public operations like the `mrgIf` (which calls `mrgIfWithStrategy` with the `rootStrategy`) first ensure their arguments are normalized by calling a `normalize` function. This `normalize` function checks the `isNormalized` flag. If a union is not marked as normalized, it recursively normalizes its children and then rebuilds the current node using `mrgIf`. The result of `mrgIf` or `mrgIfWithStrategy` is always marked as normalized. This lazy normalization approach ensures that the core semantic rules operate on inputs satisfying the invariant, making the system robust with deferred normalization.

These practical considerations ensure that the ORG merging algorithm is both efficient and correctly maintains the crucial Hierarchical Merging Invariant throughout symbolic evaluation in Grisette. We will not show these additional fields in later discussions.

4.5.2 *Monadic Operations for Union with Merging*

For a practical use of the ORG tree structure `UnionBase`, it needs to support monadic operations, particularly for sequencing computations with `do`-notation.

A basic `Monad` instance for the raw `UnionBase` type can be defined by structurally applying the bound function to the leaves in the `(>>=)` combinator.

```
1 instance Monad UnionBase where
2   return = Single
3   Single a >>= fun = fun a
4   If c t f >>= fun = If c (t >>= fun) (f >>= fun)
```

However, the standard `Monad` class provides no mechanism to access a `Mergeable` instance during the binding as it cannot make any assumption on the value stored in the resulting monad. Thus, this `(>>=)` combinator uses the basic `If` constructor, not the invariant-preserving `mrgIfWithStrategy`,

and the result will not be merged. Consequently, repeated use of this `(>>=)` can produce `UnionBase` that are not merged and lead to path explosion, unless users manually normalize results.

To provide a seamless monadic experience where ORG invariants are largely maintained automatically, GRISETTE offers a user-facing `Union` type. This is a wrapper around `UnionBase` that *may* cache the root merging strategy for its contents:

```
data Union a = Union (Maybe (MergingStrategy a)) (UnionBase a)
```

The `Monad` instance for this user-facing `Union` can then leverage these cached strategies. When its `(>>=)` operation combines results from different paths, it uses a helper function `mrgIfPropagatedStrategy`. This helper attempts to use a cached strategy from either operand; if neither has one, it constructs a basic (unmerged) `If` node within a `Union Nothing` wrapper.

```
data Union a = Union (Maybe (MergingStrategy a)) (UnionBase a)

mrgIfPropagatedStrategy cond (Union Nothing t) (Union Nothing f) =
  Union Nothing $ If (_cachedLeftMost t) False cond t f
mrgIfPropagatedStrategy cond t@(Union (Just m) _) f =
  mrgIfWithStrategy m t f
mrgIfPropagatedStrategy cond t f@(Union (Just m) _) =
  mrgIfWithStrategy m t f

bindUnionBase :: UnionBase a → (a → Union b) → Union b
bindUnionBase (Single a) fun = fun a
bindUnionBase (If _ _ cond t f) fun =
  mrgIfPropagatedStrategy c (bindUnionBase t fun) (bindUnionBase f fun)

instance Monad Union where
  return a = Union Nothing (return a)
  (UnionBase _ u) >>= fun = bindUnionBase u fun
```

The key to effective invariant maintenance is ensuring that functions within `do`-blocks produce `Union` values with cached merging strategy. GRISETTE provides a suite of `mrg*` combinators for this purpose. For example, `mrgReturn` uses the `Mergeable a` constraint to fetch the root strategy for `a` and wraps it in the resulting `Union a`:

```
mrgReturn :: (Mergeable a) ⇒ a → Union a
mrgReturn a = Union (Just rootStrategy) (return a)
```

The `(>>=)` operator can then use `mrgIfPropagatedStrategy` to merge the results with the cached strategy to ensure that the result is merged. Thus, by consistently using the `mrg*` family of combinators, monadic `do`-notation in

GRISETTE largely maintains ORG invariants automatically. This technique is known as knowledge propagation [37].

4.5.3 *Supporting Monad Transformers*

The merging principles also extend to `Unions` used with monad transformers like `ExceptT e` or `StateT s`. To generalize merging operations like `mrgIf` across these transformed stacks, GRISETTE provides some type classes, primarily the following ones:

```
-- Ensures a monadic value is normalized to its root strategy and
-- caches the strategy
class TryMerge m where
  tryMergeWithStrategy :: MergingStrategy a → m a → m a

-- SimpleMergeable1 m is akin to the standard Eq1/Ord1 classes.
-- It means that for all a, if a is SimpleMergeable,
-- then m a is SimpleMergeable.
--
-- forall a. (Mergeable a) ⇒ SimpleMergeable (m a) means that for all
-- a, if a is Mergeable, then m a is SimpleMergeable.
class
  ( SimpleMergeable1 m,
    forall a. (Mergeable a) ⇒ SimpleMergeable (m a),
    TryMerge m
  ) ⇒
  SymBranching m where
  mrgIfWithStrategy ::
    MergingStrategy a → SymBool → m a → m a → m a
  mrgIfPropagatedStrategy :: SymBool → m a → m a → m a

tryMerge :: (TryMerge m, Mergeable a) ⇒ m a → m a
tryMerge = tryMergeWithStrategy rootStrategy
mrgReturn :: (TryMerge m, Monad m) ⇒ a → m a
mrgReturn = tryMerge . return
mrgIf :: (SymBranching m, Mergeable a) ⇒ SymBool → m a → m a → m a
mrgIf = mrgIfWithStrategy rootStrategy
```

The `TryMerge` generalizes the normalization step. The `mrgReturn` uses it to ensure that the results are normalized if possible. The type class is called `TryMerge` because it can also be implemented for other monads that is only used in concrete execution to allow generic functions written with `mrg*` versions can be applied to them.

To support a new monad transformer, one typically need to implement `TryMerge` and `SymBranching` (also `SimpleMergeable1`, etc., in a very similar way). For `ExceptT e Union a`, which wraps `Union (Either e a)`:

```haskell
-- The following comes from transformers package
data ExceptT e m a = ExceptT { runExceptT :: m (Either e a) }

instance (Mergeable e, TryMerge m) ⇒ TryMerge (ExceptT e m) where
  tryMergeWithStrategy strategy (ExceptT ma) =
    -- liftRootStrategy is provided by Mergeable1 (Either e)
    -- Here, liftRootStrategy strategy transforms a strategy with type
    -- MergingStrategy a to MergingStrategy (Either e a)
    ExceptT $ tryMergeWithStrategy (liftRootStrategy strategy) ma

instance
  (SymBranching m, Mergeable e) ⇒ SymBranching (ExceptT e m) where
  mrgIfWithStrategy s c (ExceptT t) (ExceptT f) =
    ExceptT $ mrgIfWithStrategy (liftRootStrategy s) cond t f
  mrgIfPropagatedStrategy s c (ExceptT t) (ExceptT f) =
    ExceptT $ mrgIfPropagatedStrategy (liftRootStrategy s) cond t f
```

Here, the key is that operations on transformed monad stacks (for example, `ExceptT e Union a`) can be implemented by lifting the strategy for the result type `a` to use the underlying `Union`'s merging capabilities for `Union (Either e a)`. The internal structure can then be merged efficiently, and branching on symbolic conditions using `mrgIf` is now supported.

Grisette provides these instances for standard `mtl` transformers [25], enabling complex, effectful symbolic computations with ORG-based merging. This extensibility, for instance, allows easier integration of effects like continuations (`ContT`), which have posed challenges in other symbolic systems [68].

4.5.4 *Error Provenance without Overhead*

As discussed in Section 3.3.1, Grisette manages operational failures by embedding them as values within `Unions`, typically using the `ExceptT` transformer. This allows users to define rich, custom error types to capture detailed diagnostic information or provenance, such as error reasons or source locations. A significant advantage of Grisette's Ordered Guards (ORG) representation for its `Union` type is its ability to handle such detailed error information efficiently. Often, this rich provenance can be maintained without imposing unnecessary overhead on SMT solver queries, particularly when the full detail is not required for a specific analysis.

The `Mergeable` instance for sum types like `Either err res` naturally structures the `Union` for this. When merging `Union (Either err res)` values, outcomes of `Left err` are grouped separately from `Right val` outcomes by the ORG merging algorithm. This typically results in a top-level ORG structure of the form: $\text{If}(c_{\texttt{err}}, t_{\texttt{Left}}, t_{\texttt{Right}})$. Here $c_{\texttt{err}}$ is the condition that is true if *any* error occurs. $t_{\texttt{Left}}$ is itself an ORG tree containing only the error outcomes from the `Left` cases, internally merged according to the `Mergeable` instance of `err` type. Similarly, $t_{\texttt{Right}}$ is an ORG tree of successful outcomes.

This structure facilitates efficient querying via the `solveExcept` interface (first shown in Section 2.4.2), which takes a function to determine which paths are of interest to the SMT solver:

- *Querying for Any Error:* If the user wants to determine if the *any* error can occur, they can choose to interpret the outcomes of symbolic evaluation with `\case Left _ -> true; Right _ -> false`. When this predicate is applied to the ORG structure, the whole $t_{\texttt{Left}}$ can be mapped to `true`, and the whole $t_{\texttt{Right}}$ can be mapped to false. With basic SMT term simplification, the constraint generated for the solver is reduced to $c_{\texttt{err}}$. The internal distinctions within $t_{\texttt{Left}}$ for different error subtypes or origins do not need to be part of this specific SMT query.
- *Querying for Specific Errors (with Provenance):* Conversely, if the outcome interpretation targets a very specific error, including its detailed provenance, the SMT query will utilize the more detailed structure within $t_{\texttt{Left}}$. The query will then incorporate the path conditions leading to that exact error, including conditions on its provenance data.

Thus, Grisette's ORG representation, by hierarchically merging values based on their structure as defined by `Mergeable` instances, enables the maintenance of detailed error information without necessarily adding complexity to SMT queries when such details are used by specific analysis. This offers a good balance between the ability to perform rich diagnostics and the need to maintain efficient SMT solving.

4.6 PERFORMANCE EVALUATION

After discussing the design and semantics of the Ordered Guards (ORG) representation for symbolic unions in Grisette, this section evaluates its performance. Our evaluation aims to answer the following questions:

RQ1 How does GRISETTE with ORG compare in performance (evaluation time, SMT solving time, formula size) to a state-of-the-art symbolic host language (ROSETTE) that employs a different union representation?

RQ2 Within GRISETTE, how does the ORG representation perform against a Mutually Exclusive Guards (MEG) baseline for union merging?

RQ3 How does GRISETTE's all-path symbolic evaluation using ORG compare to single-path, path-enumeration-based symbolic execution systems for tasks requiring exhaustive path analysis?

RQ4 Can standard functional programming optimizations like memoization effectively enhance GRISETTE's performance when using ORG?

4.6.1 *Comparison with Rosette and a MEG implementation in Grisette*

To evaluate GRISETTE against ROSETTE [49, 71, 72], a state-of-the-art symbolic host language, we ported five benchmarks from ROSETTE 4's benchmark suite, plus an additional BONSAI benchmark.[2]. These benchmarks (statistics in Table 2) cover diverse symbolic reasoning tasks:

- FERRITE [10]: A tool for specifying and checking the crash-consistency models for file systems. It can verify an implementation against a model or synthesize barriers for crash-safety.
- IFCL [72]: A functional symbolic DSL for specifying and verifying information flow control policies in abstract stack-and-pointer machines.
- FLUIDICS [77]: A small DSL for synthesizing control programs for microfluidic array manipulations.
- COSETTE [18]: An automated SQL equivalence checker that symbolically executes SQL queries. The ROSETTE version heavily uses Racket's macros for staging, which could be achieved with Template Haskell in GRISETTE. We found a typographical error in the original paper's benchmark code that impacts performance. We fixed the ROSETTE code, and our GRISETTE port also includes the fix.
- BONSAI [17]: A framework for reasoning about type systems by synthesizing programs revealing soundness issues. It is capable of representing large search spaces compactly with a special symbolic data structure called *Bonsai* tree. We ported two benchmarks for BONSAI:

2 The benchmarks are available at https://github.com/lsrcz/grisette-benchmarks

- DOT: Detects soundness issues in the DOT type system (related to Scala 3 [2]) with null references.
- LetPoly: Verifies a historical bug concerning let-polymorphism with mutable references in ML-like languages [66, 78]. This was ported from the original Bonsai repository as it demonstrates a larger-scale problem.

We also implemented two new Regex benchmarks using coroutines (free monads or delimited continuations), which uses features not easily expressible in Rosette to further evaluate our programming model and ORG representation. Our Grisette ports are based on an earlier version of Grisette (late 2022 version); new utilities and helpers in Grisette are expected to reduce LoC significantly while keeping a similar performance.

Table 2: Statistics of the benchmarks used for Grisette performance evaluation

Name	Grisette LoC	Rosette LoC	LoC diff (G-R)	Solver
Bonsai-DOT [17]	663	439	+224 (+51%)	Boolector
Bonsai-LetPoly [17]	750	427	+323 (+76%)	Boolector
Cosette [18]	661	532	+129 (+24%)	Z3
Ferrite [10]	538	348	+190 (+55%)	Z3
Fluidics [77]	141	98	+43 (+44%)	Z3
IFCL [72]	653	483	+170 (+35%)	Boolector
RegexCont (New)	239	N/A	N/A	Z3
RegexFree (New)	219	N/A	N/A	Z3

To isolate ORG's impact versus the traditional list-based Mutually Exclusive Guards (MEG) unions, while mitigating impacts of language differences (Haskell vs. Racket), we implemented a MEG-based `Union` within Grisette. This MEG variant adapts Rosette's merging algorithm to Haskell's static types but shares all other Grisette infrastructure. This allows a direct comparison of core union merging strategies.

Performance metrics (total time, evaluation time, pure symbolic evaluation time (symbolic evaluation time only, excluding SMT lowering/solver interaction), SMT solving time, and SMT term count) were collected on an AMD Ryzen 1950X using GHC 9.0.2, Racket v8.1, Z3 [43] v4.8.8, and Boolector [13] v3.2.1. Results are in Table 3 and performance ratios in Table 4. Pure evaluation is relevant for ORG vs. MEG comparison to mitigate the impact of an external SMT solver backend library bottleneck affecting our SMT lowering.

The results (Table 3) show that Grisette with ORG consistently outperforms Rosette 3 and Rosette 4 in evaluation time and generated formula

Table 3: Performance results. Total (s), Eval (s), Pure (s), Solv (s), and #Term ($\times 10^3$) represent total time, evaluation time, evaluation time excluding SMT lowering and solver communication, solving time, and term count, respectively.

Benchmark	Grisette					Grisette (MEG)				
	Total	Eval	Pure	Solv	#Term	Total	Eval	Pure	Solv	#Term
Ferrite	2.4	.79	.42	1.6	12	8.8	2.3	.53	6.5	56
IFCL	33	4.1	.67	29	91	38	9.4	1.6	29	222
Fluidics	19	3.3	.52	15	84	198	122	4.3	77	1371
Cosette	.27	.094	.045	.18	1.5	1.6	.44	.18	1.1	8.4
DOT	1.8	1.1	.34	.68	21	4.9	4.0	1.2	.92	71
LetPoly	77	13	7.8	64	102	1009	833	108	176	1725
Cosette-1	.031	.01	.004	.021	.077	.03	.011	.007	.019	.077
RegexCont	3.9	3.6	2.9	.3	16	60	53	13	7.1	311
RegexFree	4.8	4.5	3.7	.29	16	63	54	14	8.9	318

Benchmark	Rosette 3				Rosette 4			
	Total	Eval	Solv	#Term	Total	Eval	Solv	#Term
Ferrite	25	17	8.0	34	34	17	17	40
IFCL	194	14	180	383	147	14	133	438
Fluidics	23	8.4	15	284	29	8.8	20	308
Cosette	16	7.9	7.8	114	12	8.1	4.1	128
DOT	27	23	3.8	664	52	47	4.3	1222
LetPoly	1396	186	1210	4908	1284	142	1142	5152
Cosette-1	.16	.14	.025	.13	.15	.13	.025	.13

Table 4: Performance ratios of Grisette (ORG) over baselines.

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

size, and generally also in SMT solving time (except for Fluidics). On average, Grisette (ORG) accelerates symbolic compilation over Rosette 4 by approximately 14.1×, generates 91.2% more compact formulas, and speeds up SMT solving by 5.7×. Against its own MEG baseline, Grisette (ORG) shows a mean pure evaluation time speedup of 4.9× and a term count reduction of 79.2%. The MEG semantics perform poorly on benchmarks with many conditional branches (Fluidics, LetPoly, and Regex*), where MEG's

need to maintain mutually exclusive guards leads to larger formulas and evaluation overhead. This highlights ORG's effectiveness in compactly representing and efficiently merging complex conditional values.

4.6.2 *Comparison with Single-Path Symbolic Execution (G2Q)*

To evaluate GRISETTE's performance in all-path symbolic evaluation, we compared GRISETTE with G2Q [28], a user-friendly interface to the G2 [27] path-enumeration-based symbolic execution engine in Haskell. While G2 is not an all-path evaluator by design (it explores paths by enumeration), it represents a relevant class of tools. We ported G2Q's benchmarks to GRISETTE.

Table 5: G2Q benchmarks

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

Table 5 shows that GRISETTE (which performs all-path evaluation natively) solves these benchmarks significantly faster (often <1 second) than G2Q's reported times. G2Q's enumeration-based nature is most suitable for tasks like test generation. However, for tasks requiring verification of all paths or synthesis over a search space, GRISETTE's all-path approach with ORG is more efficient.

4.6.3 *Effectiveness of Memoization with Grisette*

Grisette's purely functional nature can facilitate standard optimizations like memoization. To evaluate this, we applied memoization to the symbolic evaluation phase (excluding SMT lowering/solving) of Bonsai benchmarks (DOT and LetPoly) and the Regex synthesizers, which feature significant shared sub-computations.

Table 6: Benchmarks for memoization

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

Table 6 shows that memoization further accelerates pure symbolic evaluation by 1.6× to 7.9×, leading to overall speedups of 1.2× to 7.5×. This shows that Grisette's design allows leveraging common functional programming optimization techniques.

4.7 CHAPTER SUMMARY

This chapter detailed Grisette's Ordered Guards (ORG) as a novel representation for symbolic unions (`Union`). It is designed to manage the structurally complex values coming from Grisette's purely functional programming model. We motivated ORG by contrasting it with the limitations of traditional union representations in this context.

The ORG representation itself was defined as an if-then-else tree structure (`UnionBase`). Its fully merged, canonical form that satisfies the *Hierarchical Merging Invariant* is maintained through *type-directed merging strategies* specified via the `Mergeable` type class. This allows users to define how diverse Haskell types can be merged structurally, under symbolic conditions. This chapter then presented the formal semantics of the core ORG merging algorithm and established its key properties, including termination, determinism, correctness, and efficient complexity.

Finally, we discussed practical aspects, such as how the user-facing `Union` monad robustly maintains the invariant through knowledge propagation with `mrg*` combinators, and how this framework easily extends to work with monad transformers. We also demonstrated how ORG supports rich

error provenance often without SMT solving overhead when full provenance is not used in a query. The ORG representation thus serves as a robust and efficient foundation for GRISETTE's design and is crucial for building scalable automated reasoning tools.

5

EXPERIENCE OF GRISETTE IN TENSORRIGHT

The preceding chapters have detailed the design, capabilities, and underlying principles of Grisette, a library for symbolic compilation aimed at simplifying the construction of specialized automated reasoning tools. We explored how Grisette's features, such as its robust handling of multi-path execution via symbolic unions (Section 2.2), its transparent and predictable monadic interface (Section 3.3), and its extensible merging strategies with the ORG representation (Section 3.4.3, Section 4.3), provide a powerful and flexible foundation for developers.

This chapter presents our experience applying Grisette in the development of TensorRight [4], an automated system for *unbounded verification* of tensor graph rewrites. Such rewrites are fundamental for optimizing tensor computations in modern deep learning frameworks (e.g., TensorFlow [1], PyTorch [3], JAX [12]) and their tensor compilers (e.g., XLA [52], TorchInductor [3]), and verifying the semantic equivalence of these rewrites is crucial for compiler reliability. These rewrites often need to be rank- and size-polymorphic: in a compiler, a single rewrite rule is intended to be instantiated to operate correctly on tensors of various ranks and shapes. The rank-polymorphic aspect is particularly challenging as it involves a variable number of axes, which standard SMT solver theories do not directly support. TensorRight solves this verification problem by reducing the unbounded problem to a set of bounded sub-problems, and its core bounded verifier is implemented using Grisette.

The key observation in TensorRight is that while tensors in a rewrite rule may be instantiated to an arbitrary number of axes, these axes can typically be grouped into a fixed number of sets that are treated uniformly. To capture this, TensorRight's DSL allows developers to express rewrite rules using a bounded number of *aggregated axes*. This insight transforms the verification problem: instead of reasoning about an unbounded number of individual axes, we reason about a bounded number of aggregated axes, effectively hiding the unboundedness within these structures. For example, in a rule involving a convolution operation, the rule might treat all batch axes in the same way but differently from how it treats feature axes. In this scenario, the batch axes and feature axes would form two distinct aggregated axes.

A second key observation is that the ranks of different aggregated axes often constrain each other. For instance, a binary element-wise operation requires that corresponding aggregated axes in its two tensor operands instantiate to the same number of concrete axes for the operation to be well-formed. The TensorRight DSL expresses these correlations by grouping aggregated axes into equivalence classes called *rank classes* (RClasses). This concept is critical for scalability: it reduces the dimensionality of the verification space from the number of aggregated axes to the much smaller number of RClasses and prunes the number of instantiations that must be checked.

Leveraging the structure imposed by RClasses, TensorRight employs an induction principle, proven in the paper [4], to infer a small, finite bound on the rank of each RClass. This principle guarantees that it is sufficient to prove correctness for a finite set of *bounded verification* problems—those generated by instantiating RClasses up to their inferred bounds—to generalize the result to the unbounded case. The overall TensorRight system, depicted in Figure 10, automates this entire process.

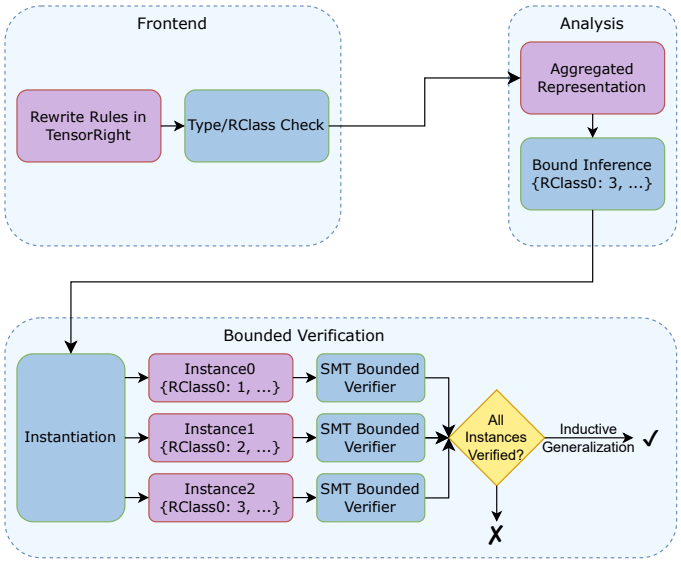


Figure 10: Overview of the TensorRight verification pipeline. Rewrite rules undergo type checking, rank class compatibility checking, and bounded verification. Rank classes (RClasses) group aggregated axes, and the system infers bounds on their ranks. It then instantiates these aggregated axes with in rank classes into specific verification problems. Here, RClasso is inferred to have a bound of 3. The induction principle requires verifying a set of base cases (here, instantiating to ran 1, 2, and 3). Successfully verifying these bounded instances generalizes correctness via induction to unbounded cases.

Our focus in this chapter is the core *bounded verifier* of TensorRight, which was initially prototyped in Rosette [49, 71, 72]. The bounded verifier symbolically executes the left-hand and right-hand sides of an instantiated rule (where RClasses are instantiated to a fixed number of concrete axes) and generates SMT queries to prove their equivalence. I contributed to designing the DSL for representing these rank- and size-polymorphic tensor operations and its instantiation logic, and migrating the bounded verifier prototype to Grisette.

This migration highlighted several practical benefits stemming from Grisette's statically-typed, functional, and monadic design. These advantages directly addressed key challenges encountered during the development of the verifier, particularly in error handling, type safety, and extensibility. This chapter serves as an experience report comparing the Rosette-based and Grisette-based implementation approaches, detailing how Grisette's design facilitated a more robust, maintainable, and extensible verifier. The following sections will elaborate on these benefits:

- *Flexible Error Handling*: Section 5.1 will show how Grisette's monadic framework provided a principled solution for managing the context-specific verification conditions required by TensorRight.
- *Early Bug Detection*: Section 5.2 will detail how Grisette's static type system and explicit error handling helped identify and prevent subtle bugs that were present in the dynamically-typed Rosette implementation.
- *Enhanced Modularity*: Section 5.3 will explain how Haskell's type class mechanism enabled a more modular and extensible design for generic tensor operations.

5.1 FLEXIBLE ERROR HANDLING WITH MONADS

The primary goal of TensorRight's bounded verifier is to prove the semantic equivalence of a left-hand side (LHS) and a right-hand side (RHS) tensor expression derived from an instantiated rewrite rule $L \Rightarrow_c R$ (where c is a rule-specific precondition). This is achieved by symbolically executing both L and R, and then establishing that, *assuming* the precondition c holds and an access to L at an arbitrary index I is valid (i.e., L is structurally sound and I is within bounds), then it must be possible to *assert* that an access to R at I is also valid, and that $L[I]$ equals $R[I]$. This context-specific interpretation of validity conditions—as assumptions for the LHS and assertions for the RHS when looking for counterexamples—is central to the verification task,

especially for rules involving reductions which often require bidirectional implication checks (requiring the same condition to be interpreted as an assertion or assumption in different queries; see the TensorRight [4] paper for details).

In the ROSETTE-based prototype, this need for flexible error interpretation was addressed by manually managing explicit symbolic Boolean conditions for structural and access validity. The constraints used for verification were then constructed by combining these Boolean conditions. This manual bookkeeping was necessary because ROSETTE's built-in `assert`/`assume` constructs hard-code a condition's interpretation within a global verification condition, making them unsuitable or harder to use for this context-specific interpretation. Consequently, tensor operations had to manually propagate these flags:

```
(define (element-wise-op op X Y)
  ; Manually combine structural validity from X and Y
  ; and check if their shapes are compatible
  (let* ([valid-tensor
          (and (tensor-valid X) (tensor-valid Y) (shape-equal? X Y))])
    ; A tensor object in the Rosette version contains
    ; 1. a structural validity condition
    ; 2. an access function that when applied to an index, returns a
    ;    pair containing:
    ;    a. an access validity condition
    ;    b. the value at the index
    (tensor
      valid-tensor
      (lambda (indices)
          (let-values ([(acc-valid-X value-X) (access X indices)]
                       [(acc-valid-Y value-Y) (access Y indices)])
              (values
                ; Manually combine access validity
                (and acc-valid-X acc-valid-Y)
                ; Compute the actual value
                ...))))
        (tensor-dims X)))
```

This manual management of Boolean conditions proved to be tedious and highly error-prone, often leading to bugs like accidentally omitting an error condition. It also resulted in a loss of diagnostic information, as distinct failure reasons were all collapsed into a single Boolean constraint.

GRISETTE offered a more principled and streamlined solution through Haskell's monadic error handling, using `ExceptT`. Tensor operations and access functions are defined as monadic computations that explicitly yield a symbolic union where each path can either succeed with a value or fail

with an error. If an operation detects a failure, it uses an `assert` function (implemented with `throwError` and `mrgIf`) to signal the error conditionally, retaining detailed error information. The `ExceptT` monad then automatically propagates this typed error, short-circuiting subsequent operations on that path.

This monadic approach eliminates manual error condition bookkeeping and preserves diagnostic information. Crucially, the symbolic evaluation result within `ExceptT` is itself a regular symbolic value. The SMT query generation phase can then inspect this value and flexibly map error conditions to assumptions or assertions as needed. The element-wise operator in the Grisette-based approach becomes:

```
elementWiseOp op x y = do
  -- Errors from monadic inputs x and y are automatically propagated
  xTensor ← x
  yTensor ← y
  -- Check shape compatibility; 'assert' uses mrgThrowError internally.
  -- The error message is retained if this assertion fails.
  assert "Shape mismatch." $
    sameAxisMap (tensorShape xTensor) (tensorShape yTensor)
  -- If all checks pass, construct the new tensor
  mrgReturn $ Tensor
    { tensorShape = tensorShape xTensor,
      tensorAccess = \indices → do
        -- Access errors from xElem and yElem are also propagated
        xElem ← tensorAccess xTensor indices
        yElem ← tensorAccess yTensor indices
        ... -- further computation
    }
```

This monadic design provides automatic error propagation, rich diagnostics, and the necessary separation between symbolic execution (yielding a structured result value in `ExceptT`) and SMT query generation (interpreting these results into symbolic constraints). This directly addresses the limitations of the manual bookkeeping required in the Rosette prototype.

5.2 EARLY BUG DETECTION VIA STATIC TYPING AND EXPLICIT ERROR HANDLING

A significant challenge in developing symbolic systems arises when handling errors or choices that might depend on symbolic conditions. As discussed in Section 3.3.1, Rosette, as a dynamically-typed language, offers flexibility but can obscure certain classes of programming errors until runtime, or even lead to subtly incorrect symbolic semantics. Our experience

with the TensorRight prototype, even when developed by an expert familiar with Rosette, revealed such subtle bugs.

For instance, TensorRight has special treatment for tensor reductions (e.g., sum, product over axes). It maintains these as special symbolic constructs and normalizes them according to algebraic rules. This normalization process requires dispatching logic based on whether operands are simple scalar values or themselves reduction constructs (represented as `SumElement` in the Rosette implementation). The complexity arises when an operand's type (scalar vs. reduction construct) depends on a symbolic condition. A simplified Rosette snippet for the `element-wise-op` function performing such a dispatch, written by an expert, used Racket's `match` utility:

```
(match (list op value-X value-Y)
  ; Sum_i(x) * Sum_j(y)                → Sum_{i,j}(x * y)
  [(list * (SumElement x) (SumElement y)) (SumElement (* x y))]
  ; Sum_i(x) * y                       → Sum_i(x * y)
  [(list * (SumElement x)            y)   (SumElement (* x y))]
  ; x * Sum_j(y)                       → Sum_j(x * y)
  [(list *             x (SumElement y))  (SumElement (* x y))]
  ; x * y                              → x * y
  [(list *             x             y)   (* x y)]
  ; Fallback for other ops or unhandled cases
  [(list op x y) (error "not implemented")])
```

This approach, however, presented two key issues in the Rosette context:

- *Undefined behavior with symbolic unions:* The `match` construct is a standard Racket library utility, not a core symbolic construct lifted by Rosette's symbolic engine to work with symbolic unions. When the values being matched were symbolic unions, `match` could interact unpredictably. We observed its behavior was to silently prune certain symbolic branches and incorrectly translate the case analysis into the global SMT verification condition, effectively asserting that only one structural pattern was feasible. This could lead to incorrect verification results (e.g., false positives or negatives) that were very difficult to trace back to the `match` utility's behavior.

- *Masking Programming Errors as Operational Failures:* Assuming we have a correct `match` implementation that works as intended, if an unimplemented case was encountered (e.g., a new `op` or an unexpected combination of value types not covered by the explicit patterns), the resulting Racket `error` call would be interpreted by Rosette's symbolic engine as if that specific execution path had encountered an

`(assert false)`. This effectively masked the programming error (an unhandled case in the verifier's logic) as an operational failure (an infeasible path), translating it into SMT constraints. Again, this could lead the verifier to report incorrect results without a clear diagnostic pointing to the unimplemented case.

These problems, being rarely triggered and sometimes leading to seemingly correct outcomes on limited test cases (e.g., reporting "verified" on some buggy rules due to overly constrained paths), were not noticed by the developer of the Rosette version until the migration to Grisette began.

Grisette helps mitigate these problems through its static type system and its explicit, type-safe approach to handling symbolic choices:

- *Type-Safe Handling of Symbolic Choices:* A Grisette `Union` value, which represents a symbolic choice over different potential concrete values or structures, cannot be directly pattern-matched using Haskell's `case` expression; such an attempt would result in a compile-time type error. Instead, developers are *required* to use monadic operations (e.g., `do` notation with `<-`) to explicitly operate on the value contained within each branch of the `Union`, under its respective path condition. This requirement ensures that the developer clearly states their intent for handling all symbolic possibilities with partial evaluation, and prevents the undefined or misleading behavior associated with applying a general-purpose host language construct (like Racket's `match`) to a symbolic union it was not designed for.
- *Clear Distinction Between Programming and Operational Errors:* Grisette maintains a clear distinction between unrecoverable programming errors (signaled via Haskell's `error`, causing immediate termination with diagnostics) and managed operational failures (represented, for example, within an `ExceptT` and raised via `mrgThrowError`). An incomplete operator definition in the verifier, being a programming error, should be signaled via Haskell's `error`. This ensures it is not silently converted into SMT constraints that could mask the bug or lead to incorrect verification results.

This combination of compile-time type enforcement for symbolic choices and distinct runtime handling for different classes of errors leads to early and clear bug detection within the verifier's implementation, contributing to a more robust tool.

5.3 ENHANCED MODULARITY AND EXTENSIBILITY WITH GENERIC PROGRAMMING

A core requirement for TensorRight is verifying tensor rewrite rules that are *generic* across various element types. Some rules may apply broadly to Booleans, integers, or reals, while others, specific to numeric types, still need to cover both integers and reals. Furthermore, advanced verification scenarios in TensorRight require reasoning about custom numeric semantics, such as reals augmented with explicit infinities, which standard SMT theories do not directly support and thus necessitate specialized handling.

The Rosette-based prototype addressed this need for genericity and custom semantics primarily through ad-hoc mechanisms. Core tensor operations often assumed a default symbolic element type (e.g., SMT-solver reals). Extending support to other element types or incorporating custom semantics (like infinities) typically involved embedding explicit conditional logic and special case handling directly within each relevant tensor operator. For example, to correctly implement algebraic rules like $\max(x, -\infty) = x$, the symbolic max operator in the Rosette-based prototype would require internal logic to recognize special constants representing infinities and apply custom evaluation rules. Assuming `match` handles symbolic unions correctly for this illustration:

```
(match (list op value-X value-Y)
  ... ; Other operator cases
  ; -inf.0 and +inf.0 are special symbols in Racket for infinities.
  [(list max x -inf.0)  x]
  [(list min x +inf.0)  x]
  [(list op  x y)       (error "not implemented")])
```

This approach, while it works for a fixed set of cases, scales poorly. Each new element type or special semantic consideration would necessitate modifications across numerous tensor operators, making the system difficult to maintain and extend robustly.

Grisette, by contrast, leverages Haskell's static type system and its powerful type class mechanism to provide a significantly more modular, extensible, and type-safe solution. Instead of embedding ad-hoc, value-based dispatch within operators, Grisette encourages defining custom symbolic data types for specialized semantics (like reals with infinities) and then implementing standard Haskell type classes (e.g., `Num` for arithmetic) and Grisette-specific ones (e.g., `SymEq` for symbolic equality, `Mergeable` for merging strategies in symbolic unions) for these types. This approach integrates new types and their semantics cleanly into the symbolic evaluation framework.

For instance, to handle reals with explicit infinities in TensorRight using Grisette, we define a custom Haskell data type: This involves a base type, `TensorRealBase`, to represent non-infinite reals, positive/negative infinities, or an unknown/unsupported state, and a wrapper type, `TensorReal`, that uses Grisette's `Union` to handle potential symbolic choices among these base states:

```haskell
data TensorRealBase
  = NonInf SymAlgReal -- Standard real number
  | Inf SymBool -- Infinite (SymBool for +/- sign)
  | Unknown -- Unknown/unsupported state
data TensorReal = TensorReal (Union TensorRealBase)

-- Example: Symbolic equality for TensorReal
instance SymEq TensorRealBase where
  NonInf l .== NonInf r = l .== r
  Inf s1   .== Inf s2 = s1 .== s2 -- Equal if signs are equal
  _        .== _ -- Other comparisons are false
instance SymEq TensorReal where
  TensorReal l .== TensorReal r = l .== r

-- Example: '+' operation for TensorReal
instance Num TensorReal where
  TensorReal lu + TensorReal ru = TensorReal $ do
    lval ← lu
    rval ← ru
    -- We can only add infinities with the same sign, or add
    -- infinities with non-infinite values, or the result is unknown.
    case (lval, rval) of
      (NonInf l, NonInf r) → mrgReturn $ NonInf (l + r)
      (Inf ls, Inf rs) →
        -- -oo + +oo = Unknown
        mrgIf (ls .== rs) (mrgReturn $ Inf ls) (mrgReturn Unknown)
      (Inf ls, NonInf _) → mrgReturn $ Inf ls
      (NonInf _, Inf rs) → mrgReturn $ Inf rs
      _ → mrgReturn Unknown
```

In this Grisette-based approach, all arithmetic and equality logic, including the rules for handling infinities, is encapsulated within the type class instances for `TensorReal` and is defined only once for the type.

With such custom symbolic types fully integrated via type classes, TensorRight can specify generic rewrite rules using Haskell's standard type class constraints. A single rule definition can then be instantiated and verified by the bounded verifier for any symbolic type that satisfies these constraints, including user-defined types like `TensorReal`:

```haskell
-- A rule applicable to any element type supporting symbolic equality
-- and merging (constraints shown here are illustrative)
genericRule :: (SymEq a, Mergeable a) ⇒ Rule a
-- A rule applicable to any numeric element type
numericRule :: (SymEq a, Num a, Mergeable a) ⇒ Rule a
```

This promotes significant code reuse and modularity. It enhances type safety, as the Haskell compiler statically verifies that only types providing the necessary operations (e.g., instances of `Num`) are used with numeric rules. Crucially, this makes the TensorRight system more extensible to new element types or custom semantics. These can be added by defining new data types and their corresponding type class instances, without requiring pervasive, ad-hoc modifications to the core logic of every existing tensor operator, as was often the case in the Rosette-based prototype.

5.4 CHAPTER SUMMARY

This chapter presented our experience migrating TensorRight's bounded verifier from a Rosette prototype to Grisette, showcasing Grisette's practical benefits for developing complex automated reasoning tools. It highlighted three key advantages of Grisette's statically-typed, functional, and monadic design.

First, Grisette's static typing and explicit, monadic handling of symbolic choices lead to *early bug detection*. The problematic uses of host language constructs with symbolic unions, or the misinterpretation of programming errors as operational failures, are subtle issues in the dynamically-typed Rosette system, which are often caught at compile-time or signaled with clear, immediate runtime diagnostics in Grisette.

Second, Grisette's monadic error handling (via `ExceptT`) provided a *flexible and safer way to manage validity conditions*. This is crucial for TensorRight's need to dynamically interpret validity conditions as assumptions or assertions for different verification queries. This task required cumbersome and error-prone manual constraint bookkeeping in the Rosette prototype, whereas Grisette enables automatic propagation of error and richer diagnostic information.

Finally, leveraging Haskell's type class mechanism, Grisette significantly improved *modularity and extensibility via generic programming*. This allows for the clean definition of generic tensor operations and rewrite rules applicable across various standard and custom symbolic element types (like reals with infinities), a task that was more ad-hoc and less extensible in the Rosette-based prototype.

Our experience with TensorRight demonstrates that Grisette provides a practical, robust, and maintainable foundation for building sophisticated, domain-specific automated reasoning tools. It effectively addresses common challenges in symbolic system development through its emphasis on type safety and explicit compositional design.

6

HIERASYNTH: A PARALLEL FRAMEWORK FOR COMPLETE SUPER-OPTIMIZATION WITH HIERARCHICAL SPACE DECOMPOSITION

The preceding chapters have established Grisette as a versatile library for symbolic evaluation and detailed its core architecture and features. We have also shown its practical application in constructing components of TensorRight [4], an automated verifier for tensor graph rewrites, in Chapter 5. This chapter presents HieraSynth, another significant system built upon Grisette. HieraSynth leverages Grisette's robust and efficient symbolic reasoning capabilities along with its own novel parallel decomposition approach to address the scalability challenges of super-optimization. Like Grisette, HieraSynth is also designed as a reusable library, enabling easy implementation of super-optimizers for various instruction sets while inheriting Grisette's advantages in safety and modularity.

6.1 INTRODUCTION

Modern optimizing compilers generate performant code but are not designed to achieve theoretical optimality. For performance-critical applications, experts still rely on hand-crafted assembly or hardware-specific intrinsics to achieve consistent high performance across different compilers. These manual optimizations are challenging for processors with complex vector ISAs that feature numerous instructions with different configurations and constraints. Our analysis of expert-written libraries like Highway [32] reveals optimizations achieving 2-4× acceleration for RISC-V targets, highlighting a significant gap between expert-optimized code and what is theoretically possible.

Super-optimizers [34, 39, 41, 48, 53] address this gap by systematically searching for instruction sequences that deliver higher empirical performance. Since achieving true *empirical optimality*—the best possible performance on specific hardware—is often intractable, many super-optimizers instead aim for *cost-model optimality*, using a formal cost model as a tractable, though necessarily inaccurate, proxy for real-world performance. Unlike traditional compilers, which rely on a set of rule-based heuristics, these super-optimizers use program synthesis to exhaustively search a defined program space for a program that satisfies a specification with the lowest

possible cost. This exhaustive search and specification checking is where a symbolic reasoning engine like GRISETTE becomes invaluable. GRISETTE provides the means to represent program spaces, define instruction semantics, and formulate SMT queries to guide the search. However, even with a powerful symbolic engine, this approach still faces two critical scalability challenges:

1. *Output program size (*k*)*: The ability to *completely/exhaustively* search a program space of up to k instructions.
2. *Instruction set size (*n*)*: The ability to consider all or a large subset of instructions from practical ISAs.

The search space grows as $\Omega(n^k)$, and existing synthesis-based super-optimizers reveal a fundamental power-law trade-off: systems that handle a large k must restrict n to a small, expert-chosen subset (e.g., BRAHMA [26]), while those that handle a large n are limited to a very small k (e.g., MINOTAUR [39]). This constraint has severely limited the application of super-optimization to modern vector ISAs, which require both extensive instruction choices and multi-instruction sequences to implement complex operations.

Prior attempts to scale complete super-optimization have focused on decomposing the problem by program length (k), for example, using peephole-style [42] techniques. While this can handle longer programs, it sacrifices the cost-model optimality guarantee, as the best global solution may not be found by optimizing small, local windows. Separately, incomplete methods, including stochastic search (e.g., STOKE [54]), machine learning (e.g., DREAMCODER [24]), or equality saturation (e.g., DIOSPYROS [73]), can handle larger programs and ISAs, but by design do not explore exhaustively and thus cannot guarantee cost-model optimality. This leads to a fundamental question: How can we decompose the vast search space to overcome the k-vs-n constraint without sacrificing the optimality guarantee?

Our key innovation is to decompose the search space by instruction set size (n) using an *adaptive, hierarchical divide-and-conquer* strategy. While prior work like PASSES [33] has also explored decomposition on n, its complete variants relies on a static, pre-defined partitioning of the instruction set. This non-adaptive approach struggles to scale to large ISAs, as it can result in subspaces that are either too large to solve or too many to manage. In contrast, HIERASYNTH's adaptive approach starts with the entire instruction set and recursively partitions the search space until the resulting subspaces are small enough for a base synthesizer like BRAHMA, which excels at synthesizing long programs from small ISAs. This strategy enables aggressive,

solver-driven pruning of unrealizable subspaces, making the complete, lossless exploration of huge instruction sets tractable while preserving the cost-model optimality guarantee.

This chapter presents HIERASYNTH, a novel super-optimizer that achieves a *qualitative leap* in scalability for complete program space exploration. HIERASYNTH breaks the power-law constraint between k and n, enabling the discovery of programs that are provably optimal under a cost model and empirically faster than expert-written code. HIERASYNTH builds on three core capabilities that enable a parallel divide-and-conquer strategy:

1. *Optimality-Preserving Hierarchical Decomposition*: Our program space representation enables lossless hierarchical decomposition on n (rather than k), dividing the search space into manageable subspaces without overlooking cost-model optimal solutions.

2. *Efficient Unrealizability Detection*: Our constraint encoding allows an SMT solver to quickly prove the unrealizability of a subspace, enabling aggressive pruning of the search tree.

3. *Independent Parallel Exploration*: Decomposing based on instruction selection creates independent subproblems, enabling near-linear parallel speedup, which is impossible with traditional decomposition on k.

To implement these capabilities, HIERASYNTH leverages GRISETTE's symbolic execution engine. We designed a program space representation combining component-based synthesis [26, 34] with program sketching [61, 63]. This representation enables a compact encoding of large program spaces and allows for systematic, hierarchical, lossless partitioning. The synthesis task for each subspace is translated into an angelic programming problem, which is then solved via SMT constraints generated using GRISETTE, with an encoding inspired by BRAHMA's approach, which excels in the large-k, small-n scenario. We organize these subspaces into a search tree and use the solver to prune unrealizable branches.

To evaluate our approach, we implement HIERASYNTH as a reusable library on top of GRISETTE and develop a prototype super-optimizer for the complex RISC-V Vector (RVV) [75] architecture. We chose RVV because its flexible design offers rich optimization opportunities while posing significant challenges for both human experts and automated tools. Our results demonstrate that HIERASYNTH substantially overcomes the k-vs-n constraint of existing approaches, finding cost-model optimal programs in spaces defined by large instruction sets ($n \approx 700$) and long programs

($k \approx 8$). It synthesizes significantly larger programs ($k \approx 8$ vs. $k \approx 1 \sim 3$) than state-of-the-art tools designed for large ISAs (e.g., Bansal and Aiken [6] and MINOTAUR [39]), and handles instruction sets more than $10\times$ larger than other complete synthesizers capable of producing large programs (e.g., SOUPER [53] and LENS [48]).

In summary, this work, presented in this chapter, makes the following contributions:

1. A novel approach to component-based synthesis that compactly encodes large program spaces by embedding instruction choices within components.
2. A systematic, optimality-preserving decomposition strategy for the program space based on instruction set size (n), offering a new perspective on scaling exhaustive super-optimization.
3. A generic, reusable library for building exhaustive, complete super-optimizers that can be adapted to different ISAs, which inherits GRISETTE's advantages in safety and modularity.
4. The first super-optimizer targeting the flexible RISC-V Vector ISA, demonstrating HIERASYNTH's scalability, showing significant parallel speedups, and discovering optimizations that are both provably cost-model optimal and empirically superior to expert-written code.

The subsequent sections are organized as follows: Section 6.2 provides an overview of HIERASYNTH, Section 6.3 discusses our program space representation and the parallel divide-and-conquer approach, Section 6.4 describes further refinements to our algorithm, Section 6.5 formulates the extended component-based synthesis algorithm for solving individual program spaces, Section 6.6 discusses adapting HIERASYNTH to build an RVV super-optimizer, and Section 6.7 presents empirical evaluations demonstrating HIERASYNTH's effectiveness in discovering code that is either cost-model optimal or likely so for real-world applications.

6.2 OVERVIEW

In this section, we first examine the fundamental trade-off between output program size (k) and instruction set size (n) that limits existing exhaustive techniques. We then analyze how previous work attempted to overcome this constraint through decomposition on k, highlighting why this approach sacrifices optimality. Finally, we introduce our novel program space

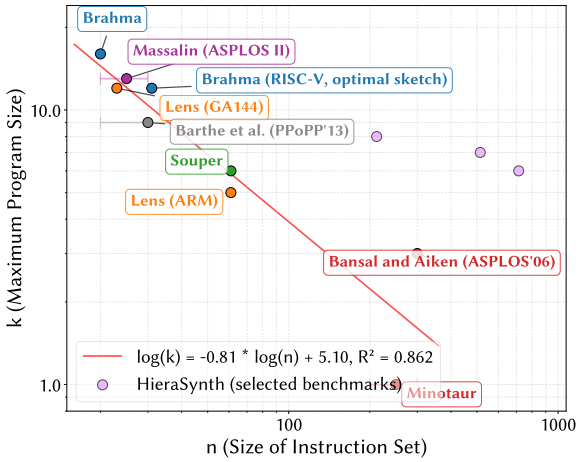


Figure 11: Trade-off between maximum output program size k and instruction set size n in existing complete super-optimization systems.[1]Estimated ranges for n for some systems are marked with bars. As n increases, achievable k decreases substantially, roughly following a power law relationship. By partitioning the program space along the n dimension, HieraSynth breaks through this constraint by enabling large k with very large instruction sets (n).

representation that enables decomposition on n and demonstrate how it allows systematic partitioning and parallel exploration of the search space.

6.2.1 *Background: The k-vs-n Trade-off in Exhaustive Super-Optimization*

Exhaustive super-optimization aims to find a *cost-model optimal* program for a given specification by systematically searching the entire space of possible programs. Unlike non-exhaustive approaches like stochastic search, this method guarantees cost-model optimality: if a correct program exists, the one with the lowest cost according to the model will eventually be found.

For a super-optimization problem, we consider a search space with k instructions, each having n possible instruction choices. The challenge of exhaustive super-optimization is characterized by these two parameters, and the resulting search space grows exponentially at least as $\Omega(n^k)$, making exhaustive exploration intractable as either parameter increases. It is important to note that this does not account for immediate constants or inter-instruction dependencies through variables or registers.

Decades of research show a consistent pattern: as instruction set size (n) increases, the maximum feasible program size (k) decreases substantially. Figure 11 illustrates this trade-off across various systems, includ-

ing Massalin's original super-optimizer [41], Brahma [26, 34], Lens [48], Souper [53], Barthe et al.'s auto-vectorizer [7], Minotaur [39], and Bansal and Aiken's peephole super-optimizer [6]. Our analysis of existing systems reveals a power law relationship: $\log(k) = -0.81 * \log(n) + 5.10$ with $R^2 = 0.862$. This inverse relationship between n and k seems to be a fundamental constraint for the problem. The work by Bansal and Aiken and the Minotaur super-optimizer deviate from this trend, likely due to their distinct approaches to handling immediates. The former only searches over a predefined set of constants, while the latter allows for arbitrary immediates. Synthesizing these arbitrary immediates, a feature also supported by HieraSynth, is a significantly more challenging synthesis problem.

Our system, HieraSynth, substantially outperforms the constraint in Figure 11. While previous systems could only achieve small k with large n (e.g., Minotaur's $k = 1$ with $n \approx 250$), HieraSynth shows $k = 8$ with $n > 200$ or $k = 7$ with $n > 500$ across various benchmarks. This breakthrough is very important in synthesizing complex vector kernels, as demonstrated in Section 6.7.5.

6.2.2 *Previous Approach: Decomposition on k*

To address the scalability challenge, some previous approaches have used peephole-style optimizations [42], which effectively decompose the problem based on output program size k. These methods examine small instruction windows to find cost-model optimal implementations, apply peephole rewrites, and iterate until a fixed point. This process is inherently sequential for a given program snippet and cannot be parallelized.

For example, Bansal and Aiken extracts windows of 6 instructions and replace them with cost-model optimal sequences of up to 3 instructions ($k = 3$ for the output program size). Lens improves on this approach through *context-aware window decomposition*, which leverages surrounding code as preconditions and postconditions to relax correctness constraints and synthesize better code. While effective for many scenarios, these approaches cannot synthesize large programs that must be created all at once. In Section 6.7.5, we demonstrate a vector kernel where finding the cost-model op-

1 Brahma restricts the effective n with expert-selected components, making the program space much smaller than $\Omega(n^k)$. For Souper, which adopts Brahma's approach without expert-selected components, we observe $k = 6$ with $n = 61$. For Lens, we counted the supported instruction set size from their artifact. For other works where instruction set size was not directly reported, we estimated the values from descriptions and examples in their papers. Note that reported k and n values may not be simultaneously achievable in some existing tools.

timal implementation seems to require $k \geqslant 6$. In this kernel, the cost-model optimal implementation requires a different data representation, which our algorithm discovers. Such global representational changes are difficult to achieve via local peephole rewrites.

6.2.3 *Alternative Approach: Decomposition on n*

The limitations of decomposition on k motivate exploring an optimality-preserving decomposition on the instruction set size, n. However, designing a scalable strategy requires solving two fundamental and related challenges.

First, the sub-problems created by any decomposition must be efficiently solvable. This requires a base synthesis technique that excels in a large-k (program length, not decomposed) but small-n (instruction set size, decomposed) scenario. As Figure 11 suggests, the component-based approach of BRAHMA is the most capable in this specific scenario. As shown in Figure 12a, BRAHMA uses a standard library of common instructions, augmented with *expert-curated, problem-specific* operators, and determines both operation order and dataflow connections. This approach works best with the right instruction selection but does not scale to instruction sets with hundreds of instructions. Systems like SOUPER (Figure 12b) adopt BRAHMA's approach without expert instruction selection, but can only synthesize programs with $k = 6$ given $n \approx 61$. We therefore adopt the BRAHMA style as the base engine for our subproblems, with the goal of automating the "expert curation" of components through partitioning.

Second, having chosen this base synthesizer, we must now manage the combinatorial explosion of creating these subproblems. A naive application that creates the simplest possible subproblems for BRAHMA would define each subproblem by a k-element multiset of instructions drawn from the full ISA. This is computationally infeasible, as it would require enumerating $C(n + k - 1, k)$ subproblems, approximately 10^{14} for a realistic benchmark like MIN128 with $k = 8$ and $n > 200$.

To make this tractable, we introduce a more compact program representation that combines ideas from BRAHMA and SKETCH (Figure 12c). As illustrated in Figure 12d, instead of using individual instructions as components, we may partition the instruction set into g subsets, and our components are SKETCH-like constructs that can themselves choose an instruction from a given *subset*. This allows us to partition the problem by creating subspaces defined by different instruction subsets, reducing the number of enumerated subspaces to roughly $C(n/g + k - 1, k)$, where g is the number of instruction subsets.

```
fn f_brahma(a, b) {
  reorder_and_wire {
    // stdlib
    add, sub, and, ...
    // user-extended
    or, ...
  }
}
```

(a) BRAHMA

```
fn f_souper(a, b) {
  reorder_and_wire {
    add, add, add,
    sub, sub, sub,
    or, ...
  }
}
```

(b) SOUPER

```
fn f_sketch(a, b) {
  // ?op chooses an operator
  // from add/sub/popcnt/...
  o1 = ?op(??, ??)
  o2 = ?op(??, ??)
  o3 = ?op(??, ??)
  return o3
}
```

(c) SKETCH

```
fn f_polysynth(a, b) {
  // here k=3 and n=2
  reorder_and_wire {
    ?op from {add/sub}
    ?op from {and/or}
    ?op from {and/or}
  }
}
```

(d) HIERASYNTH

Figure 12: Program space definition by different tools. Each approach represents the program space differently: BRAHMA uses a standard library augmented with expert-selected operators; SOUPER uses all possible instructions with potential duplication; SKETCH uses holes and choices for operators and operands without reordering statements; HIERASYNTH combines BRAHMA's reordering with SKETCH's choices.

However, even with this more powerful and compact representation, any *static* partitioning of the instruction set leads to a dilemma. Our analysis of the MIN128 benchmark reveals this trade-off:

- Using *large instruction subsets* (e.g., group size $g = 40$) creates a manageable number of subproblems, but many are too difficult for the base synthesizer to solve.
- Using *small instruction subsets* (e.g., group size $g = 13$) makes each subproblem easier, but creates an intractably large number of them (7.4×10^5).

This choice between too few, too hard subproblems and too many, too easy subproblems shows the limitations of any static partitioning strategy, including those used in prior work like PASSES [33]. It demonstrates the critical need for an *adaptive* approach.

6.2.4 *Our Approach: Hierarchical Parallel Divide-and-Conquer on n*

We introduce a hierarchical divide-and-conquer approach that adaptively partitions the instruction space. Rather than selecting a fixed number of instructions per group, we start with k components, each containing all instructions, and progressively partition only when necessary, aggressively pruning unrealizable branches of the decomposition tree. Our approach works as follows:

1. Begin with the full instruction set and try to synthesize a program. If successful, we are done. If times out, partition the instruction set into subsets and try each subset individually.
2. For successful subsets, keep and refine the best program found to find better solutions.
3. For subsets where synthesis times out, further partition them and continue recursively.
4. For unrealizable subsets, where the solver proves that no program in the space satisfies the specification, prune the entire branch, eliminating potentially millions of subproblems.

Figure 13 illustrates this approach with $k = 3$ and $n = 4$. We start with a program space allowing three components, each with any of the four instructions. This space is partitioned into four subspaces, each representing different combinations of instruction subsets: x instructions from `{add,sub}` and use $3 - x$ instructions from `{and,or}`. We find that SMT solvers are usually better at proving unrealizability than synthesis, even for large program spaces, allowing us to prune entire branches of the search tree without explicit exploration. For successful subspaces, incrementally refining a solution is often faster than solving from scratch. For timed-out subspaces, we further pick components to split into smaller ones, gradually reducing the synthesis difficulty.

This hierarchical approach effectively addresses the dilemma between harder or more subtasks by starting with larger subsets and progressively partitioning only when necessary. By aggressively pruning unrealizable branches, we dramatically reduce the number of explored subproblems. For example, our approach reduces the number of explored subspaces before synthesizing the known best program for MIN128 from potentially millions to approximately 500, making it practical on a single server.

Hierarchical decomposition on n is a general technique that could enhance other synthesizers, not just our BRAHMA-inspired base synthesizer.

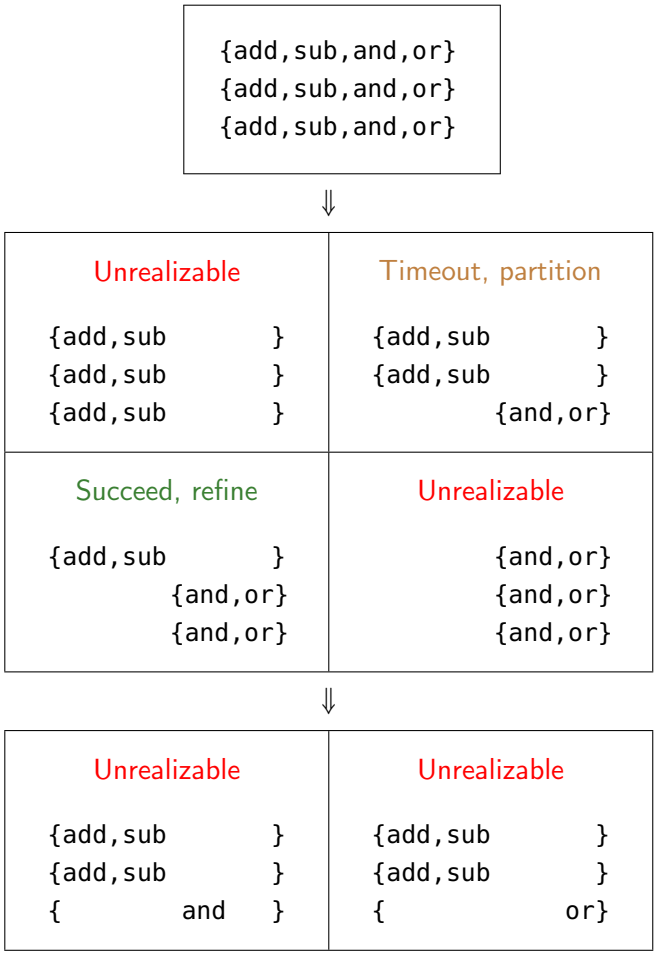


Figure 13: Example of hierarchical divide-and-conquer for a $k = 3$, $n = 4$ program space. For brevity, only the components are shown. 'Unrealizable' means the solver has proven no valid program exists in that subspace, allowing pruning of entire branches. 'Succeed, refine' means a valid program was found and will be further optimized. 'Timeout, partition' means that synthesis exceeds time limits, triggering further decomposition. The example shows two-level decompositions, where in the second level, we partition the `{and,or}` subset in the timed-out subspace into two new subsets.

Additionally, our approach complements rather than replaces decomposition on k, as they are orthogonal strategies that could be combined to further extend super-optimization scalability. While this hierarchical approach addresses the core scalability challenges, it introduces practical considerations for managing the search, such as task scheduling and coordinating

parallel workers. The following sections detail our solutions to these challenges.

6.3 PARALLEL DIVIDE-AND-CONQUER

In this section, we present our approach to breaking the k-vs-n constraint. We begin by introducing our program space representation, which enables hierarchical decomposition on instruction set size (n). We then present our basic parallel divide-and-conquer algorithm and extend it for cost-model optimal program search.

6.3.1 *Program Space Representation*

```
fn space(a, b) {
  reorder_and_wire {
    3 * {{+,-}!2,{*,/}!3}!1
  }
}
fn f(a, b) {
  r1 = a + b
  r2 = r1 * b
  return r2 - a
}
fn subspace_n(a, b) {
  reorder_and_wire {
    n * {+,-}!2 // n is 0~3
    (3-n) * {*,/}!3
  }
}
fn subspace_2_1(a, b) {
  reorder_and_wire {
    +, -, {*,/}!3
  }
}
```

Figure 14: Hierarchical splitting of a $k = 3$, $n = 4$ program space. The top-level space contains 3 components, each choosing from operators `{+,-,*,/}`. `f` is an example program in the space. The space can be split into a family of subspaces (`subspace_n`) where n components choose from `{+,-}` and $3 - n$ from `{*,/}`.

As discussed in Section 6.2, our key innovation is to decompose the search based on instruction set size (n) rather than program length (k),

in a hierarchical and adaptive way. This requires a program space representation that enables systematic partitioning while preserving completeness.

In our representation, a program space consists of k components, each able to choose from a set of instructions. These instructions are organized into a tree structure, where the leaf nodes are individual instructions like `+` and `-`. Related instructions are grouped into sets, annotated with sequence numbers (e.g., `{+,-}!2`) for partitioning priority. The `reorder_and_wire` notation indicates that components may be reordered and wired into a program by the synthesizer.

Our representation enables systematic partitioning of the program space while preserving completeness. To illustrate, consider the example in Figure 14, where we have a program space with $k = 3$ components, each able to select from $n = 4$ operators `{+,-,*,/}`. We partition this space by first splitting on the choice node with the smallest sequence number (here, `!1`). This creates subspaces where components choose from different subsets of instructions. In our example, `subspace_n` represents a parameterized family of subspaces where n components choose from `{+,-}` and the remaining $3 - n$ components choose from `{*,/}`. We can further partition these subspaces for finer-grained exploration. For instance, `subspace_2_1` shows a subspace where `{+,-}!2` in `subspace_2` is further split into one `+` and one `-`.

As each program in the original space appears in some subspaces, our partitioning scheme is *lossless* and preserves the cost-model optimality guarantee. This hierarchical approach addresses the combinatorial explosion that would result from naively enumerating all possible combinations of instructions. Instead of generating $C(n + k - 1, k)$ subspaces immediately, we progressively refine the partitioning, solve subspaces in parallel, prune infeasible spaces, and focus computational resources on promising areas of the search space.

6.3.2 *Counter-Example Guided Inductive Synthesis*

Before presenting our parallel synthesis algorithm, we briefly describe the Counter-Example Guided Inductive Synthesis (CEGIS) framework [61, 63] that serves as the base synthesizer for subspaces. CEGIS operates through an iterative loop with two main components:

1. *Proposer*: Generates candidate programs that satisfy the specification on a known set of inputs. We implement this by encoding the program space and specification into SMT constraints. Our encoding, inspired by BRAHMA, will be discussed in Section 6.5.

2. *Challenger*: Checks whether the proposed program satisfies the specification. If not, it generates counter-examples (inputs where the program violates the specification) and adds them to the known input set for the next iteration.

This iterative process continues until either:

1. *Success*: The challenger confirms that the candidate satisfies the specification for all inputs.
2. *Unrealizable*: The proposer proves that no program in the space satisfies the specification on the known inputs.
3. *Timeout*: A pre-defined timeout is reached.

6.3.3 *Parallel Divide-and-Conquer Algorithm*

With our hierarchically partitionable program space representation and the base synthesizer established, we now present our parallel divide-and-conquer algorithm. This algorithm coordinates multiple base synthesizers (workers) to explore different subspaces simultaneously while avoiding combinatorial explosion through effective pruning strategies.

```
1  results ← CheckAllRunningNodeResponse();
2  foreach node,result ← results do
3  |  match result do
4  |  |  case unrealizable → Prune(node);
5  |  |  case success (prog) → return prog;
6  |  |  case timeout → DoNothing();
7  |  end
8  end
9  TryStartNodes();
10 if #running tasks = o then
11 |  if no leaf timed out then return unrealizable;
12 |  return failed;
13 end
```

Algorithm 1: Global scheduler loop body. The global scheduler periodically checks running node status, prunes unrealizable branches, and start new tasks when needed.

Our hierarchical partitioning forms a tree of subspaces with two fundamental properties:

1. A child space is a subset of its parent.

2. The union of all immediate child spaces equals the parent space.

These properties enable the following pruning strategies:

1. *Downward pruning*: When a parent space is proven unrealizable, all its child spaces are also unrealizable and can be pruned without exploration.

2. *Upward pruning*: When all child spaces of a parent are proven unrealizable, the parent space is also unrealizable.

These pruning rules allow us to eliminate large portions of the search space efficiently. Importantly, we leverage an asymmetry in the synthesis process: proving unrealizability is often easier than finding a program that satisfies the specification. The intuition is that proving unrealizability only requires finding a single counter-example for which no candidate program satisfies the specification. In contrast, synthesizing a correct program requires iteratively satisfying the specification across a growing set of counter-examples, which is often a more difficult search. This difference enables us to quickly prune large branches of the search tree by ruling out infeasible branches without exhaustive enumeration.

```
  Data: q: priority search queue of nodes, prioritized by
        ⟨Depth, Priority⟩ lexicographically
  Data: q_s: priority search queue of nodes to split
1 while q is empty and q_s is not empty do
2 |  node ← PopTop(q_s);
3 |  if NotPruned(node) then Split(node);
4 end
5 while q is not empty and #running task < #cores do
6 |  node ←PopTop(q);
7 |  if NotPruned(node) then Start(node);
8 end
```

Algorithm 2: `TryStartNodes` algorithm for task prioritization and splitting.

```
  Data: bestProg: initialized to empty.
  Data: bestCost: initialized to inf or ref cost.
1 responses ← CheckAllRunningNodeResponse();
2 foreach node,response ← responses do
3 |  match response do
4 |  |  case unrealizable → Prune(node);
5 |  |  case success (prog) → UpdateBestProg(prog);
6 |  |  case timeout | gotNewCex → DoNothing();
7 |  end
8 |  if node is running then SyncCurrentCost(node);
9 end
10 TryStartNodes();
11 if #running tasks = 0 then
12 |  match (#timed out leaves, bestProg) do
13 |  |  case (0, empty) → return unrealizable;
14 |  |  case (_, empty) → return failed;
15 |  |  case (0, prog) → return optimal (prog);
16 |  |  case (_, prog) → return mayBeOptimal (prog);
17 |  end
18 end
```

Algorithm 3: Global scheduler loop body for optimal program search.

Our parallel algorithm uses a global scheduler to orchestrate the synthesis process, as shown in Algorithm 1. It runs in a loop, spawns workers for subspaces, periodically checks their results, and manages partition and exploration strategy. It prunes the subspaces when a worker confirms unrealizability. Selecting the next program space to explore is handled by Algorithm 2, which maintains two priority queues:

1. q: Contains program spaces ready for exploration, prioritized first by depth in the tree and then by a random priority.

2. q_s: Contains program spaces that need to be split into smaller subspaces.

When no nodes are ready for exploration (q is empty), the algorithm takes nodes from q_s and splits them, adding their children to q. Then it selects nodes from q to start new workers.

If any node finds a valid program, the algorithm terminates and reports the solution. If all program spaces are proven unrealizable, the algorithm proves that the original program space is unrealizable. If some leaf nodes timed out, we report a synthesis failure but cannot guarantee unrealizability.

For applications requiring a definitive unrealizability proof, we can disable timeouts for leaf nodes, though this may significantly increase runtime.

6.3.4 *Search for Optimal Programs*

In practice, we typically seek not just any valid program but the optimal one according to a cost model, which assigns numerical costs to programs, with lower values indicating better programs. Common costs include instruction count or estimated cycles.

We extend our scheduler in Algorithm 3 to maintain the best-known cost, initially set to infinity or the reference cost. `UpdateBestProg` compares the cost of the program with the current best cost and updates the best program and cost if the new program has a lower cost. This cost serves as a dynamic upper bound that tightens throughout the search process. To effectively propagate cost information to running workers, we modify workers to also report to the scheduler whenever a new counter-example is generated during CEGIS. These reporting events create natural synchronization points to transfer the current best cost to the worker. At these points, the proposer is in a standby state, which allows us to inject additional cost constraints without disrupting the running solver.

Another change not shown in Algorithm 3 is to the worker behavior. Instead of terminating when a solution is found, the worker continues to refine the solution with lower costs. It awaits the current cost from the scheduler, injects cost constraints, and solves incrementally. In this way, it leverages the lemmas learned in synthesizing the earlier programs to find improved results more quickly.

Our approach provides clear optimality guarantees: if all program spaces have been explored with no running tasks remaining and no leaf nodes timed out, we can definitively guarantee that the found solution is cost-model optimal. In cases where some leaf nodes timed out, we report the best solution but acknowledge that we cannot guarantee its optimality. Similar to the unrealizable case, if definitive optimality guarantee is desired, we can turn off the timeout for the leaves and wait indefinitely.

6.4 OPTIMIZATION TO THE PARALLEL SOLVING

While our basic parallel divide-and-conquer approach already offers significant advantages and is the key to breaking the trade-off between instruction set size (n) and program length (k) by decomposing on n, we can further enhance its efficiency through additional optimizations. In this section, we

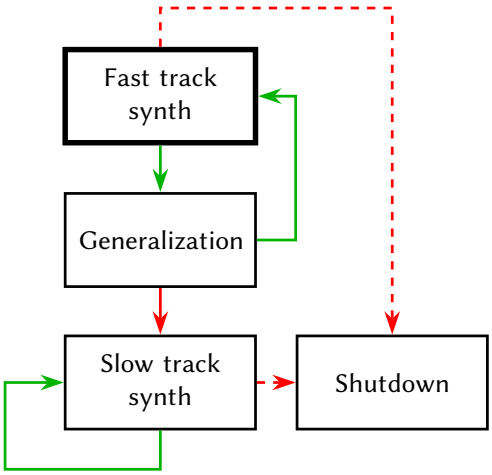


Figure 15: Worker state machine. Green, solid lines are transitions on success. Red lines are transitions on failure.

introduce two key optimizations—two-track synthesis and biased search—that improve the performance of our approach on complex synthesis tasks.

6.4.1 *Two-Track Synthesis*

As described earlier, we use timeouts for workers to avoid wasting resources on program spaces unlikely to give results in a reasonable time. However, this approach risks premature termination of promising tasks. To mitigate this, we use a two-track synthesis approach that identifies promising spaces and adjusts timeouts accordingly. The approach is a state machine for workers, as illustrated in Figure 15.

The key insight behind our two-track approach is that we can use different levels of verification strictness to quickly identify promising program spaces before committing more resources to them. Instead of using a single challenger that performs rigid verification to ensure the equivalence between the specification and the proposed program, we enhance our CEGIS implementation with two classes of challengers:

1. *Easy challengers*: These operate with restricted data ranges or smaller bit-widths. The synthesis on these restricted inputs will be faster, but the result is not guaranteed to be correct on all possible inputs. However, failing to synthesize a program against easy challengers indicates that the program space is unrealizable and can be pruned.

2. *Hard challengers*: These utilize full data ranges and bit-widths. A program that passes all hard challengers is guaranteed to satisfy the complete specification.

Our synthesis proceeds through three phases using these challengers:

1. *Fast-track synthesis*: The worker initially synthesizes using only easy challengers. This phase quickly identifies promising program spaces while filtering out obviously unrealizable ones.
2. *Generalization*: If fast-track synthesis succeeds, rather than immediately proceeding to the hard challengers, we first attempt to generalize the discovered program. This involves creating a smaller, more abstract program space (a "sketch") based on the structure of the fast-track solution. For example, we may use a sketch (not component-based) program with the same structure as the fast-track synthesis result but with symbolic immediates. We then synthesize using this smaller program space with the hard challengers, which is typically faster than synthesis with the full program space.
3. *Slow-track synthesis*: If generalization fails, we fall back to synthesis with hard challengers.

After the fast track successfully finds a solution, we deem this program space as promising and increase its timeout to allocate more resources for the generalization and slow-track phases.

6.4.2 *Biased Search*

Data: p: probability for biased search
Data: q: priority search queue of nodes, prioritized by ⟨Depth, Random priority⟩ lexicographically
Data: q_p: priority search queue of nodes, prioritized by ⟨Distance to the nearest (fast-track) successful ancestor, Depth, Random priority⟩ lexicographically

```
1 while q is not empty and #running task < #cores do
2 |  t ← UniformRandVar(0, 1);
3 |  if t < p then node ← PopTop(q_p); Remove(q, node);
4 |  else node ← PopTop(q); Remove(q_p, node);
5 |  if NotPruned(node) then Start(node);
6 end
```

Algorithm 4: `TryStartNodes` algorithm with biased search, splitting omitted for brevity.

While the two-track synthesis approach helps identify promising program spaces, we also use this information to guide parallel exploration. This insight leads to our biased search strategy, which prioritizes the children of promising program spaces.

Algorithm 4 presents our enhanced task selection strategy with a biased search. Here, the algorithm maintains an additional priority queue q_p which prioritizes nodes first by the distance to the nearest ancestor that successfully passed the fast-track. If no successful ancestor is available, we treat the distance as infinity. When selecting the next task to start, we generate a random value t, and with probability p, we select from q_p (biased search); otherwise, we select from the standard queue q. This probabilistic approach balances exploitation (exploring more promising program spaces) with exploration (exploring diverse program spaces). By prioritizing children of fast-track successful nodes, we focus computational resources on the most promising search spaces and accelerate the discovery of solutions.

6.5 COMPONENT-BASED SYNTHESIS WITH CHOICES IN COMPONENTS

In this section, we describe the base synthesizer constraint encoding for individual program spaces. Our approach extends the formalism described in BRAHMA [34], reusing its concepts of I/O variables, locations, and dataflow relations, while introducing support for instruction choices within each component. This leads to several new constraints and changes to the existing constraints in the BRAHMA framework.

6.5.1 *Notations for Program Space Representation*

We first present the formal representation of our component-based synthesis with choices. A synthesis problem consists of a specification and a program space. Following BRAHMA, a program's specification is a tuple $\langle \vec{I}, \vec{O}, \phi_{spec}(\vec{I}, \vec{O}) \rangle$, where:

1. $\vec{I}$ and $\vec{O}$ are tuples of input and output variables of the program,
2. $\phi_{spec}(\vec{I}, \vec{O})$ is an expression that specifies the desired input-output relationship.

We extend the program space to support choices in components. We call the variables being synthesized *structural variables*, as they determine the final program structure. A program space is formally represented as a set of

N components, each with its own instruction choices C_i and its associated structural variables S_i, along with global structural variables S.

$$\begin{aligned} &\langle \{\langle C_i, S_i\rangle \mid i = 1, \cdots, N\}, S\rangle \\ \text{where } C_i &= \{\langle \vec{I_{i,j}}, \vec{O_{i,j}}, \phi_{i,j}(\vec{I_{i,j}}, \vec{O_{i,j}})\rangle \mid j = 0, \cdots, M_i - 1\} \\ S_i &= \langle E_i, P_i, N_{I,i}, N_{O,i}, \vec{L_{I,i}}, \vec{L_{O,i}}\rangle \\ S &= \langle \vec{L_I}, \vec{L_O}\rangle \end{aligned}$$

This representation has two main parts:

1. *Instruction choices*: C_i is the set of possible instructions for the i-th component. Each component can have M_i different instruction choices, where:
   - $\vec{I_{i,j}}$ and $\vec{O_{i,j}}$ are the I/O variables for instruction choice j (intermediate values).
   - $\phi_{i,j}$ specifies the relationship between inputs and outputs for this instruction.
   - Instruction choices within a component may have different numbers of I/O variables.
2. *Structural variables*: S_i and S represent what is being synthesized:
   - E_i is a Boolean indicating whether component i is enabled.
   - P_i is an integer selecting which instruction to use for component i.
   - $N_{I,i}$ and $N_{O,i}$ are the number of inputs/outputs for the selected instruction.
   - $\vec{L_{I,i}}$ and $\vec{L_{O,i}}$ are "location" values (represented as integers) that encode the program's dataflow graph. The synthesizer ensures that if two I/O variables are assigned the same location, their values must be equal. This constraint, combined with an acyclicity constraint on the locations, defines a valid dataflow DAG.

Intuitively, the structural variables define how we reorder and connect the instructions to form a program, and what choices we make for each component. The location values L constrain the connections between components, and two intermediate values assigned to the same location must be equal. P and N denote the specific instruction choice for each component. The enabled bit E allows the synthesizer to disable a component, which is necessary to handle ISA constraints where not all instruction choices can be legally wired together.

For notational clarity, we use $\vec{V}^k$ for the element of $\vec{V}$ at index k (starting from 0). When element order is irrelevant, we use tuples as sets. We use $\theta(\vec{V})$ for the number of elements in a tuple or set.

We also introduce these intermediate notations:

- $\mathcal{C}$: all the choice sets in all the components (all C_i).
- $\mathcal{S}$: all the structural variables in the set of all S_i along with S,
- $\mathcal{I}$: all the inputs to the choices (union of all $\vec{I_{i,j}}$),
- $\mathcal{O}$: all the outputs of the choices (union of all $\vec{O_{i,j}}$),
- $\mathcal{U}$: all the uses in the program ($\mathcal{I}$ along with $\vec{O}$),
- $\mathcal{D}$: all the definitions in the program ($\mathcal{O}$ along with $\vec{I}$)

We also define a location function L that maps from $\mathcal{U} \cup \mathcal{D}$ to integers:

$$L(\vec{I}^k) = \vec{L_I}^k, \quad L(\vec{O}^k) = \vec{L_O}^k, \quad L(\vec{I_{i,j}}^k) = \vec{L_{I,i}}^k, \quad L(\vec{O_{i,j}}^k) = \vec{L_{O,i}}^k$$

This function is well defined when $\theta(\vec{L_I}) = \theta(\vec{I})$, $\theta(\vec{L_O}) = \theta(\vec{O})$, $\theta(\vec{L_{I,i}}) = \max_j \theta(\vec{I_{i,j}})$, and $\theta(\vec{L_{O,i}}) = \max_j \theta(\vec{O_{i,j}})$. We will assume that L is well-defined throughout our discussion.

6.5.2 *Synthesis Constraints*

The synthesis constraints with our program space representation can be described as:

$$\exists \mathcal{S}.\phi_{wfp}(\mathcal{S}) \wedge (\forall \vec{I}.\forall \vec{O}.\exists \mathcal{I}, \exists \mathcal{O}.\phi_{conn}(\mathcal{S}, \mathcal{D}, \mathcal{U}) \wedge \phi_{lib}(\mathcal{S}, \mathcal{I}, \mathcal{O}) \wedge \phi_{spec}(\vec{I}, \vec{O}))$$

This formula states that our synthesis problem is to find a set of structural variables $\mathcal{S}$ such that for all program input/output pairs ($\vec{I}$ and $\vec{O}$), there exists a set of intermediate values ($\mathcal{I}$ and $\mathcal{O}$) that satisfy the constraints.

6.5.3 *Encoding Well-Formed Programs*

We now establish the constraints ϕ_{wfp} for the structural variables $\mathcal{S}$ to be valid. A valid assignment to these variables leads to a well-formed program in static single-assignment (SSA) form within our program space.

First, we need to assign integers to locations in a principled way. Only equality relationships matter for location values, so we can choose a canonical assignment for them. The canonicity constraint ensures the following properties and simplifies subsequent constraints:

- Locations are assigned non-negative integers smaller than the total number of definitions.
- Program inputs receive the smallest possible numbers.
- Outputs of a component receive consecutive location values.

$$\phi_{\text{cano}} = \bigwedge_{v \in \mathcal{D} \cup \mathcal{U}} 0 \leqslant L(v) < \theta(\mathcal{D}) \wedge \bigwedge_{0 \leqslant k < \theta(\vec{I})} L(\vec{I}^{k}) = k \wedge \bigwedge_{\substack{0 \leqslant i < \theta(\mathcal{C}), \\ 0 \leqslant j < \theta(C_i), \\ 0 \leqslant k < \theta(\vec{O_{i,j}}) - 1}} L(\vec{O_{i,j}}^{k}) = L(\vec{O_{i,j}}^{k+1}) - 1$$

The canonicity constraint does not prevent multiple definitions from sharing the same location, which would violate SSA form. Therefore, we need a consistency constraint:

$$\phi_{\text{cons}} = \bigwedge_{x,y \in \mathcal{D}, x \neq y} L(x) \neq L(y)$$

In a well-formed program, variables must be defined before they are used, and the data dependency graph must be a DAG. The canonicity constraint simplifies how we enforce acyclicity. To ensure the data dependency graph is a DAG, we simply require that within any component, the location of a definition must be greater than the locations of its uses:

$$\phi_{\text{acyc}} = \bigwedge_{0 \leqslant i < \theta(\mathcal{C}), 0 \leqslant j < \theta(C_i), x \in \vec{I_{i,j}}, y \in \vec{O_{i,j}}} L(x) < L(y)$$

The choice constraint ensures that a valid instruction is selected for each component:

$$\phi_{\text{choi}} = \bigwedge_{0 \leqslant i < \theta(\mathcal{C})} 0 \leqslant P_i < \theta(C_i) \wedge N_{I,i} = \theta(\vec{I_{i,P_i}}) \wedge N_{O,i} = \theta(\vec{O_{i,P_i}})$$

Another thing to note is that each component can choose from operators with different numbers of inputs and outputs and can be disabled. This means we must ensure that all uses only reference valid, enabled outputs. Here, the antecedent identifies a use x (from an enabled component u) that is wired to an output of component d. The consequent then asserts that component d must be enabled and that the specific output being used is valid.

$$\phi_{\text{valid}} = \bigwedge_{0 \leqslant u, d < \theta(\mathcal{C}), x \in \vec{I_{u,P_u}}} \begin{pmatrix} (E_u \wedge \vec{L_{O,d}}^{0} \leqslant L(x) < \vec{L_{O,d}}^{0} + \theta(L_{O,d})) \Rightarrow \\ (E_d \wedge L(x) - \vec{L_{O,d}}^{0} < N_{O,d}) \end{pmatrix}$$

The well-formed constraint can then be defined as the conjunction of the five conditions:

$$\phi_{wfp} = \phi_{cano} \wedge \phi_{cons} \wedge \phi_{acyc} \wedge \phi_{choi} \wedge \phi_{valid}$$

6.5.4 *Encoding Dataflow Semantics*

For dataflow semantics with choices, we only constrain values involved in execution based on structural variables. The input-output relation for the instructions should only apply when the instruction is chosen and the component is enabled:

$$\phi_{lib} = \bigwedge_{0 \leqslant i < \theta(\mathcal{C}), 0 \leqslant j < \theta(C_i)} (E_i \wedge j = P_i) \Rightarrow \phi_{i,j}(\vec{I_{i,j}}, \vec{O_{i,j}})$$

We also need a constraint modeling connections between components such that the I/O variables should be equal when the locations are equal, the components are enabled, and the I/O variables are associated with chosen instructions.

$$\phi_{conn} = \bigwedge_{u \in \mathcal{U}, d \in \mathcal{D}} (\text{chosen}(u) \wedge \text{chosen}(d) \wedge L(u) = L(d)) \Rightarrow u = d \quad \text{where}$$
$$\text{chosen}(x) = x \in \vec{I} \vee x \in \vec{O} \vee \bigvee_{0 \leqslant i < \theta(\mathcal{C})} (E_i \wedge (x \in \vec{I_{i,P_i}} \vee x \in \vec{O_{i,P_i}}))$$

6.5.5 *Solving the Constraint*

Putting the constraints together, we get our synthesis constraint:

$$\exists \mathcal{S}. \phi_{wfp}(\mathcal{S}) \wedge (\forall \vec{I}. \forall \vec{O}. \exists \mathcal{I}. \exists \mathcal{O}. \phi_{conn}(\mathcal{S}, \mathcal{D}, \mathcal{U}) \wedge \phi_{lib}(\mathcal{S}, \mathcal{I}, \mathcal{O}) \wedge \phi_{spec}(\vec{I}, \vec{O}))$$

This constraint has nested quantifiers and cannot be solved efficiently with modern solvers. While the CEGIS algorithm [63] has been proposed to address this problem, and the original Brahma effectively applies the approach with angelic variables, Brahma formulates the problem as a single, monolithic constraint system. The monolithic nature of this formulation makes it difficult to support composable features like sub-procedures, which can be used to implement techniques like context-aware window decomposition [48] and user-introduced pseudo-instructions. We address this limitation by reformulating the construction of the synthesis constraint as an angelic programming [9] problem, which is inherently composable.

6.5.6 *Finite Synthesizer with Angelic Programming*

To handle the nested quantifiers, we first skolemize the constraint. This standard logical transformation effectively replaces the existentially-quantified intermediate variables ($\mathcal{I}$, $\mathcal{O}$) with functions over the universally-quantified program inputs/outputs $\vec{I}$ and $\vec{O}$. For a finite set of I/O pairs (concrete instantiations of $\vec{I}$ and $\vec{O}$), we can further replace the universal quantifier with a conjunction:

$$\exists \mathcal{S}. \left(\begin{array}{l} \phi_{\text{wfp}}(\mathcal{S}) \wedge \\ \left(\exists \mathcal{I}, \exists \mathcal{O}. \bigwedge_{\vec{I},\vec{O}} \left(\begin{array}{l} \phi_{\text{conn}}(\mathcal{S}, \mathcal{D}(\vec{I},\vec{O}), \mathcal{U}(\vec{I},\vec{O})) \wedge \\ \phi_{\text{lib}}(\mathcal{S}, \mathcal{I}(\vec{I},\vec{O}), \mathcal{O}(\vec{I},\vec{O})) \wedge \phi_{\text{spec}}(\vec{I},\vec{O}) \end{array} \right) \right) \end{array} \right)$$

For each I/O pair, we can now generate its constraints as a relation ϕ over $\vec{I}$, $\vec{O}$, $\mathcal{S}$, and intermediate variables ($\mathcal{I}(\vec{I},\vec{O})$ and $\mathcal{O}(\vec{I},\vec{O})$). We view this as an angelic programming problem [9]. As shown in Figure 16, ϕ can be transformed into an angelic interpreter that produces symbolic outputs for given inputs. The interpreter generates *fresh* symbolic variables for the intermediate values and asserts the constraints that define their relationships. Crucially, each call to this angelic interpreter generates a distinct set of fresh variables, ensuring that they are not interfering with each other. Here, the angelic programming and constraints can be managed as monadic effects in HieraSynth's host library Grisette [40], and the interpreter can be written as a conventional monadic interpreter and be used anywhere an interpreter is expected.

```
interpret(input):
  output,intermediate = fresh_symbolic_variables()
  assert(phi(input,output,structural,intermediate))
  return (output)
```

Figure 16: Transforming a constraint into an angelic interpreter with fresh variables and assertions.

6.5.7 *Composable Synthesis with Angelic Programming*

The primary motivation for using angelic programming is to provide a clean and composable interface for super-optimization. This allows us to treat both the concrete program semantics (which is naturally defined as interpreters) and symbolic program space semantics (which become angelic interpreters) under a unified interpreter abstraction. The uniformity

is crucial for seamlessly supporting features like sub-procedures. We can now offer a composable interface where both program specifications and instruction semantics are expressed as interpreters. The monad M encapsulates the effects, such as generating fresh angelic variables and asserting constraints:

$$\text{interpret}(\mathit{space}) \coloneqq \vec{I} \mapsto M(\vec{O}) \quad \text{interpret}(\mathit{inst}_{i,j}) \coloneqq \vec{I_{i,j}} \mapsto M(\vec{O_{i,j}})$$

This monadic approach encapsulates angelic programming in the interpreter, allowing concrete programs and component-based program spaces to be mixed freely, as all are implemented as composable monadic interpreters. Figure 17 illustrates two applications: context-aware window decomposition and user-defined pseudo instructions.

```
int32 programSpace(int32 a, int32 b) {
  reorder_and_wire { ... }
}
int32 program(int32 a, int32 b) {
  int32 add = a + b;
  int32 mul = a * b;
  return programSpace(add, mul);
}
```

(a) Using program spaces as sub-routines to synthesize with a prologue.

```
int32 pseudoMulAdd(int32 a, int32 b) {
  return a + a * b;
}
int32 programSpace(int32 a, int32 b) {
  reorder_and_wire {
    2 * {..., pseudoMulAdd, ...}
  }
}
```

(b) Using concrete programs as sub-routines to allow user-introduced pseudo instructions.

Figure 17: Using sub-routines to provide a more flexible and composable interface for super-optimization.

In Figure 17a, a program fragment under super-optimization can be part of a larger program. The code can leverage assumptions about its calling context and be simplified accordingly — this is the context-aware window decomposition idea from Lens [48]. We will show an example of this in Section 6.7.5. Another application (Figure 17b) is using concrete sub-routines as pseudo-instructions. When users identify useful code patterns, they can

provide them as sub-routines, potentially reducing the k required to synthesize an optimization. We will discuss this further in Section 6.6.3.

6.6 ADAPTING HIERASYNTH TO SUPER-OPTIMIZATION FOR RISC-V VECTOR CODE

In this section, we describe how we adapt HieraSynth to build a RISC-V Vector (RVV) code super-optimizer. We implement HieraSynth as a reusable library with the Grisette [40] symbolic evaluation framework, enabling super-optimizer developers to obtain a program synthesizer simply by defining the instruction semantics with standard monadic Haskell code. The main challenge is adapting our framework to the complex RVV type system, particularly for our two-track synthesis approach. Given that conventional SIMD ISAs like AVX or NEON have simpler type systems, we believe our approach is generalizable. The remainder of this section details our solutions for RVV, including our use of sub-procedures to scale synthesis and our heuristics for inferring a relevant program space from a reference implementation.

6.6.1 *RISC-V Vectors*

The RISC-V "V" extension introduces a flexible vector programming model with distinctive features:

- *Scalable vector length (VLEN)*: Hardware vendors can implement different vector register lengths (e.g., SiFive X280 uses 512-bit registers, while P670 is 128 bits). RVV code is usually (and is encouraged to) written in a portable, architecture-agnostic way.
- *Vector grouping*: Instructions can operate across multiple registers. This is controlled through the "vector group multiplier" (LMUL) parameter. For example, the RVV type `vuint8m2_t` represents an 8-bit unsigned vector, with LMUL=2, meaning it spans two registers. RVV also allows vectors to occupy only part of a register using fractional LMULs, like `vuint8mf2_t` with LMUL=1/2. Note that the choice of LMULs can affect performance: on SiFive X280 with a 256-bit vector ALU but 512-bit VLEN, using LMUL=1/2 can halve execution time compared to LMUL=1.
- *Selective operations*: Most RVV instructions support masking and configuring to process only the first `vl` elements. Elements outside the

mask or exceeding `vl` can be configured to "agnostic" (undefined values) or "undisturbed" (preserved values) modes.

These features provide developers with fine-grained control and facilitate optimizations. However, they also increase programming complexity for both developers and super-optimizers as numerous instruction and configuration choices need to be made.

6.6.2 *RVV Types and Two-Track Synthesis*

For two-track synthesis (described in Section 6.4.1), the apparent approach is using smaller VLEN for easy challengers. However, this isn't always efficient since the minimum 128-bit VLEN specified by RVV remains challenging for solvers, especially with complex non-linear operations.

To make fast-track more effective, we further reduce bit-width to hypothetical machines with a modified RVV type system. Instead of using absolute bit-widths (e.g., the 8 in `vint8m1_t`, or the 64 in `uint64_t`), our modified type system represents bit-width as a ratio relative to the machine's scalar register width (XLEN). For example, assuming 64-bit machines, `vint8m1_t` will become `vint{1/8}m1_t`, while `uint64_t` will become `uint{1}`. This enables systematic machine configuration scaling with reasonable semantics, using the ratio as an invariant. Programs with these types become polymorphic across bit-width reductions — a program running with 32-bit data on a 64-bit machine with 128-bit vectors can also process 8-bit data on a hypothetical 16-bit machine with 32-bit vectors.

As our super-optimizer uses the reference code as the specification, scaling doesn't always work when magic numbers assume specific bit-widths. We mitigate this by providing multiple scaling models:

- Scale the reference program with reduced bit semantics.
- Only scale the inputs/outputs with zero/sign extension and truncation.
- Use only real 64-bit configurations (as a last resort).

The choice of scaling mode is currently left to users. Automatic mode detection or parallel searching with multiple modes remains future work.

6.6.3 *Sub-Procedures*

As shown in Section 6.5.7, reformulating synthesis constraints as angelic programming enables sub-procedures. Our RVV synthesizer leverages this to help synthesize more complex programs.

As a general super-optimizer, we refrain from adding many pseudo instructions, as the benefit is unclear. We include some scalar pseudo instructions recommended by RISC-V specification and recognized by assemblers. We add only one type of vector pseudo instrinsic: a set of operations for manipulating mask vectors using standard vector instructions. We included this because we found the pattern to be both very common and general. One example of such pseudo instruction is shown in Figure 18. We believe that it is useful for users to develop their own domain-specific pseudo instructions to further aid synthesis.

```
vbool64_t vadd_mm(vbool64_t a, vbool64_t b) {
  size_t vl = __riscv_vsetvlmax_e8mf8();
  vuint8mf8_t av =
    __riscv_vlmul_trunc_u8mf8(__riscv_vreinterpret_u8m1(a));
  vuint8mf8_t bv =
    __riscv_vlmul_trunc_u8mf8(__riscv_vreinterpret_u8m1(b));
  vuint8mf8_t rv = __riscv_vadd(av, bv, vl);
  return __riscv_vreinterpret_b64(__riscv_vlmul_ext_u8m1(rv));
}
```

Figure 18: Using reduced LMUL and `vadd` to shift masks.

6.6.4 *Program Space Inference*

Though our system can handle large sets of choices, it is not beneficial to add all of them to the program space when doing the synthesis. We adopt simple heuristics to infer a program space from the reference programs: we collect all types and instructions present in the reference program and add all potentially relevant instructions to the choices.

For example, if multiple vector types are used, we introduce all possible conversion/narrowing/widening instructions. If we see multiplication, we add multiplications-related instructions. In contrast, we will not add multiplication to the choices when only linear arithmetic is present. Despite not using all instructions for an arbitrary program, we ensure all possibly useful instructions are added to the best of our knowledge. The resulting

program space is still comprehensive and allows us to synthesize interesting optimizations, as shown in Section 6.7.

The instructions in this inferred set are then clustered into subsets using simple heuristics (e.g., grouping all addition-related instructions). Developing a more sophisticated, automated grouping or prioritizing methodology remains an area for future work.

6.7 EVALUATION

We evaluate HIERASYNTH to understand the following research questions.

RQ1 How does HIERASYNTH compare to the original BRAHMA?

RQ2 How effectively can HIERASYNTH synthesize vector programs and prove their cost-model optimality?

RQ3 Can HIERASYNTH overcome the fundamental trade-off between output program size k and instruction set size n?

RQ4 Can HIERASYNTH discover optimizations beyond those identified by human experts?

RQ5 How efficiently does HIERASYNTH scale with increasing parallelism?

6.7.1 *Benchmarks*

We evaluate HIERASYNTH on 25 scalar and 26 vector benchmarks. Statistics are in Table 7 and Table 8.

The scalar benchmark set is a widely used benchmark for evaluating super-optimizers drawn from BRAHMA [34], with 25 bit-manipulation programs from Hacker's delight [74] rewritten using RISC-V instructions. For these benchmarks, HIERASYNTH synthesizes using a subset of RISC-V basic ISA with Zicond (integer conditional) and Zbb (basic bit-manipulation) extensions. We include four register-immediate instructions (`andi`, `slli`, `srai`, and `srli`), and 1 component returning an immediate (equivalent to the `li` pseudo-instruction). Immediates are treated as angelic values [9] to be synthesized by the solvers. We extend the program space for p20 with division and p25 with multiplication instructions from the M extension. We use a simple cost model where most instructions have a cost of 1, with higher costs assigned to multiplication and division instructions since they are usually more expensive on modern CPUs. The initial search bound is set just above the reference program's cost by 1, ensuring our search looks for

a solution that is *at least as good according to this cost model*, and the search space always has a valid solution.

Table 7: Benchmark statistics for Highway and vqsort benchmarks. Program space metrics include number of components (k), choices per component (n), theoretical number of subspaces ($C(n+k-1,k)$) if fully partitioned (Leaf), and number of well-typed programs in the program space (Prog). Performance metrics show instruction count (Inst) and formal cost (Cost) according to our cost model for both reference (R) and best synthesized (B) programs. Question marks indicate tasks where HIERASYNTH timed out before finding a solution, so we do not yet know the required k for this task. The Highway benchmarks and the vqsort benchmarks are sorted first by k and then by n.

Name	Space				Inst		Cost	
	k	n	Leaf	Prog	R	B	R	B
ZeroExtendResizeBitCast	1	726	7.3e02	2.0e00	12	1	5	0
ConcatLowerLower	2	65	2.1e03	3.7e02	5	2	4	2
ConcatUpperLower	2	65	2.1e03	3.7e02	6	2	8	2
ConcatUpperUpper	2	65	2.1e03	3.7e02	7	2	12	2
ZeroExtendResizeBitCast.Ext	3	473	1.8e07	2.2e03	11	3	22	6
ConcatLowerUpper	4	65	8.1e05	4.2e06	6	4	8	4
AddSub	4	594	5.2e09	6.2e08	11	4	10	5
InsertLane	5	283	1.6e10	5.0e10	10	5	13	6
OddEvenBlocks	5	365	5.5e10	3.8e12	9	5	16	8
Lt128.WithPrologue	6	118	4.2e09	2.1e15	18	6	28	13
MulEven	6	450	1.2e13	2.3e12	13	6	24	19
MulOdd	6	450	1.2e13	2.3e12	13	6	24	19
Min128.WithPrologue	7	116	6.7e10	7.3e23	14	7	22	15
Lt128	7	214	4.5e12	6.3e17	18	7	28	11
Min128	8	212	1.2e14	1.6e22	14	8	22	13
PrevValue_32	2	40	8.2e02	4.4e01	4	2	4	2
PrevValue_64	2	40	8.2e02	4.4e01	4	2	4	2
PrevValue_128	5	286	1.7e10	4.4e10	16	5	26	6
SortPairsDistance4_64	6	77	3.5e08	8.7e10	10	6	12	8
SortPairsDistance1_64	6	212	1.4e11	8.1e12	12	6	18	9
SwapAdjacentPairs_32	6	231	2.3e11	7.9e12	10	6	12	10
SwapAdjacentPairs_64	6	356	2.9e12	2.9e14	11	6	20	9
SwapAdjacentQuads_32	6	714	1.9e14	2.9e14	13	6	20	10
SortPairsDistance1_32	7	514	2.0e15	2.2e17	18	7	28	20
SortPairsDistance1_128	?	284	?	?	26	?	50	?
SortPairsDistance4_32	?	542	?	?	21	?	26	?

Table 8: Hacker's delights benchmark statistics.

Name	Space				Inst		Cost		Name	Space				Inst		Cost	
	k	n	Leaf	Prog	R	B	R	B		k	n	Leaf	Prog	R	B	R	B
p01	2	31	5.0e02	9.3e03	3	2	2	2	p14	4	31	4.6e04	2.3e10	5	4	4	4
p02	2	31	5.0e02	9.3e03	3	2	2	2	p15	4	31	4.6e04	2.3e10	5	4	4	4
p03	2	31	5.0e02	9.3e03	2	2	2	2	p16	1	31	3.1e01	3.0e02	7	1	6	1
p04	2	31	5.0e02	9.3e03	3	2	2	2	p17	4	31	4.6e04	1.1e09	5	4	4	4
p05	2	31	5.0e02	9.3e03	3	2	2	2	p18	3	31	5.5e03	2.5e06	7	3	6	3
p06	2	31	5.0e02	9.3e03	3	2	2	2	p19	6	31	1.9e06	3.4e17	6	6	6	6
p07	2	31	5.0e02	9.3e03	4	2	3	2	p20	6	32	2.3e06	7.9e14	7	6	39	39
p08	2	31	5.0e02	9.3e03	4	2	3	2	p21	7	31	1.0e07	6.6e21	13	7	12	7
p09	2	31	5.0e02	9.3e03	4	2	3	2	p22	3	31	5.5e03	2.5e06	12	3	10	2
p10	4	31	4.6e04	2.3e10	5	4	4	4	p23	1	31	3.1e01	6.3e01	24	1	15	1
p11	2	31	5.0e02	8.1e04	3	2	3	2	p24	5	31	3.2e05	7.0e11	17	5	12	4
p12	3	31	5.5e03	3.5e07	5	3	4	3	p25	1	35	3.5e01	3.5e02	18	1	24	3
p13	2	31	5.0e02	9.3e03	5	2	4	1									

Our first set of vector benchmarks include operators from Highway [32], a C++ library providing portable SIMD/vector intrinsics optimized for various platforms, including RVV. While Highway operators aim to achieve performance within 10–20% of hand-coded assembly, we identified 12 operators for further optimization with HieraSynth. Section 6.7.5 explains the `Lt128` operator, and Highway's documentation [31] covers other operators semantics.

Highway operators are polymorphic to different RISC-V vector register groupings (LMUL) and vector lengths (VL). We instantiate the operators with LMUL=1 and full vectors and compile them to LLVM IR with `clang -O3`. The `ZeroExtendResizeBitCast` operator is benchmarked with both extension from LMUL=1 to LMUL=2 and truncation from LMUL=1 to LMUL=1/2. For most benchmarks, the program space is fully inferred from the source program (Section 6.6.4). For `Lt128` and `Min128`, we include two benchmarks each, one using the inferred program space for the entire operator and the other incorporating a prologue. Our cost model for vector synthesis is derived from the LLVM cost model for SiFive X280. We verify correctness up to 512-bit vectors for all benchmarks.

In addition, we gathered 11 kernels from the vqsort (vectorized quick sort) algorithm [8]. Some kernels do not cover all 32/64/128-bit values either because they are not supported or are proven optimal with the inferred program space.

All experiments ran on three Rocky Linux 9.4 servers, each with two Intel Xeon Platinum 8160M CPUs (48 cores/96 threads total) and 768GB RAM. We use bitwuzla 0.7.0 SMT solver [47] with a 3600s timeout for benchmarks, 500s for fast-track tasks, and 1500s for slow-track tasks. We report mean runtime across three runs per benchmark.

6.7.2 *RQ1: Comparison with Brahma*

BRAHMA is our primary baseline for direct comparison. The key methodological difference is that BRAHMA does not explore subspaces automatically but relies on human insights with a *standard library* approach to reduce the effective ISA size. Following Brahma [34], we built a standard library of 12 components for commonly used RISC-V instructions. This fixed set proves insufficient for optimal results in 14 of the 25 scalar benchmarks, requiring *human insights* to identify additional components. Trying to reduce reliance on human insights can greatly limit the synthesizable program size, as seen in SOUPER [53]. For the fairest comparison, we simulate BRAHMA's *best-case* performance by providing it with the exact set of components that HIERASYNTH discovered were necessary for synthesizing the cost-model optimal programs. With this advantage, BRAHMA's effective ISA size is significantly smaller than HIERASYNTH's full 31–35 instruction choice space.

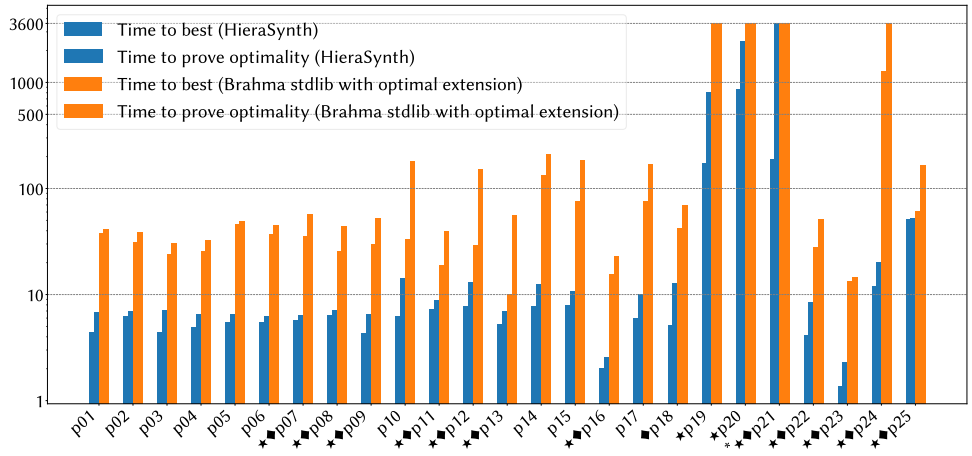


Figure 19: Synthesis time (seconds) for the benchmarks. ♦: HIERASYNTH found cheaper solutions than reference. ∗: solution found, but not proven optimal (timed out) in 3600s using 72 cores. ⋆: Hacker's delight benchmarks where extra components are required to achieve the best solution with BRAHMA's standard library approach.

We do a more comprehensive evaluation than the original BRAHMA evaluation, which only reported time to the first valid solution. Figure 19 reports two key metrics: time to synthesize the solution with the best-known cost, and the time required to prove its cost-model optimality within the specified program space. Note that HIERASYNTH can take advantage of parallelism with 72 cores, while BRAHMA is inherently sequential. The results demonstrate HIERASYNTH's consistent superiority, achieving 1.20–106.73× acceleration in finding the best solution and 3.16–17.03× acceleration in proving optimality.

Notably, HIERASYNTH maintains its advantage even when using a single core (detailed results omitted due to space constraints but derivable from the parallel synthesis speedup figures). This demonstrates that HIERASYNTH, by decomposing on n in a hierarchical and adaptive way, is more effective than BRAHMA, enabling exploration of complex program spaces and synthesizing cost-model optimal solutions with large ISAs.

6.7.3 *RQ2: Scaling to Vector Synthesis*

The scalar benchmarks use a relatively small instruction set and cannot fully demonstrate HIERASYNTH's capability to synthesize with large vector ISAs. In Figure 20, we evaluate HIERASYNTH on synthesizing vector programs. We do not compare with BRAHMA for these vector benchmarks, as constructing a standard library covering the diverse range of vector instructions needed is impractical. The sheer number of vector instructions and their complex semantics make manual library construction or instruction selection extremely challenging for human experts.

These benchmarks reveal our tool's capability boundaries. Among the 26 benchmarks on Highway and vqsort, our approach successfully synthesizes programs with lower cost than the reference for 25 benchmarks, and proves cost-model optimality for 19 of them. The synthesis time generally correlates with program complexity. These results demonstrate that our tool can handle complex vector kernels with up to 8 output instructions, demonstrating a significant advance in super-optimization for modern vector ISAs.

For the cases where synthesis failed or optimality couldn't be proven within the time limit, we analyzed the exploration patterns by tracking the number of nodes explored at each depth. Figure 21 shows these patterns for four representative benchmarks: two successful cases (`AddSub` and `MulOdd`) where optimality was proven, one partially successful case (`Min128`) where a solution was found but optimality wasn't proven, and one failure case (`SortPairsDistance1_128`). The presence of pending nodes at deeper

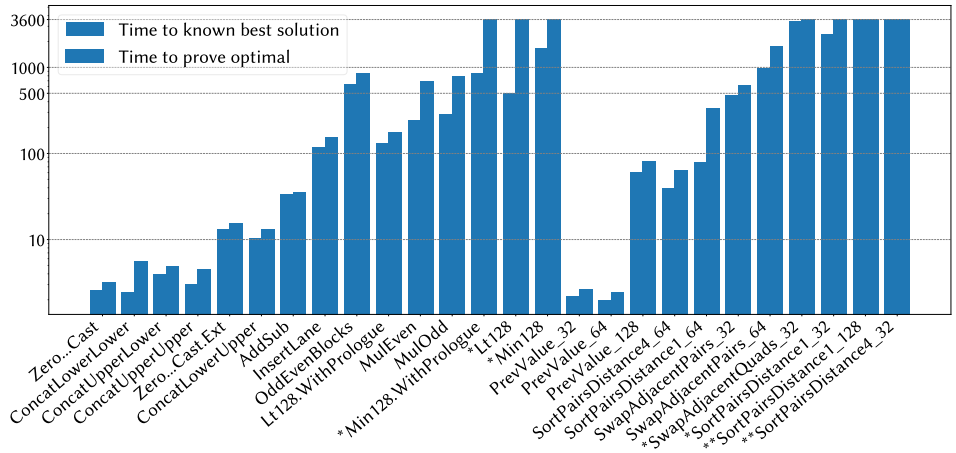


Figure 20: Synthesis time (seconds) with HIERASYNTH for vector benchmarks. Each using 72 cores with 3600s timeout. The Highway and vqsort benchmarks are sorted by k and then by n. *: HIERASYNTH cannot prove the optimality of the synthesized program. **: HIERASYNTH cannot synthesize a solution.

depths (shown by dotted lines in Figure 21) for complex cases indicates that the search space continues to grow beyond our computational resources.

Successful cases show node exploration peaking at moderate depths, indicating effective pruning. However, some benchmarks like `Min128` and `SortPairsDistance1_128`, with $n \approx 200$–300 and $k \geqslant 8$, experience explosive growth, exceeding the search capacity. Future heuristics to detect unrealizable subspaces more efficiently could help address this. Though more parallelism might also help, we lack the resources to explore it. Despite these challenges, HIERASYNTH successfully handles many complex benchmarks, outperforming prior super-optimizers by a huge margin.

6.7.4 *RQ3: The k-vs-n Trade-off*

Super-optimizers face a fundamental trade-off between output program size (k) and instruction set size (n). Since different tools use different instruction sets, we focus on analyzing this k-vs-n relationship rather than direct tool comparisons. Our analysis reveals that existing synthesis techniques follow a power law constraint: $\log(k) = -0.81 * \log(n) + 5.10$ with $R^2 = 0.862$, as shown in Figure 22. This strong correlation suggests a fundamental limitation in previous approaches. Points deviating from the power-law line (MINOTAUR [39] and Bansal and Aiken [6]) may result from their different handling of immediate constants, where MINOTAUR (and HI-

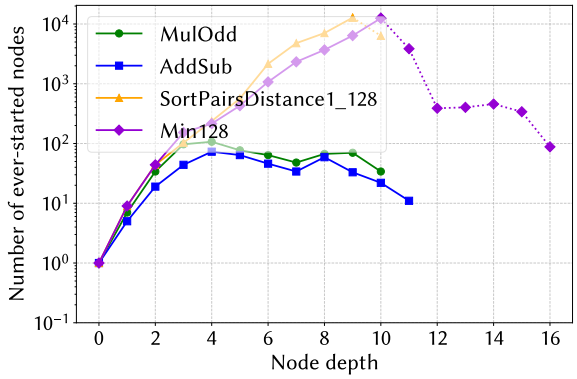


Figure 21: The number of nodes ever started at each depth. AddSub and MulOdd finished the optimality proof. Min128 synthesized a solution but not proven optimal. SortPairsDistance1_128 failed to synthesize a solution. Here Min128 and SortPairsDistance1_128 ran for two days on 72 cores to collect the statistics. The dotted line indicates depths with pending nodes so the number cannot represent how many nodes to start there.

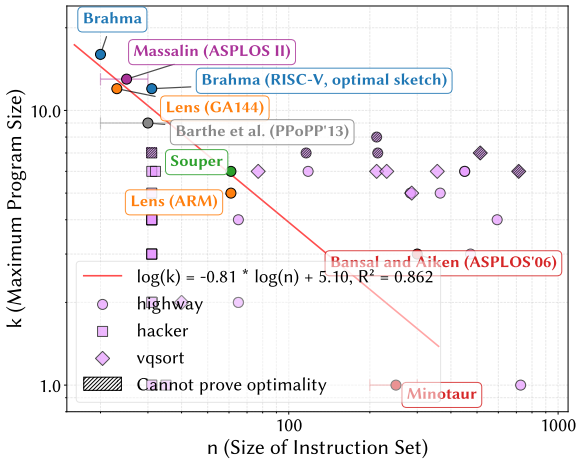


Figure 22: Trade-off between k and n. We report *maximum* k and n for the existing tools, which might not be achievable simultaneously. For HieraSynth, we report the actual k and n for each benchmark. Points below the line indicate benchmarks that are too easy for HieraSynth.

eraSynth) can synthesize arbitrary immediates, while Bansal and Aiken only consider a small, fixed set of immediates.

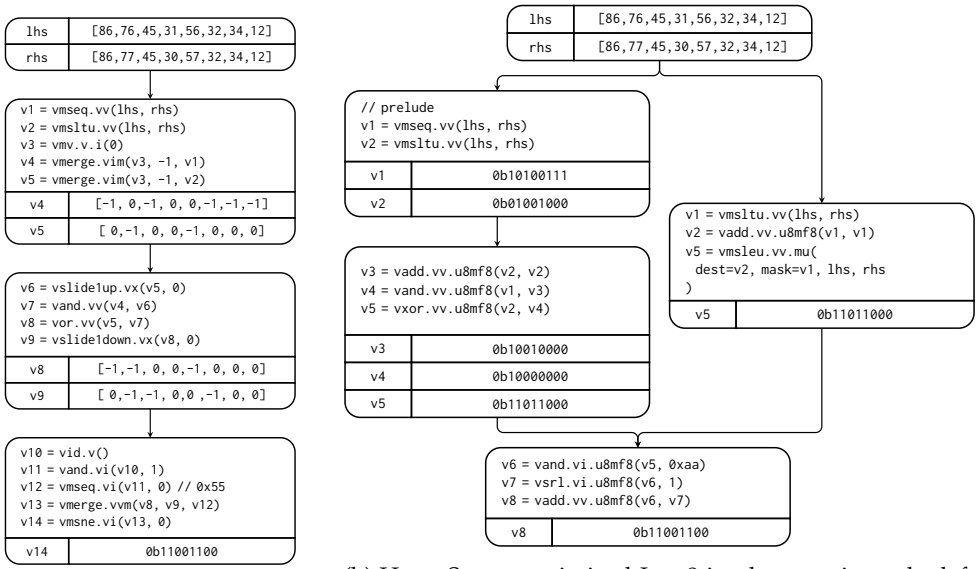
In contrast, HieraSynth successfully solves many benchmarks that lie significantly beyond this power law constraint. When compared directly with existing tools, our approach can synthesize programs with both larger k and larger n than Souper and Minotaur. For Brahma, as detailed in our

response to RQ1, our tool consistently outperforms it even when BRAHMA is given the advantage of human knowledge in addition to the standard library.

This ability to break through the power law enables our tool to address real-world synthesis problems that were previously intractable, particularly in the domain of vector programming, where the programs are complex and the instruction sets are very large.

6.7.5 *RQ4 & Case Study: Synthesis of Lt128 Operator in Highway*

To assess HIERASYNTH's capability to discover non-trivial optimizations, we examine its synthesis of the `Lt128` operator from Highway. This operator compares 128-bit integers formed by concatenating adjacent 64-bit elements from two input vectors, returning a mask indicating whether each 128-bit integer in the first vector is less than the corresponding one in the second. Each comparison corresponds to 2 (duplicated) bits in the resulting mask. See Figure 23a for an illustration.



(a) Original Highway implementation of Lt128

(b) HIERASYNTH-optimized Lt128 implementations: the left path is synthesized with a prologue; the right path is synthesized as a whole.

Figure 23: Lt128 operator implementations. We assume VLEN=512 and LMUL=1, thus there are 8 64-bit values in each vector. The first lane is shown as the right-most value in vectors. Bitcasts are omitted for brevity.

The original `Lt128` implementation follows a step-by-step approach: (1) 64-bit element-wise less than and equal to comparisons, (2) assembly of in-

termediate results, (3) duplication of results for adjacent bits. This approach requires mask manipulation operations not directly provided by RVV (like shifting and ternary operations). The conventional trick involves converting masks to vectors (with each mask bit corresponding to one element), applying vector instructions, then converting back to masks. This is a recognized technique, but known to be inefficient [50]. In the original implementation (Figure 23a), `v4` and `v5` are such converted masks.

This implementation is suboptimal, particularly on X280. As masks occupy only a fraction of vector registers, using LMUL=1 instructions on converted masks takes twice the time of mask operations or vector operations with fractional LMUL. We tried to optimize this program with an early version of our tool, which was not scalable enough to synthesize the operator as a whole. However, it seemed that the first two comparisons must be present in any optimized solution. Under this assumption, we successfully synthesized the left path in Figure 23b using the comparisons as a prologue. The program bitcasts masks to LMUL=1/8 vectors and use corresponding vector instructions, reducing cycle count by more than 50%. For Lt128 instantiated with LMUL=8, similar optimization yields approximately 4x speedup. The impressive formal cost of the synthesized program, and an observed empirical speedup in the similar scale, led us to believe this version was likely the best achievable. We thus sent a patch to Highway developers, which was merged upstream.

With later improvements to HieraSynth, we tried to synthesize `Lt128` without a prologue. Surprisingly, HieraSynth discovered an implementation with an even lower formal cost that is also empirically faster (right path in Figure 23b) leveraging masked undisturbed intrinsics to fuse the first two steps. While this final version could theoretically be found via peephole optimization on the *first synthesized* program, no prior system could synthesize the first synthesized program itself. This is a task beyond the scope of local peephole techniques.

This case highlights HieraSynth's ability to surpass expert optimizations by exploring innovative paths without the biases that experts' familiarity with established solutions can introduce.

6.7.6 *RQ5: Parallel Speedup*

To evaluate HieraSynth's parallel scalability, we present the speedup with different numbers of cores for synthesizing the best solution and proving optimality in Figure 24 and Figure 25. In each figure, we omit benchmarks that finished in 10 seconds on a single core, as short tasks have limited potential for parallel acceleration. The benchmarks are arranged in ascending

order of single-core time to highlight the scaling trend from less to more computationally intensive tasks.

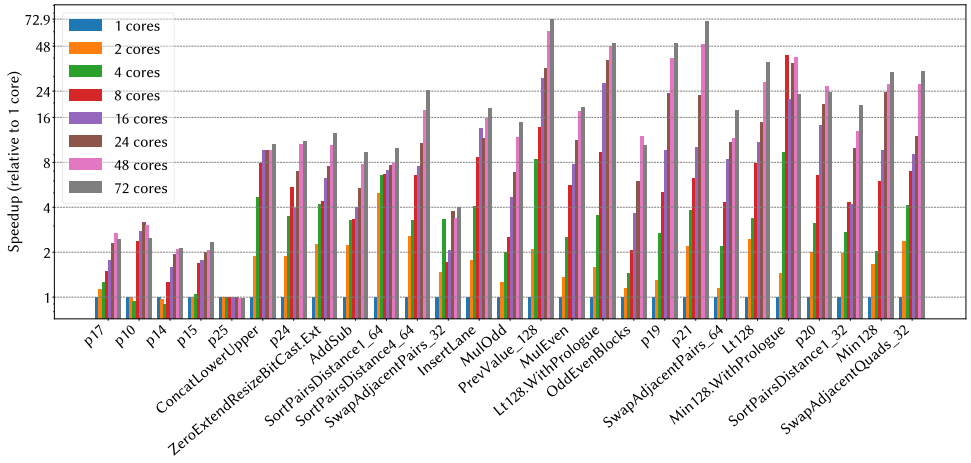


Figure 24: Speedup for synthesizing the best solution.

In Figure 24, several benchmarks achieve nearly linear scaling up to 72 cores, including `PrevValue_128` and `p21`. The system typically achieves more than 24× speedup for other complex tasks with 48-72 cores. This result is promising, showing that computationally intensive tasks scale best with parallelism. The limitations to this scaling come from that, as the number of cores increases, more cores may work on tasks that are ultimately pruned, and the work is wasted. A more advanced scheduling strategy to prioritize tasks better could mitigate this issue by reducing redundant computation. Developing such strategies remains as future work.

In Figure 25, we observe a similar trend for proving optimality as in Figure 24, although the overall speedup is typically smaller. This mostly comes from the "long tail" effect observed during the final stage of parallel synthesis. At this stage, most tasks have been pruned, leaving a small number of remaining tasks. When the number of remaining tasks is fewer than the available cores, some cores remain idle. This reduces effective parallelization and diminishes the speedup.

Overall, although scalability can be limited by the long tail effect in the optimality-proving phase and redundant work on pruned tasks, HieraSynth demonstrates effective parallel scalability, especially for complex tasks requiring extensive search space exploration.

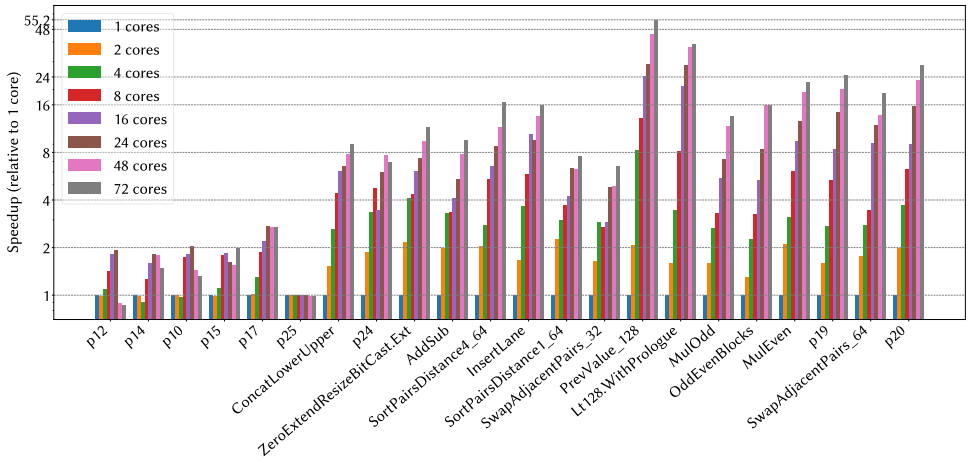


Figure 25: Speedup for proving optimality.

6.8 CHAPTER SUMMARY

This chapter presents HIERASYNTH, a novel framework designed to address the scalability challenges inherent in complete super-optimization that guarantees provable cost-model optimality. HIERASYNTH's core innovation is its *adaptive, hierarchical* strategy of decomposing the vast search space along the instruction set size (n) rather than program length (k), breaking the critical k-vs-n power-law constraint that has limited prior approaches.

The framework's key technical contributions begin with a program space representation that combines component-based synthesis with embedded instruction choices. This enables a compact encoding and a systematic, hierarchical decomposition on instruction sets. This decomposition is orchestrated by a parallel divide-and-conquer search algorithm, which leverages efficient unrealizability detection powered by the underlying GRISETTE symbolic reasoning engine to aggressively prune infeasible program subspaces. Implemented as a reusable library, HIERASYNTH inherits GRISETTE's modular design, allowing for easy adaptation to new instruction sets and synthesis tasks.

Our evaluation on the complex RISC-V Vector (RVV) ISA demonstrates HIERASYNTH's significant contributions. HIERASYNTH successfully synthesizes programs of greater length ($k \approx 7 \sim 8$) that are *provably cost-model optimal* for significantly larger and more complex instruction sets ($n \approx 700$) than previously possible. Notably, HIERASYNTH has discovered novel optimizations surpassing those crafted by human experts. This work not only introduces a new, scalable methodology for super-optimization but also serves as a compelling demonstration of GRISETTE's power and flexibility

in constructing sophisticated, high-performance automated reasoning systems for challenging domains.

7

CONCLUSION

This dissertation addressed the significant challenges in constructing robust, reusable, and efficient automated reasoning tools by proposing and developing Grisette, a symbolic compilation framework designed as a statically-typed, purely functional, monadic library. My central thesis argued that such a library-based approach facilitates the creation of specialized symbolic reasoning tools by providing a safe, modular, and extensible foundation. I detailed Grisette's core programming model, especially its symbolic union type, showed its facilities for Symbolic Domain-Specific Language (SDSL) construction, and its common patterns for symbolic evaluation.

Grisette's design is centered on key principles including efficient and safe integration of host language features via partial evaluation, configurable path merging with the Ordered Guards (ORG) representation, and transparent and predictable monadic effect management. The ORG representation, with its hierarchical structure and type-directed merging strategies, is the foundation of Grisette's symbolic union type. It is crucial for efficiently handling the complex symbolic values that arise in purely functional symbolic evaluation, leading to more compact SMT formulas and improved performance compared to traditional list-based union approaches or systems relying on a global state for effect management.

The utility and effectiveness of Grisette were demonstrated through two significant applications. First, in the development of the bounded verifier for TensorRight, an automated system for verifying rank- and size-polymorphic tensor graph rewrites, Grisette enabled earlier bug detection, more flexible error handling for complex verification queries, and enhanced modularity for generic programming with custom symbolic types. This experience highlighted the benefits of Grisette's typed, functional design over dynamically-typed alternatives for building reliable verification tools. Second, this work presents HieraSynth, a novel parallel superoptimization library, leverages Grisette for its core symbolic reasoning. HieraSynth introduces a novel, adaptive strategy of decomposing program search spaces along instruction set size (n) rather than program length (k). This approach allowed HieraSynth to significantly improve superoptimization scalability, successfully synthesizing programs that are *provably cost-model optimal* for larger and more complex programs (e.g., for RISC-

V Vector ISAs) than previously feasible. In some cases, these synthesized programs delivered optimizations with *empirical performance* beyond that of human-expert code.

In conclusion, this dissertation has presented GRISETTE as a practical and powerful instantiation of the "symbolic reasoning as a library" philosophy. By addressing fundamental challenges in symbolic value representation, effect management, and extensibility through a principled, typed, functional design, GRISETTE offers a robust foundation for building specialized automated reasoning tools. The successful development of both the TENSORRIGHT verifier and HIERASYNTH super-optimizer framework on this foundation serves as a compelling validation of this approach.

BIBLIOGRAPHY

[1] Martín Abadi et al. "TensorFlow: a system for large-scale machine learning." In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283. ISBN: 9781931971331.

[2] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. "The Essence of Dependent Object Types." In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella. Cham: Springer International Publishing, 2016, pp. 249–272. ISBN: 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_14. URL: https://doi.org/10.1007/978-3-319-30936-1_14.

[3] Jason Ansel et al. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation." In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, 929–947. ISBN: 9798400703850. DOI: 10.1145/3620665.3640366. URL: https://doi.org/10.1145/3620665.3640366.

[4] Jai Arora et al. "TensorRight: Automated Verification of Tensor Graph Rewrites." In: *Proc. ACM Program. Lang.* 9.POPL (Jan. 2025). DOI: 10.1145/3704865. URL: https://doi.org/10.1145/3704865.

[5] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing Symbolic Execution with Veritesting." In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 1083–1094. ISBN: 9781450327565. DOI: 10.1145/2568225.2568293. URL: https://doi.org/10.1145/2568225.2568293.

[6] Sorav Bansal and Alex Aiken. "Automatic generation of peephole superoptimizers." In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: Association for Computing Machinery, 2006, pp. 394–403. ISBN: 1595934510. DOI: 10.1145/1168857.1168906. URL: https://doi.org/10.1145/1168857.1168906.

[7] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. "From relational verification to SIMD loop synthesis." In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '13. Shenzhen, China: Association for Computing Machinery, 2013, 123–134. ISBN: 9781450319225. DOI: 10.1145/2442516.2442529. URL: https://doi.org/10.1145/2442516.2442529.

[8] Mark Blacher, Joachim Giesen, Peter Sanders, and Jan Wassenberg. *Vectorized and performance-portable Quicksort*. 2022. arXiv: 2205.05982 [cs.IR]. URL: https://arxiv.org/abs/2205.05982.

[9] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. "Programming with angelic nondeterminism." In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain: Association for Computing Machinery, 2010, pp. 339–352. ISBN: 9781605584799. DOI: 10.1145/1706299.1706339. URL: https://doi.org/10.1145/1706299.1706339.

[10] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. "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*. ASPLOS '16. Atlanta, Georgia, USA: Association for Computing Machinery, 2016, pp. 83–98. ISBN: 9781450340915. DOI: 10.1145/2872362.2872406. URL: https://doi.org/10.1145/2872362.2872406.

[11] James Bornholt and Emina Torlak. "Finding Code That Explodes under Symbolic Evaluation." In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018). DOI: 10.1145/3276519. URL: https://doi.org/10.1145/3276519.

[12] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

[13] Robert Brummayer and Armin Biere. "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. 2009, pp. 174–177. DOI: 10.1007/978-3-642-00768-2_16.

[14] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "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. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[15] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death." In: *ACM Trans. Inf. Syst. Secur.* 12.2 (2008). ISSN: 1094-9224. DOI: 10.1145/1455518.1455522. URL: https://doi.org/10.1145/1455518.1455522.

[16] Cristian Cadar and Koushik Sen. "Symbolic execution for software testing: three decades later." In: *Commun. ACM* 56.2 (Feb. 2013), 82–90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795. URL: https://doi.org/10.1145/2408776.2408795.

[17] Kartik Chandra and Rastislav Bodik. "Bonsai: Synthesis-Based Reasoning for Type Systems." In: *Proc. ACM Program. Lang.* 2.POPL (2017). DOI: 10.1145/3158150. URL: https://doi.org/10.1145/3158150.

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

[19] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Kurt Jensen and Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 168–176. ISBN: 978-3-540-24730-2. DOI: 10.1007/978-3-540-24730-2_15.

[20] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo Theories: Introduction and Applications." In: *Commun. ACM* 54.9 (2011), pp. 69–77. ISSN: 0001-0782. DOI: 10.1145/1995376.1995394. URL: https://doi.org/10.1145/1995376.1995394.

[21] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. "Modular Verification of Code with SAT." In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ISSTA '06. Portland, Maine, USA: Association for Computing Machinery, 2006, pp. 109–120. ISBN: 1595932631. DOI: 10.1145/1146238.1146251. URL: https://doi.org/10.1145/1146238.1146251.

[22] Julian Dolby, Mandana Vaziri, and Frank Tip. "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*. ESEC-FSE '07. Dubrovnik, Croatia: Association for Computing Machinery, 2007, pp. 195–204. ISBN: 9781595938114. DOI: 10.1145/1287624.1287653. URL: https://doi.org/10.1145/1287624.1287653.

[23] Mark Dowson. "The Ariane 5 software failure." In: *SIGSOFT Softw. Eng. Notes* 22.2 (Mar. 1997), p. 84. ISSN: 0163-5948. DOI: 10.1145/251880.251992. URL: https://doi.org/10.1145/251880.251992.

[24] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. "DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning." In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, 835–850. ISBN: 9781450383912. DOI: 10.1145/3453483.3454080. URL: https://doi.org/10.1145/3453483.3454080.

[25] Andy Gill. `mtl`. Version 2.2.2. 2021. URL: https://hackage.haskell.org/package/mtl.

[26] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. "Synthesis of loop-free programs." In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 62–73. ISBN: 9781450306638. DOI: 10.1145/1993498.1993506. URL: https://doi.org/10.1145/1993498.1993506.

[27] William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. "Lazy Counterfactual Symbolic Execution." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 411–424. ISBN: 9781450367127. DOI: 10.1145/3314221.3314618. URL: https://doi.org/10.1145/3314221.3314618.

[28] William T. Hallahan, Anton Xue, and Ruzica Piskac. "G2Q: Haskell Constraint Solving." In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: Association for Computing Machinery, 2019, pp. 44–57. ISBN: 9781450368131. DOI: 10.1145/3331545.3342590. URL: https://doi.org/10.1145/3331545.3342590.

[29] Trevor Hansen, Peter Schachte, and Harald Søndergaard. "State Joining and Splitting for the Symbolic Execution of Binaries." In: *Runtime Verification*. Ed. by Saddek Bensalem and Doron A. Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 76–92. ISBN: 978-3-642-04694-0. DOI: 10.1007/978-3-642-04694-0_6.

[30] Klaus Havelund and Thomas Pressburger. "Model checking java programs using java pathfinder." In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 366–381. DOI: 10.1007/s100090050043.

[31] *Highway API synopsis/quick reference*. 2024. URL: https://web.archive.org/web/20240615094120/https://github.com/google/highway/blob/master/g3doc/quick_reference.md (visited on 06/15/2024).

[32] Highway Developers. *Highway*. Version 1.2.0. May 31, 2024. URL: https://github.com/google/highway.

[33] Jingmei Hu, Stephen Chong, and Margo Seltzer. "Parallel Assembly Synthesis." In: *Logic-Based Program Synthesis and Transformation: 34th International Symposium, LOPSTR 2024, Milan, Italy, September 9–10, 2024, Proceedings*. Milan, Italy: Springer-Verlag, 2024, 3–26. ISBN: 978-3-031-71293-7. DOI: 10.1007/978-3-031-71294-4_1. URL: https://doi.org/10.1007/978-3-031-71294-4_1.

[34] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. "Oracle-guided component-based program synthesis." In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 215–224. ISBN: 9781605587196. DOI: 10.1145/1806799.1806833. URL: https://doi.org/10.1145/1806799.1806833.

[35] Phillip Johnston and Rozi Harris. "The Boeing 737 MAX saga: lessons for software organizations." In: *Software Quality Professional* 21.3 (2019), pp. 4–12.

[36] James Cornelius. King. "A program verifier." eng. PhD thesis. 1969.

[37] Oleg Kiselyov. *Efficient Set Monad*. archived 2020-11-12. 2013. URL: https://web.archive.org/web/20201112030922/http://okmij.org/ftp/Haskell/set-monad.html (visited on 11/12/2020).

[38] N.G. Leveson and C.S. Turner. "An investigation of the Therac-25 accidents." In: *Computer* 26.7 (1993), pp. 18–41. DOI: 10.1109/MC.1993.274940.

[39] Zhengyang Liu, Stefan Mada, and John Regehr. "Minotaur: A SIMD-Oriented Synthesizing Superoptimizer." In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689766. URL: https://doi.org/10.1145/3689766.

[40] Sirui Lu and Rastislav Bodík. "Grisette: Symbolic Compilation as a Functional Programming Library." In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). DOI: 10.1145/3571209. URL: https://doi.org/10.1145/3571209.

[41] Henry Massalin. "Superoptimizer: a look at the smallest program." In: *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS II. Palo Alto, California, USA: Association for Computing Machinery, 1987, pp. 122–126. ISBN: 0818608056. DOI: 10.1145/36206.36194. URL: https://doi.org/10.1145/36206.36194.

[42] W. M. McKeeman. "Peephole optimization." In: *Commun. ACM* 8.7 (July 1965), 443–444. ISSN: 0001-0782. DOI: 10.1145/364995.365000. URL: https://doi.org/10.1145/364995.365000.

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

[44] A Muir and J Lopatto. *Final report on the August 14, 2003 blackout in the United States and Canada : causes and recommendations*. 2004.

[45] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. "Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 225–242. ISBN: 9781450368735. DOI: 10.1145/3341301.3359641. URL: https://doi.org/10.1145/3341301.3359641.

[46] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. "Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 41–61. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/nelson.

[47] Aina Niemetz and Mathias Preiner. "Bitwuzla." In: *Computer Aided Verification*. Ed. by Constantin Enea and Akash Lal. Cham: Springer Nature Switzerland, 2023, pp. 3–17. ISBN: 978-3-031-37703-7.

[48] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. "Scaling up Superoptimization." In: *SIGARCH Comput. Archit. News* 44.2 (Mar. 2016), 297–310. ISSN: 0163-5964. DOI: 10.1145/2980024.2872387. URL: https://doi.org/10.1145/2980024.2872387.

[49] Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. "A Formal Foundation for Symbolic Evaluation with Merging." In: *Proc. ACM Program. Lang.* 6.POPL (2022). DOI: 10.1145/3498709. URL: https://doi.org/10.1145/3498709.

[50] *RISC-V vector extension for integer workloads: An informal gap analysis*. 2024. URL: https://web.archive.org/web/20241113090818/https://gist.github.com/camel-cdr/99a41367d6529f390d25e36ca3e4b626 (visited on 11/13/2024).

[51] Richard L Rudell. *Multiple-valued logic minimization for PLA synthesis*. Tech. rep. CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB, 1986.

[52] Amit Sabne. *XLA : Compiling Machine Learning for Peak Performance*. 2020.

[53] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. *Souper: A Synthesizing Superoptimizer*. 2018. arXiv: 1711.04422 [cs.PL]. URL: https://arxiv.org/abs/1711.04422.

[54] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization." In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 305–316. ISBN: 9781450318709. DOI: 10.1145/2451116.2451150. URL: https://doi.org/10.1145/2451116.2451150.

[55] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript." In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: Association for Computing Machinery, 2013, pp. 488–498. ISBN: 9781450322379. DOI: 10.1145/2491411.2491447. URL: https://doi.org/10.1145/2491411.2491447.

[56] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. "MultiSE: Multi-Path Symbolic Execution Using Value Summaries." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 842–853. ISBN: 9781450336758. DOI: 10.1145/2786805.2786830. URL: https://doi.org/10.1145/2786805.2786830.

[57] Vaibhav Sharma, Soha Hussein, Michael W. Whalen, Stephen McCamant, and Willem Visser. "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*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 123–134. ISBN: 9781450370431. DOI: 10.1145/3368089.3409734. URL: https://doi.org/10.1145/3368089.3409734.

[58] Ali Shokri. *[Question] Running CBMC on a C file with dynamic linking and/or shared libraries*. archived 2025-05-15. 2025. URL: https://web.archive.org/web/20250515124825/https://github.com/diffblue/cbmc/issues/8606 (visited on 05/15/2025).

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

[60] Armando Solar-Lezama. "Program Synthesis by Sketching." AAI3353225. PhD thesis. USA, 2008. ISBN: 9781109097450.

[61] Armando Solar-Lezama. "Program Synthesis by Sketching." AAI3353225. PhD thesis. USA: University of California, Berkeley, 2008. ISBN: 9781109097450.

[62] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. "Combinatorial Sketching for Finite Programs." In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: Association for Computing Machinery, 2006, pp. 404–415. ISBN: 1595934510. DOI: 10.1145/1168857.1168907. URL: https://doi.org/10.1145/1168857.1168907.

[63] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. "Combinatorial Sketching for Finite Programs." In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: Association for Computing Machinery, 2006, pp. 404–415. ISBN: 1595934510. DOI: 10.1145/1168857.1168907. URL: https://doi.org/10.1145/1168857.1168907.

[64] TechTarget. *CrowdStrike outage explained: What caused it and what's next*. 2024. URL: https://web.archive.org/web/20250208071027/https://www.techtarget.com/whatis/feature/Explaining-the-largest-IT-outage-in-history-and-whats-next (visited on 10/29/2024).

[65] Johan Tibell. *unordered-containers*. Version 0.2.20. 2024. URL: https://hackage.haskell.org/package/unordered-containers.

[66] Mads Tofte. "Type inference for polymorphic references." In: *Information and Computation* 89.1 (1990), pp. 1–34. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(90)90018-D. URL: https://www.sciencedirect.com/science/article/pii/089054019090018D.

[67] Emina Torlak. *unsound behavior?* archived 2020-11-22. 2016. URL: https://web.archive.org/web/20201122222820/https://github.com/emina/rosette/issues/36 (visited on 11/22/2020).

[68] Emina Torlak. *Use early return in rosette?* archived 2022-02-22. 2021. URL: https://web.archive.org/web/20220223043310/https://github.com/emina/rosette/issues/201 (visited on 02/22/2022).

[69] Emina Torlak. *Performance, Rosette Guide*. archived 2024-10-01. 2024. URL: https://web.archive.org/web/20241001212556/https://docs.racket-lang.org/rosette-guide/ch_performance.html (visited on 10/01/2024).

[70] Emina Torlak. *Debugging, Rosette Guide*. archived 2025-01-30. 2025. URL: https://web.archive.org/web/20250130202806/https://docs.racket-lang.org/rosette-guide/ch_error-tracing.html (visited on 01/30/2025).

[71] Emina Torlak and Rastislav Bodik. "Growing Solver-Aided Languages with Rosette." In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pp. 135–152. ISBN: 9781450324724. DOI: 10.1145/2509578.2509586. URL: https://doi.org/10.1145/2509578.2509586.

[72] Emina Torlak and Rastislav Bodik. "A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages." In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 530–541. ISBN: 9781450327848. DOI: 10.1145/2594291.2594340. URL: https://doi.org/10.1145/2594291.2594340.

[73] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. "Vectorization for digital signal processors via equality saturation." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing

Machinery, 2021, pp. 874–886. ISBN: 9781450383172. DOI: 10.1145/3445814.3446707. URL: https://doi.org/10.1145/3445814.3446707.

[74] Henry S Warren. *Hacker's delight*. Pearson Education, 2013.

[75] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. UCB/EECS-2014-54. 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

[76] Guannan Wei, Oliver Bračevac, Shangyin Tan, and Tiark Rompf. "Compiling Symbolic Execution with Staging and Algebraic Effects." In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020). DOI: 10.1145/3428232. URL: https://doi.org/10.1145/3428232.

[77] Max Willsey et al. "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*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 183–197. ISBN: 9781450362405. DOI: 10.1145/3297858.3304027. URL: https://doi.org/10.1145/3297858.3304027.

[78] A.K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness." In: *Information and Computation* 115.1 (1994), pp. 38–94. ISSN: 0890-5401. DOI: https://doi.org/10.1006/inco.1994.1093. URL: https://www.sciencedirect.com/science/article/pii/S0890540184710935.

[79] Yichen Xie and Alex Aiken. "Scalable Error Detection Using Boolean Satisfiability." In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, California, USA: Association for Computing Machinery, 2005, pp. 351–363. ISBN: 158113830X. DOI: 10.1145/1040305.1040334. URL: https://doi.org/10.1145/1040305.1040334.