OPTIMIZING THE AUTOMATED PROGRAMMING STACK

JAMES BORNHOLT

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

Doctor of Philosophy

University of Washington

2019

Reading Committee:
Emina Torlak, Chair
Dan Grossman, Chair
Luis Ceze, Chair

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

ABSTRACT

OPTIMIZING THE AUTOMATED PROGRAMMING STACK

James Bornholt

Chairs of the Supervisory Committee:
Associate Professor Emina Torlak
Professor Dan Grossman
Professor Luis Ceze
Paul G. Allen School of Computer Science & Engineering

The scale and pervasiveness of modern software poses a challenge for programmers: software reliability is more important than ever, but the complexity of computer systems continues to grow. Automated programming tools are a powerful way for programmers to tackle this challenge: verifiers that check software correctness, and synthesizers that generate new correct-by-construction programs. These tools are most effective when they apply domain-specific optimizations, but doing so today requires considerable formal methods expertise.

This dissertation shows that new abstractions and techniques can empower programmers to build specialized automated programming tools that ensure software reliability. We first demonstrate the importance and effectiveness of automated tools in the context of memory consistency models, which define the behavior of multiprocessor CPUs and whose subtleties often elude even experts. *MemSynth* is a tool that automatically synthesizes formal descriptions of memory consistency models from examples of CPU behavior, and has found ambiguities and underspecifications in two major computer architectures. We then introduce two new programmer techniques for developing automated programming tools. *Metasketches* are a new abstraction for building program synthesis tools that integrate search strategy into the problem definition, allowing a metasketch solver to solve synthesis problems that other tools cannot. *Symbolic profiling* is a technique for systematically identifying and resolving scalability bottlenecks in automated programming tools. Symbolic profiling generalizes across different symbolic evaluation engines and has been used to improve the performance of state-of-the-art automated tools by orders of magnitude. Together, these three contributions demonstrate the value of automated programming tools for building reliable software, and offer guidance on how to build such tools efficiently for new problem domains.

To AJ (again!)

ACKNOWLEDGMENTS

I've had the incredible privilege of being advised by Emina Torlak, who I somehow fooled into working with me during our first quarter at UW. Beyond her fearsome technical skill, she has been a source of impeccable research taste and a fierce advocate for my work. Above all else, she has been endlessly enthusiastic and supportive of my research and of me (and my writing skills, which I hope this dissertation shows she has considerably improved). I could not ask for a better mentor or friend.

I originally came to UW to work with my coadvisors, Dan Grossman and Luis Ceze, and have not for one second regretted that choice. Dan is a font of wisdom and humor, has steered me through sticky situations (usually of my own making), and is, to my mind, the quintessential PL researcher. Luis is a fearless academic and visionary, and a reliable source of (good) distracting ideas and food advice. I am grateful for their encouragement, even as my interests veered in and out of their natural research areas.

As if three advisors were not enough, I've been lucky to collaborate with Xi Wang for much of my PhD. What started as a hastily written ASPLOS submission my first summer turned into an immensely fun line of work at the intersection of systems and formal methods. Xi is deeply creative and a crazy hacker, and working with him has helped me focus on having real impact.

I am grateful to the other members of my committee: Ras Bodik, whose unique insights on my research have always improved it, and Andy Ko, who encouraged me to consider the human side of my work.

The Allen School is an amazing environment for collaboration, and I want to thank everyone I've worked with during my time here: Andrew Baumann, Doug Carmean, Bruno Castro-Karney, Ronghui Gu, Dylan Johnson, Antoine Kaufmann, Arvind Krishnamurthy, Rustan Leino, Jialin Li, Randolph Lopez, Mark Oskin, Dan Ports, Georg Seelig, Karin Strauss, Irene Zhang, and Kaiyuan Zhang.

The Sampa, PLSE, and UNSAT groups made my time in grad school a blast. There are too many of these wonderful people to list here, but thanks especially to my friends Armin Alaghi, Meghan Cowan, Brandon Holt, Vincent Lee, Amrita Mazumdar, Thierry Moreau, Luke Nelson, Sorawee Porncharoenwase, Adrian Sampson, Helgi Sigurbjarnarson, John Toman, and Jacob Van Geffen. My officemates Eunice Jun, Kit Kuksenok, Brandon Myers, Kyle Rector, and Kendall Stewart made coming to school worth it every day.

Elise Dorough, Melody Kadenko, and Andrei Stabrovski have each made countless administrative problems disappear. I'm especially thankful for Elise, who effortlessly moves between administrator and counselor as needed, and without whom the Allen School would surely implode.

Steve Blackburn, Kathryn McKinley, and Todd Mytkowicz are collectively responsible for convincing me to give grad school a try. Thanks—you were right!

Finally, I am forever grateful for the love of my family: Karen, Mark, and Michael. Moving halfway across the world for a PhD is a truly insane thing to do. Thank you for believing in me.

CONTENTS

1

INTRODUCTION

Software reliability is more critical than ever. Today's computer systems control essential infrastructure, medical devices, and our personal lives. Accompanying this proliferation, today's computer systems are also more complex than ever, combining heterogeneous hardware, distributed networks, and sophisticated algorithms. How can we help programmers to ensure reliability for these far-reaching, complex systems?

One particularly effective approach to software reliability is *automated programming tools*. Automated *verification* tools check a program's adherence to a desired specification, often written as a logical predicate defining the program's allowed behaviors. Automated *synthesis* tools take this task a step further, generating correct-by-construction programs from such logical specifications. Together, these tools promise programmers a new paradigm for building reliable software, in which automation carries the burden of ensuring reliability.

Two key challenges make realizing the promise of automated programming tools particularly difficult. The first challenge is *intractability*. Automated programming tools must solve intractable problems when reasoning about programs; for example, a tool might verify that a property holds for *every* possible input. Building an automated programming tool thus requires careful design to navigate this intractability, working with a suite of heuristic-driven techniques that are effective but sensitive to small changes.

The second challenge is *specification*. Automated programming tools relate software behavior to a specification, but for many non-trivial programs, constructing such a specification is at least as difficult as developing the program itself. Applying automated programming thus demands an ecosystem of tools for constructing specifications that reflect programmer intent and capture potential program flaws.

The common solution to both these challenges is *domain specialization* of automated programming tools. Specializing such tools to a particular problem domain addresses the intractability challenge by reducing the size of the problem space—rather than reasoning about all possible programs (or program behaviors), a specialized tool need reason only about those programs (behaviors) relevant to the problem domain. Specialization addresses the specification challenge by providing domain-specific language constructs to concisely and precisely capture programmer intent without resorting to expressive-but-complex general-purpose logics.

However, domain specialization is a challenging programming task in its own right. My thesis is that *new abstractions and techniques can empower programmers to build specialized automated programming tools that ensure software reliability*. This dissertation explores both new specialized automated tools as well as new abstractions and techniques for constructing them.

1.1 APPLYING AUTOMATED PROGRAMMING TO MEMORY CONSISTENCY

What does it take to construct an automated programming tool for a new application domain? Traditionally, such a task was a substantial engineering effort that required formal methods expertise. The complexity stems from needing to distill domain-specific concepts down to the generic interfaces of off-the-shelf formal methods components such as satisfiability solvers. More recently, however, this burden has been alleviated by new programming languages [84, 122, 126, 127], which layer a familiar programming environment over such low-level components. Automated tools built atop these languages are radically easier to construct and to scale.

To illustrate the effectiveness of this approach to constructing automated programming tools, Chapter 3 presents MemSynth [26], a tool for automatically synthesizing formal specifications of memory consistency models. These models define the ordering behavior of memory operations on a multiprocessor, and are critical to writing concurrent software correctly. However, they are usually underspecified, using natural language and example programs ("litmus tests") to define their behaviors. Research efforts to fully specify and formalize even well-studied models (such as total store ordering) have taken multiple person-years [114] and yielded incorrect results that require later clarification [119].

MemSynth takes as input example behaviors of a multiprocessor, and *automatically* synthesizes a formal descriptions of a memory model consistent with those examples. The synthesis process relies on Ocelot [25], a new embedding of relational logic into a solver-aided programming language. The solver-aided language takes care of the reduction from relational logic to boolean satisfiability. This abstraction allows us to easily implement domain-specific optimizations in MemSynth, improving its scalability by orders of magnitude. Building on this scalability allows MemSynth to include a powerful new *disambiguation* query that can detect ambiguities and underspecifications in its synthesized outputs. We have used MemSynth to synthesize memory model specifications for two major architectures, and in both cases we found ambiguities in existing documentation of the expected models.

1.2 SCALING PROGRAM SYNTHESIS WITH METASKETCHES

One of the key scalability challenges facing a synthesis tool like MemSynth is a large, irregular search space of candidate programs. A synthesis tool works by traversing this search space to find a program that satisfies the specification, and this search is most effective when it can rapidly prune away large parts of the space that do not contain valid solutions. Synthesis tools based on constraint solvers achieve this pruning effect by *learning* from invalid candidate solutions to rule out entire classes of candidates [58, 93]. The effectiveness of such a synthesis tool thus depends on its ability to execute a good search *strategy* that rapidly rules out large parts of the search space to hone in on a valid solution.

Most synthesis tools do not explicitly accept a search strategy as input; instead, they rely on the underlying solver to infer a good strategy. Scaling a synthesis

tool thus requires careful engineering to guide the solver towards good strategies for a particular problem. To give programmers more control over this critical aspect of performance, Chapter 4 introduces *metasketches*, a general abstraction for specifying synthesis problems together with a search strategy for solving them [27]. The metasketch abstraction enables programmers to implement domain-specific search strategies without having to build their own synthesis engine. Metasketches also support *optimal* synthesis problems [42, 85], in which the task is to find not just any solution but an optimal one according to a specified cost function. A metasketch for optimal synthesis uses the cost function as part of the search strategy, guiding the search process towards cheaper solutions.

Metasketches have formed the foundation of several state-of-the-art program synthesis tools, and have enabled them to solve synthesis problems that existing synthesis tools cannot. We have developed a solving engine for metasketches that can solve them efficiently and in parallel. Our results show that metasketches can express a wide variety of search strategies and cost functions, including dynamic examples such as worst-case execution time and simple machine learning models.

1.3 TARGETING SCALABILITY ISSUES WITH SYMBOLIC PROFILING

How should a programmer decide where to apply domain-specific scalability optimizations such as metasketches? In traditional programs, *performance engineering* begins by using profiling tools that measure a program's performance and rank parts of the program according to their cost (usually their run time or resource usage) [65]. The underlying hypothesis of these tools is that, by improving the performance of the slowest parts of the program, the entire program's performance will improve in concert [51]. But automated programming tools often defy this hypothesis: speeding up the slowest part of the program can have unpredictable effects on its performance. This instability stems from the unusual *all-paths* program execution model that forms the foundation of automated programming tools. These tools evaluate every possible path through the program; for example, a program verifier checks that a desired property holds on every possible execution.

The key technical challenge for improving this situation is identifying a performance model for automated programming tools that can guide programmers' optimization efforts. Chapter 5 presents symbolic profiling [24], an approach to identifying and diagnosing scalability issues in automated programming tools [24]. Symbolic profiling instruments the symbolic evaluation engines that programmers use to implement automated programming tools, and provides diagnostic guidance to identify parts of a tool where more programming effort is required for scalability. Underlying symbolic profiling is a new performance model that summarizes the behavior of any forward-based symbolic evaluation technique, including symbolic execution [47, 77], bounded model checking [17, 46], and hybrids of the two approaches [118, 126]. Layered on top of this performance model is a ranking heuristic for identifying potential optimization opportunities even in parts of the program that are not "slow" according to a traditional profiler.

After identifying a potential optimization opportunity using symbolic profiling, a programmer must still determine the appropriate change to make to the program to improve its performance. This part of the process requires manual programmer insight; often, the appropriate optimization involves exploiting domain-specific properties. However, to help programmers diagnose issues and explore potential fixes, we have developed a catalog of common *anti-patterns* in automated programming tools. Each anti-pattern lends itself to a different repair, and our catalog includes examples of both problematic and repaired code.

We and others have used symbolic profiling to improve the scalability of automated programming tools by orders of magnitude, making it possible for them to solve larger problems out of reach for existing tools. We have performed several case studies showing that symbolic profiling is effective on real-world tools, that our catalog of anti-patterns covers common issues, and that our performance model for symbolic evaluation is more effective than traditional profilers. Symbolic profiling is not specific to any one language or execution model—any forward symbolic evaluation engine can be analyzed with the same performance model. As evidence of this generality, we have integrated symbolic profiling into two different engines [117, 126], and Galois, Inc. has integrated it into their Crucible engine [61].

1.4 CONTRIBUTIONS AND OUTLINE

The remainder of this dissertation is arranged in four chapters. Chapter 2 presents the state of the art in building automated programming tools through a primer on the Rosette solver-aided language [126, 127]. Rosette abstracts away many of the challenges of building these tools by automatically lifting programs to operate over *symbolic* values. This abstraction allows programmers to build verification and synthesis tools for *domain-specific languages* (DSLs) with minimal effort: they need only define the concrete syntax and semantics of the DSL, and Rosette provides symbolic reasoning for free. However, scaling such tools to real-world problems still requires substantial programmer effort.

Chapter 3 presents MemSynth, an automated programming tool for memory consistency models, to demonstrate the programmer effort required to build scalable tools today. MemSynth demonstrates the importance of specialization—a number of critical domain-specific designs and optimizations help make MemSynth scale to industrial memory models.

Chapter 4 presents metasketches, a general abstraction for specifying and solving synthesis problems. The metasketch abstraction enables programmers to implement domain-specific synthesis search strategies without having to build their own synthesis engine, easing the burden of building a specialized tool. MemSynth and other applications have used metasketches to solve problems that off-the-shelf tools cannot.

Finally, Chapter 5 presents symbolic profiling, an approach to identifying and diagnosing scalability issues in automated programming tools. Symbolic profiling provides diagnostic guidance to identify parts of an automated programming tool

where more programming effort (e.g., a custom metasketch) is required to make a problem tractable.

Together, these chapters support the thesis at the core of this dissertation—that automated programming tools are effective at helping experts solve problems that would otherwise take years of manual work, and that new abstractions and techniques make such tools easier to build and scale in practice.

2 PRELUDE: BUILDING A SYNTHESIS TOOL

This dissertation focuses on the task of building automated programming tools—verifiers for checking software correctness, and synthesizers for automatically generating programs from specifications. We argue that new abstractions and techniques can help programmers build these tools more effectively, and demonstrate this thesis through examples of real-world large-scale tools. But before introducing these new contributions, we should first understand the current state of the art: how do we build a new synthesis tool today?

2.1 GETTING STARTED WITH ROSETTE

There are several effective off-the-shelf frameworks for program synthesis. One of the earliest is Sketch [121, 122], which offers a C-like language equipped with synthesis features. In an effort to standardize across several different synthesis engines, the syntax-guided synthesis (SyGuS) language [11] is a domain-specific language for specifying synthesis problems.

This dissertation focuses primarily on the Rosette solver-aided language [125–127], which extends the Racket programming language [60, 110] with support for verification and synthesis. This support includes *lifted* versions of many of Racket's core language features, meaning that a tool based on Rosette has access to a rich library of components (lists, vectors, pattern matching, etc.) that programmers expect from a modern language. Though our contributions are not specific to Rosette, its extensibility provides a vehicle to explore new abstractions and techniques without rebuilding the infrastructure underpinning them.

Rosette's key feature is programming with, and solving, *constraints*. Rather than a program in which all variables have known values, a Rosette program has some *unknown* values called *symbolic constants*. The values of the symbolic constants will be determined automatically at run time according to the constraints we generate.

For example, we can find an integer whose absolute value is 5:

```
1  ; Compute the absolute value of `x`
2  (define (absv x)
3    (if (< x 0) (- x) x))
4
5  ; Define a symbolic constant called y of type integer
6  (define-symbolic y integer?)
7
8  ; Solve a constraint saying |y| = 5
9  (solve
10   (assert (= (absv y) 5)))
```

This program outputs:

```
(model
```

```
[y -5])
```

This output is a *model*—an assignment of values to the symbolic constants—in which `y` takes the value -5. The model satisfies all the constraints we provided using Rosette's **`assert`** form, which in this case was just the single assertion that `(absv y)` be equal to 5.

Of course, not all constraints are satisfiable, and so models do not always exist. For example, evaluting this program:

```
(solve (assert (< (absv y) 0)))
```

outputs the value:

```
(unsat)
```

which represents an *unsatisfiable* result. In this case, the result tells us there is no possible value of `y` for which `(absv y)` is negative.

The **`solve`** form searches for an assignment of values to symbolic constants consistent with the constraints it is passed (via **`assert`**). To execute this search, Rosette compiles the program-level constraints down to logical constraints that an off-the-shelf satisfiability modulo theories (SMT) solver [53] can check. This compilation process is known as *symbolic evaluation*, and is often a bottleneck in automated programming tools, as Chapter 5 explores. When the solver finds a solution, Rosette translates the resulting model back up to the program-level symbolic values.

2.2 DOMAIN-SPECIFIC LANGUAGES

Program synthesis is the task of finding a program that satisfies a specification. The concept, then, is similar to the simple constraint solving examples above: there are some unknowns whose values we wish to fill in subject to some constraints. The difference is that, in program synthesis, the unknowns are *programs*—we wish to find concrete programs that satisfy the desired constraints.

To define the space of candidate programs for a synthesis tool to consider, we can define a *domain-specific language* (DSL). A DSL is a programming language equipped with exactly the features required for a particular domain. Parts of this dissertation deal with building DSLs for memory consistency models (Chapter 3) and for approximate computing (Chapter 4), but we have also developed DSLs for other problems such as file system crash safety [23]. Other researchers and practitioners have developed DSLs for a myriad of tasks, from network configuration [135] to K–12 algebra tutoring [34].

The design of the DSL is a key factor in defining the tractability of synthesis. If a DSL is too complex, it may be difficult to solve the resulting synthesis problem, because there are many programs to consider. But if a DSL is too simple, it will not be able to express interesting behaviors. Navigating this trade-off is critical to building practical synthesis tools.

2.2.1 *Example: A Simple Arithmetic DSL*

Consider the problem of synthesizing simple arithmetic expressions; for example, a compiler might want to discover new expressions that are equivalent to a given input expression for optimization purposes. Defining a DSL for this problem requires us to specify two components: the DSL's *syntax* and its *semantics*.

Syntax. The syntax for our DSL will use abstract data types to represent the different expressions in the language. This representation is a *deep embedding* of our arithmetic DSL inside Rosette. Our abstract data types will be implemented using *structures* (Racket's implementation of records). We declare the three operators in our DSL as follows:

```
(struct plus (left right) #:transparent)
(struct mul (left right) #:transparent)
(struct square (arg) #:transparent)
```

Here, each **`struct`** line declares a new data type named for its first argument (`plus`, `mul`, `square`), with a field for each name in the second argument list (so `plus` and `mul` have two arguments while `square` has one). The `#:transparent` annotations are boilerplate for generating string representations and equality definitions for the new data types.[1]

Our syntax allows us to write programs in the DSL by instantiating the relevant structures; for example, we can write:

```
(define prog (plus (square 7) 3))
```

to assign to `prog` a program that represents the mathematical expression $7^2 + 3$.

Semantics. To define our DSL's semantics, we implement an *interpreter* for programs in it. The interpreter takes as input a program, performs the computations that program describes, and returns an output value. For example, when passed the `prog` above as input, the interpreter should return the value 52—the result of evaluating the mathematical expression $7^2 + 3$.

Since our DSL syntax represents programs as abstract syntax trees, our interpreter recurses on the structure of the tree using pattern matching:

```
(define (interpret p)
  (match p
    [(plus a b)  (+ (interpret a) (interpret b))]
    [(mul a b)   (* (interpret a) (interpret b))]
    [(square a)  (expt (interpret a) 2)]
    [_ p]))
```

The recursive cases for each of the three DSL operations first evaluate their arguments `a` and `b`, and then apply the corresponding Racket arithmetic operation

1 `#:transparent` also has a Rosette-specific meaning: during symbolic evaluation, structures with this annotation can have their values merged using bounded model checking, rather than requiring symbolic execution to split into separate paths for each possible value of the structure. Chapter 5 explores the performance implications of this distinction.

to the resulting values. The recursion has a base case `[_ p]` (in Racket patterns, `_` matches any value) that returns the input program itself. This base case handles constant values (e.g., `7` and `5` in `prog` above) by returning the value itself.

2.3 SYNTHESIS WITH DSLS

The interpreter above is pure Racket code—it operates on concrete values (instances of our DSL data types), and returns concrete values. Rosette *lifts* pure Racket code to operate over symbolic values, and so the concrete interpreter works symbolically without any changes. For example, we can evaluate a program that includes a *symbolic* integer `y`:

```
(define-symbolic y integer?)
(interpret (square (plus y 2)))
```

which returns the value

```
(* (+ 2 y) (+ 2 y))
```

since `y` is symbolic. Note that this returned expression is still a value—the instances of `square` and `plus` from our DSL have been erased—but it is a symbolic value in Rosette's term language.

Building on this ability to work symbolically, we can use Rosette's **`solve`** form again, this time to answer simple questions about our DSL. For example, we can ask whether there exists a value of `y` that makes the program `(square (plus y 2))` evaluate to 25 (in other words, whether there exists a y such that $(y + 2)^2 = 25$) by invoking the solver with such a constraint:

```
(solve
  (assert
    (= (interpret (square (plus y 2))) 25)))
```

This program returns a model:

```
(model
 [y -7])
```

Here, the model says that by setting `y` to -7, the program `(square (plus y 2))` evaluates 25. Of course, setting `y` to 3 would also be a valid model; when multiple satisfying models exist, Rosette (by virtue of its underlying SMT solver) chooses one such model arbitrarily.

This example already demonstrates simple program synthesis: we have asked our tool to find us a program (of a restricted shape) that evaluates to 25.[2] To do so, our tool effectively evaluates *all possible programs* (of the restricted shape), and then searches for one that satisfies the specification using an SMT solver.

2 This problem is known as *angelic execution* [41, 78, 98] and has a number of applications beyond synthesis.

2.3.1 *Dealing with Program Inputs*

In the synthesis example above, the synthesized program is a constant expression. But most programs are not constant: they take *inputs* and compute with them to produce outputs. For example, rather than finding a constant expression that evaluates to a particular number, we might want to find a constant `c` such that, *for every possible* `x`, the program `(mul c x)` evaluates to $x + x$. More formally, we want to solve the query:

$$\exists c.\ \forall x.\ cx = x + x$$

While we can express this problem using symbolic values, the synthesis approach above will not give the correct results, because it does not account for the universal quantifier over `x`. Concretely, trying the following program:

```
(define-symbolic x c integer?)
(solve
  (assert
    (= (interpret (mul c x)) (+ x x))))
```

gives the output:

```
(model
 [x 0]
 [c 0])
```

This model assigns concrete values to *both* `c` and `x`; when both are set to zero, it is true that $cx = x + x$, but this says nothing about other values of `x`.

To find a value of `c` that works for every `x`, Rosette includes a **`synthesize`** form. This form takes as input a constraint (just as **`solve`** does) together with a list of quantified variables, and finds a model that satisfies the constraint for every value of those quantified variables. To solve the $cx = x + x$ problem above, we specify `x` as a quantified variable while using the same constraint as above:

```
(synthesize
  #:forall (list x)
  #:guarantee (assert (= (interpret (mul c x)) (+ x x))))
```

This program finds a model that binds only the unquantified constant `c`:

```
(model
 [c 2])
```

This model says that $2x = x + x$ for every value of x, as we would expect.

This example already demonstrates the potential of automated synthesis tools: here, we have discovered a fact about our DSL (that `(mul 2 x)` and `(plus x x)` are equivalent) without having to write any facts explicitly. The synthesizer discovered this fact based solely on the semantics of the DSL we defined. This same idea has been extended to find novel program rewrites and optimizations in a number of application domains [70, 86].

2.4 BUILDING SKETCHES

Our example above synthesized a constant `c`, but we had to specify the shape of the program around that constant. A realistic program synthesis tool should be able to generate entire programs, rather than just constants. But as discussed above, generating entire programs is a second-order problem. To reduce this difficult query to a first-order one our tools can solve efficiently, our program synthesis tool will use *sketches* of programs from our DSL.

A sketch is a syntactic template for a program, with missing expressions called *holes* for the synthesizer to fill in. Each hole is associated with a set of possible completions, which are expressions that the synthesizer can try to place in the hole. The sketch defines the search space for the synthesizer, which will explore only those programs that correspond to a completion of the sketch. The synthesis example above used a very simple sketch—it contained a single hole `c` whose possible completions were integer constants—but we can also write sketches with multiple holes, and define more complex holes whose completions are program fragments in our DSL.

To define these more complex holes, we write a function that nondeterministically evaluates to one of several possible expressions:

```
1 (define (??expr terminals)
2   (define a (apply choose* terminals))
3   (define b (apply choose* terminals))
4   (choose* (plus a b)
5            (mul a b)
6            (square a)
7            a))
```

This `??expr` function takes as input a list of *terminal* elements, defining the expressions that can appear as leaves of our synthesized program fragment. The `choose*` form is provided by Rosette and expresses nondeterministic choice: `(choose* x y z ...)` can evaluate to any of its arguments. We use `choose*` in two ways. First, we define two leaf expressions `a` and `b` that can evaluate to any of the possible terminal elements. Second, we define the return value of `??expr` as a choice between `plus`, `mul`, or `square` expressions in our DSL, or one of the terminal elements directly. Our `??expr` function therefore returns expressions that apply any of our DSL operators to any of the given terminal elements, or return one of the terminals directly. For example, `(??expr 2 x)` can evaluate to programs such as `2`, `x`, `(plus 2 x)`, `(plus x x)`, or `(mul 2 x)`, but not to multiple nestings of the DSL operators, such as `(plus (plus 2 x) 2)`.

The `??expr` procedure demonstrates how Rosette reduces second-order synthesis queries quantified over programs to first-order queries quantified over values. The key is the `choose*` operator. Given a call to $(\texttt{choose*}\ x_0\ \ldots\ x_n)$, Rosette implicitly declares a set of symbolic boolean constants $b_0, \ldots, b_{n-1}$, and returns an if-then-else expression of the form $(\texttt{if}\ b_0\ x_0\ (\texttt{if}\ b_1\ x_1\ (\texttt{if}\ \ldots\ (\texttt{if}\ b_{n-1}\ x_{n-1}\ x_n)))$ that guards each possible program fragment with a corresponding symbolic boolean constant. This return value reduces the choice between the given program frag-

ments to a first-order formula over the boolean constants—each assignment to the booleans corresponds to a program.

Given the ability to construct holes, we can now write a sketch:

```
(define-symbolic x p q integer?)
(define sketch
  (plus (??expr (list x p q)) (??expr (list x p q))))
```

This sketch defines a search space of programs that have a sum operation at the top level, but whose operands are unknown expressions built out of the symbolic constants `x`, `p`, and `q`. Because `??expr` only returns expressions with at most one operator, the resulting program will have no more than three DSL operations (the top-level `plus` and one in each leaf). However, we could write an alternative `??expr` that allowed nested operations (up to some finite depth).

Using this sketch, we can pose more complex synthesis problems. For example, we can ask the synthesizer whether it is possible to rewrite an operation such as `(mul 10 x)` as the sum of two expressions, based on the sketch above:

```
(define M
  (synthesize
    #:forall (list x)
    #:guarantee (assert (= (interpret sketch)
                           (interpret (mul 10 x))))))

(evaluate sketch M)
```

Since the returned model will bind the many implicit booleans created by `choose*`, we save it into a variable `M` rather than viewing it directly. Rosette's **`evaluate`** form allows us to convert this model into readable output. It takes as input an expression (`sketch`) and a model (`M`), and returns `sketch` but with any symbolic constants in `sketch` substituted with their bindings in the model `M`. The result of evaluating `sketch` against `M` is therefore a completion of `sketch` that satisfies the constraint we gave to **`synthesize`**:

```
(plus (mul 8 x) (plus x x))
```

This program tells us that $10x = 8x + (x + x)$, which is one way of decomposing $10x$ into a sum of two expressions. There are many other possible decompositions, and so Rosette is free to choose any of them arbitrarily.

2.5 BENEFITS AND PITFALLS

Our example works with trivial specifications over a simple DSL. The key advantage Rosette gave us was abstraction: we were able to implement our DSL (syntax and semantics) without thinking at all about synthesis or verification. The interpreter for concrete programs in our DSL worked automatically with symbolic values and even symbolic programs, and Rosette gave us access to complex queries for synthesizing programs in our DSL without any changes to the interpreter.

The challenge this dissertation focuses on is the next step: how do we turn our simple synthesizer into one that can solve real-world problems? As Chapter 1 discusses, the two key impediments to this progress are intractability and specification. Real-world problems have more complex DSLs with richer semantics, which can make them intractable for a synthesizer to reason about (intuitively, they lead to a bigger search space of programs). Real-world problems also carry more complex specifications to define richer behaviors, which are both difficult for programmers to write and difficult for synthesizers to reason about.

We argue in this dissertation that *domain specialization* is a potent technique for addressing these two challenges. On the surface, however, specialization is in opposition to our successes in the simple examples above: Rosette's abstractions (and those of other solver-aided languages) reduce the workload of programmers building automated tools, but specialization requires more work to identify and apply appropriate optimizations. How can we allow programmers to build specialized automated programming tools while still retaining the productivity benefits of solver-aided languages?

The remainder of this dissertation addresses this challenge in two parts. First, Chapter 3 demonstrates the effectiveness of specialized automated programming tools in the case of memory consistency models. With the appropriate specializations, the tools built in that domain can solve important problems that no off-the-shelf tool can. Second, Chapters 4 and 5 present techniques that allow such tools to be built efficiently. These techniques help focus programmer effort on discovering and implementing specializations. Together, these two contributions demonstrate that new abstractions and techniques can empower programmers to build specialized automated programming tools.

3

SYNTHESIS OF MEMORY CONSISTENCY MODEL SPECIFICATIONS

A memory consistency model specifies which writes to shared memory a given read may see. Ambiguities or errors in these specifications can lead to bugs in both compilers and applications. Yet architectures usually define their memory models with prose and *litmus tests*—small concurrent programs that demonstrate allowed and forbidden outcomes. Recent work has formalized the memory models of common architectures through substantial manual effort, but as new architectures emerge, there is a growing need for tools to aid these efforts.

This chapter presents MemSynth, a synthesis-aided system for reasoning about axiomatic specifications of memory models.[1] MemSynth takes as input a set of litmus tests and a *framework sketch* that defines a class of memory models. The sketch comprises a set of axioms with missing expressions (or *holes*). Given these inputs, MemSynth synthesizes a completion of the axioms—i.e., a memory model—that gives the desired outcome on all tests. The MemSynth engine employs a novel embedding of bounded relational logic in a solver-aided programming language, which enables it to tackle complex synthesis queries intractable to existing relational solvers. This design also enables it to solve new kinds of queries, such as checking whether a set of litmus tests uniquely identifies a memory model, and if not, synthesizing a new litmus test summarizing the ambiguity. MemSynth can synthesize specifications for x86 in under two seconds, and for PowerPC in 12 seconds from 768 litmus tests. Our ambiguity check identifies missing tests from both the Intel x86 documentation and the validation suite of a previous PowerPC formalization.

3.1 OVERVIEW

Reasoning about concurrent code on a multiprocessor requires a *memory consistency model* that specifies the memory reordering behaviors the hardware will expose. Architectures typically define their memory consistency model with prose and *litmus tests*, small programs that illustrate allowed and forbidden outcomes. These imprecise definitions make reasoning about correctness difficult for developers and tool builders. Researchers have argued for formalizing memory models [140], and have recently created formal models for common architectures, including x86 [119] and PowerPC [91]. But each such formalization required several person-years of effort and several revisions (e.g., [5, 7, 103, 113, 114]).

These formalization efforts have been aided by tools for *verification* and *comparison* of memory models. Verification tools check whether a model allows a lit-

1 This chapter was first published as the paper *Synthesizing Memory Models from Framework Sketches and Litmus Tests*, by James Bornholt and Emina Torlak, at PLDI 2017 [26]. This version adds proofs of the major theorems about MemSynth's reasoning engine.

mus test [7, 105, 129], while comparison tools synthesize litmus tests on which two models disagree [89, 136]. These tools provide verification and comparison queries for memory models within a given *axiomatic framework* (e.g., [9]). The framework supplies basic axioms that every memory model must follow, expressed as first-order constraints on relations that order memory events (such as reads and writes). The tools then answer queries about specific models from the framework with respect to a given litmus test (in the case of verification) or a space of litmus tests (in the case of comparison). But no existing tools can answer queries about the framework itself, e.g., whether it contains a memory model that satisfies a set of litmus tests.

This chapter proposes using *program synthesis* to answer novel queries about memory models and their frameworks. The core idea behind the proposal is a *framework sketch*, which describes a class of memory models with a syntactic template. The template consists of a set of axioms with *holes* [122] (i.e., missing expressions) whose completion defines a memory model from the target class. The sketch is provided by the memory model designer and can capture domain-specific insights and assumptions, such as the use of scopes [4] to describe GPU memory models. Given a framework sketch, synthesis-based tools can answer a variety of new queries about memory models. For example, they can search for a memory model specification that satisfies a set of example litmus tests, automating a tedious development cycle currently performed by hand [95]. Synthesis also enables more complex queries, such as determining whether a synthesized model is ambiguous by checking whether a second, semantically distinct model also explains the same example litmus tests.

We realize this proposal with MemSynth, a new system for synthesizing axiomatic specifications of memory models from framework sketches and litmus tests. MemSynth provides a language for writing framework sketches, and an efficient engine for synthesizing models in those frameworks. The language and the engine are both based on a deep embedding of bounded relational logic [72, 128] in Rosette [126, 127] (Chapter 2). Relational logic combines first-order logic with relational algebra and transitive closure, providing an expressive semantics that subsumes many recent frameworks for memory models [7, 90, 129, 136]. The bounded version of the logic is decidable by reduction to boolean satisfiability, and existing relational solvers [72, 99, 128] are based on such a reduction. MemSynth takes a radically simpler approach—it delegates the reduction to its host language. Rosette includes a symbolic evaluator that compiles the semantics of its guest languages to efficiently-solvable SMT constraints. MemSynth layers a specialized synthesis algorithm on top of this evaluator, scaling to produce specifications of real memory models in seconds.

The MemSynth synthesizer takes as input a framework sketch and a set of litmus tests. The sketch is a formula in relational logic with missing expressions (holes) over relations defined by the framework (e.g., happens-before [83]). Given these inputs, MemSynth completes the sketch by solving a synthesis query of the form $\exists \varphi_M \in F.\ \bigwedge_{T \in \mathcal{T}_P} \exists I.\ [\![(\!| U_T; \mathcal{V}_T; \varphi_M |\!)]\!] I \wedge \bigwedge_{T \in \mathcal{T}_N} \forall I.\ \neg [\![(\!| U_T; \mathcal{V}_T; \varphi_M |\!)]\!] I$ where F is a framework sketch, and $\mathcal{T}_P$ and $\mathcal{T}_N$ contain litmus tests that demonstrate allowed and forbidden behaviors, respectively. In principle, such a query can be

discharged by generic relational solvers [99] that support higher-order quantification (over the relations E). In practice, however, our queries are intractable for these solvers: their languages lack the constructs (such as sketches and partial interpretations [128]) that enable MemSynth's embedded engine to employ aggressive optimizations based on the structure of litmus tests and framework sketches.

But MemSynth's novel design offers advantages that go beyond scalable synthesis. Being embedded in Rosette, MemSynth provides a platform for rapid development of high-performance tools for reasoning about memory models. For example, we use MemSynth to implement the verification query in five lines of code, obtaining a tool that outperforms dedicated relational solvers [72, 99] and is comparable to existing hand-crafted verifiers [7, 90]. We also implement a novel *ambiguity query* for identifying ambiguities in the set of litmus tests with respect to a framework sketch. The ambiguity query checks whether a memory model uniquely explains a set of litmus tests, and if not, synthesizes another model along with a *distinguishing test* that illustrates the difference between the two models.

We evaluate the scalability and utility of MemSynth's queries using a framework sketch based on work by Alglave et al. [7]. Given this sketch, MemSynth synthesizes a specification for the notoriously relaxed PowerPC architecture from 768 litmus tests in under 12 seconds, including definitions for the subtle cumulativity behavior of PowerPC fences. We also synthesize a specification for the total store ordering (TSO) memory model used by the x86 architecture in under two seconds, using the litmus tests from the Intel Software Developer's Manual [71]. In both cases, our ambiguity query finds that the given litmus tests do not uniquely define their intended memory model—several other models are also consistent with the set of tests. MemSynth synthesizes sets of tests missing from the validation suite of Alglave et al. [7] (for PowerPC) and the Intel manual (for x86) that resolve these ambiguities.

We evaluate MemSynth as a tool-building platform by reproducing results from an existing paper [90] on comparing memory models. In the process, we automatically synthesize a repair for a discrepancy between our framework sketch and the original work—due to a misprint in the paper—which we were unable to fix by hand. The repaired sketch of the paper's framework was developed in two days and achieves the same performance as the existing tool.

3.2 OCELOT: A SOLVER-AIDED RELATIONAL LOGIC LANGUAGE

MemSynth is an engine for automated reasoning about memory models. It builds on a new domain-specific language for relational logic called Ocelot [25]. Ocelot extends bounded relational logic [72, 128] with *expression holes*, which enable *sketching* of memory model frameworks. Thanks to its expressive underlying logic, MemSynth can host many existing frameworks for reasoning about classes of memory models. This section reviews the syntax and semantics of relational logic, and presents our extensions for synthesis problems.

3.2.1 *Bounded Relational Logic*

Relational logic [72] extends classic first-order logic with transitive closure and relational algebra. The inclusion of closure and relations makes this logic ideally suited for reasoning about memory models. In fact, many recent axiomatic memory model frameworks [7, 90, 129, 139] are expressed as first-order constraints on relations that order memory events. MemSynth is based on Ocelot, a new embedding of bounded relational logic [128] in the Rosette solver-aided language [126, 127]. This embedding includes an explicit construct for sketching, and its engine offers optimizations for answering (satisfiability) queries about memory models orders of magnitude faster than general-purpose relational solvers [72, 99].

Syntax. Bounded relational logic (Fig. 3.1) includes the standard connectives and quantifiers of first-order logic, along with the standard operators of relational algebra. A *specification* $(\!|U; D; f|\!)$ in this logic consists of a *universe* of discourse U, a set of *relation declarations* D, and a *formula* f. The universe U is a finite, non-empty set of uninterpreted symbols. A relation declaration $r :_k [R_l, R_u]$ introduces a free variable r (in essence, a Skolem constant), which denotes a *relation* of arity k. Each tuple in this relation consists of k elements drawn from the universe U. The relations R_l and R_u are called the *lower* and *upper* bound on r, and specify the tuples that r must and may contain, respectively. The formula f may refer to the variables r declared in D, but it may not include any other free (unquantified) variables.

Semantics. We define the meaning of a relational specification $s = (\!|U; D; f|\!)$ with respect to an *interpretation* as follows. An interpretation I consists of a universe $U(I)$ and a map of variables to relations drawn from $U(I)$. We say that I satisfies the specification s, written as $I \models s$, if I and s have the same universe of discourse (i.e., $U(I) = U$), if $R_l \subseteq I(r) \subseteq R_u$ for each $r :_k [R_l, R_u]$ in D, and if the formula f evaluates to 'true' in the environment defined by I, i.e., $[\![f]\!]I = \top$.

The semantics of formulas and expressions are standard [128], but we review the most relevant constructs next. The constant univ denotes the universal relation $\{\langle a\rangle \mid a \in U\}$, and iden is the identity relation $\{\langle a, a\rangle \mid a \in U\}$. The multiplicity predicates no, some, and one constrain their argument to contain zero, at least one, and exactly one tuple, respectively. The cross product $X \to Y$ of two relations is the Cartesian product of their tuples. The join $X.Y$ of two relations is the pairwise join of their tuples, omitting the last column of X and first column of Y, on which the two relations are matched. As we will see in Section 3.3.2, memory model specifications make heavy use of these constructs.

Example 3.1. Let the universe be $U = \{a, b, c, d\}$, $X = \{\langle a\rangle, \langle c\rangle\}$ a relation of arity 1 with two tuples, and $Y = \{\langle a, b\rangle, \langle b, d\rangle\}$ a relation of arity 2 with two tuples. We can take the cross product, join, and transitive closure of these relations as follows: $X \to Y = \{\langle a, a, b\rangle, \langle a, b, d\rangle, \langle c, a, b\rangle, \langle c, b, d\rangle\}$, $X.Y = \{\langle b\rangle\}$, $Y.Y = \{\langle a, d\rangle\}$, and $\hat{}Y = \{\langle a, b\rangle, \langle b, d\rangle, \langle a, d\rangle\}$. If we provide the declarations $p :_1 [\{\}, \{\langle a\rangle, \langle c\rangle, \langle d\rangle\}]$ and $q :_2 [\{\langle a, b\rangle\}, \{\langle a, b\rangle, \langle b, d\rangle\}]$, then the interpretation $I = \{p \mapsto X, q \mapsto Y\}$ satisfies the specification $(\!|U; p, q; \text{no } q.p|\!)$ but does not satisfy $(\!|U; p, q; q.q \text{ in } q|\!)$.

$$
\begin{array}{lll}
\text{specification} & s & ::= (\!|U; D; f|\!) \\
\text{universe} & U & ::= \{a[, a]^*\} \\
\text{declarations} & D & ::= \{\} \mid \{d[, d]^*\} \\
\text{declaration} & d & ::= r :_k [b, b] \\
\text{bound} & b & ::= \{(\langle a[, a]^* \rangle)^*\} \\
\text{formula} & f & ::= \text{true} \mid \text{false} \mid e \text{ in } e \mid e = e \mid \text{no } e \mid \\
 & & \quad \text{some } e \mid \text{one } e \mid \text{not } f \mid f \text{ and } f \mid f \text{ or } f \mid \\
 & & \quad f \text{ implies } f \mid f \text{ iff } f \mid \\
 & & \quad \text{all } x : e.\ f \mid \text{exists } x : e.\ f \\
\text{expression} & e & ::= r \mid c \mid e + e \mid e \,\&\, e \mid e - e \mid e.e \mid \\
 & & \quad e \to e \mid {}^\wedge e \mid {\sim} e \mid \{x : e \mid f\} \\
\text{arity} & k & ::= \text{positive integer} \\
\text{relation} & r & ::= \text{identifier} \\
\text{variable} & x & ::= \text{identifier} \\
\text{scalar} & a & ::= \text{identifier} \\
\text{constant} & c & ::= \text{univ} \mid \text{iden}
\end{array}
$$

(a) Abstract syntax

$$
\begin{array}{rl}
[\![(\!|U; d_1, \ldots, d_n; f|\!)]\!]I = & \bigwedge_{i=1}^{n} [\![d_i]\!]I \wedge [\![f]\!]I \wedge (U(I) = U) \\
[\![r :_k [b_L, b_U]]\!]I = & b_L \subseteq I(r) \subseteq b_U \\
[\![\text{true}]\!]I = & \top \\
[\![\text{false}]\!]I = & \bot \\
[\![p \text{ in } q]\!]I = & [\![p]\!]I \subseteq [\![q]\!]I \\
[\![p = q]\!]I = & [\![p]\!]I = [\![q]\!]I \\
[\![\text{no } p]\!]I = & [\![p]\!]I \subseteq \emptyset \\
[\![\text{some } p]\!]I = & \emptyset \subset [\![p]\!]I \\
[\![\text{one } p]\!]I = & |[\![p]\!]I| = 1 \\
[\![\text{not } f]\!]I = & \neg [\![f]\!]I \\
[\![f \text{ and } g]\!]I = & [\![f]\!]I \wedge [\![g]\!]I \\
[\![f \text{ or } g]\!]I = & [\![f]\!]I \vee [\![g]\!]I \\
[\![f \text{ implies } g]\!]I = & [\![f]\!]I \Rightarrow [\![g]\!]I \\
[\![f \text{ iff } g]\!]I = & [\![f]\!]I \Leftrightarrow [\![g]\!]I
\end{array}
\qquad
\begin{array}{rl}
[\![\text{all } x : p.\ f]\!]I = & \wedge_{v \in [\![p]\!]I} [\![f]\!](I[x := v]) \\
[\![\text{exists } x : p.\ f]\!]I = & \vee_{v \in [\![p]\!]I} [\![f]\!](I[x := v]) \\
[\![r]\!]I = & I(r) \\
[\![\text{univ}]\!]I = & \{\langle a \rangle \mid a \in U(I)\} \\
[\![\text{iden}]\!]I = & \{\langle a, a \rangle \mid a \in U(I)\} \\
[\![p + q]\!]I = & [\![p]\!]I \cup [\![q]\!]I \\
[\![p \,\&\, q]\!]I = & [\![p]\!]I \cap [\![q]\!]I \\
[\![p - q]\!]I = & [\![p]\!]I \setminus [\![q]\!]I \\
[\![p.q]\!]I = & \{\langle p_1, \ldots, p_n, q_1, \ldots, q_m \rangle \mid \\
 & \quad \langle p_1, \ldots, p_n, z \rangle \in [\![p]\!]I \wedge \langle z, q_1, \ldots, q_m \rangle \in [\![q]\!]I\} \\
[\![p \to q]\!]I = & \{\langle p_1, \ldots, p_n, q_1, \ldots, q_m \rangle \mid \\
 & \quad \langle p_1, \ldots, p_n \rangle \in [\![p]\!]I \wedge \langle q_1, \ldots, q_m \rangle \in [\![q]\!]I\} \\
[\![{}^\wedge p]\!]I = & [\![p]\!]I \cup [\![p.p]\!]I \cup [\![p.p.p]\!]I \cup \ldots \\
[\![{\sim} p]\!]I = & \{\langle p_2, p_1 \rangle \mid \langle p_1, p_2 \rangle \in [\![p]\!]I\} \\
[\![\{x : p \mid f\}]\!]I = & \{v \in [\![p]\!]I \mid [\![f]\!](I[x := v])\}
\end{array}
$$

(b) Semantics

Figure 3.1: The syntax and semantics of bounded relational logic [128].

INTERPRET(p, I)

Inputs: Relational syntax p; interpretation I

Output: Encoding of the semantics of p (according to Fig. 3.1) with respect to (possibly symbolic) bindings in I

INSTANTIATE(D)

Input: Set of relation declarations $D = \{d_1, \ldots, d_n\}$

Output: Interpretation I that binds each decl. $r :_k [R_l, R_u]$ in D to a matrix with entries

$$m[i_1, \ldots, i_k] = \begin{cases} \top & \langle u_{i_1}, \ldots, u_{i_k} \rangle \in R_l \\ \text{freshSymBool}() & \langle u_{i_1}, \ldots, u_{i_k} \rangle \in R_u \setminus R_l \\ \bot & \text{otherwise} \end{cases}$$

Figure 3.2: Functions provided by the Ocelot DSL for interpreting relational formulas.

3.2.2 *Expression Holes*

To support synthesis, we extend relational logic with *expression holes*, which define the search space for a synthesis query to explore [122]. An expression hole $\mathcal{G}(N, T, d, k)$ is a relational expression that evaluates nondeterministically to one of a finite set of concrete expressions. The set contains all expressions of arity k that can be produced with derivation trees of depth d from a context-free grammar with non-terminals N and terminals T, where the non-terminals are drawn from expression operators in relational logic. Expression holes are a key difference between Ocelot and other relational logic languages such as Kodkod [128] and Alloy* [99], which would require another layer of embedding—building an interpreter for relational logic inside relational logic—to achieve the same result.

Example 3.2. Let X be a relation of arity 1, Y a relation of arity 2, $T = \{X, Y\}$, and $N = \{+, \rightarrow\}$. Then $\mathcal{G}(N, T, 1, 1)$ contains only the expressions X and $X+X$, $\mathcal{G}(N, T, 2, 1)$ additionally contains $X+X+X$ and $X+X+X+X$, and $\mathcal{G}(N, T, 1, 2)$ contains Y, $Y + Y$, and $X \rightarrow X$.

3.2.3 *Relational DSL*

Ocelot is implemented (Fig. 3.2) as a domain-specific language (DSL) in Rosette [126, 127]. The Ocelot interpreter INTERPRET(p, I) takes as input relational syntax p and an interpretation I, and executes the semantics in Fig. 3.1. The interpreter represents relations of arity k in the standard way [72, 128], as boolean matrices of size $|U|^k$, with each cell denoting the presence or absence of a given k-tuple. Relational expressions are then interpreted as matrix operations and formulas as constraints over matrix entries; e.g., relational join becomes matrix multiplication.

Being embedded in Rosette, Ocelot is both an interpreter for bounded relational logic and an engine for answering *relational satisfiability queries*—such as finding an interpretation I that satisfies a specification s, if one exists. We obtain this engine for free by exploiting Rosette's symbolic evaluation facilities. To search for a satisfying interpretation $I \models s$, Ocelot simply evaluates $\textsc{Interpret}(s, I)$ against an interpretation I that binds the free variables in s to matrices populated with *symbolic boolean values* (using the INSTANTIATE function in Fig. 3.2). The result of $\textsc{Interpret}(s, I)$ is a symbolic encoding of the semantics of s, which is then checked for satisfiability with an off-the-shelf SMT solver [53]. This lifted evaluation works both on symbolic interpretations and on specifications that are made symbolic by the inclusion of expression sketches. This evaluation strategy also offers precise state space control: by exploiting domain-specific knowledge to reduce the number of symbolic values in I, Ocelot outperforms state-of-the-art relational solvers [99] as we show in Section 3.6.

3.3 FRAMEWORK SKETCHES

Like existing tools [9, 129, 136], MemSynth specifies memory models as axioms in relational logic that constrain the set of executions allowed for a concurrent program. But unlike existing tools, which take a complete memory model specification as input, MemSynth accepts a sketched specification in the form of a *framework sketch* provided by a memory model designer. Framework sketches are at the core of MemSynth's flexibility as a tool-building platform.

This section defines framework sketches in terms of relational logic, and introduces F_{Alglave}, an example sketch of the Alglave et al. [7] framework for memory models. We use F_{Alglave} to illustrate the automated reasoning queries (Section 3.4) supported by our engine (Section 3.5), and to demonstrate their scalability (Section 3.6).

3.3.1 *Definitions*

A framework sketch (Definition 3.3) consists of two components: a set of axioms φ that contain expression holes (Section 3.2.2), and a function ENC that encodes the syntax and semantics of a litmus test in bounded relational logic. Concurrent programs (without unbounded control flow) have a natural representation [129] in our logic: a program defines a finite universe of discourse U and a set of relations $\mathcal{V} = S \cup E$ over U that encode the test's syntax (S) and its candidate executions (E). For example, S often includes unary relations for each type of instruction in a concurrent program P (such as Read and Write), as well as the *program-order* relation po that relates instructions in the same thread. The relations in E encode possible executions of P by, for example, defining a *happens-before* ordering [83] on the instructions in P (see [7, 89, 129]). The holes in the axioms φ are specified over the relations $S \cup E$ emitted by ENC. A framework sketch $(\varphi, \textsc{Enc})$ thus defines a class of memory models (Definition 3.4) with respect to a framework-specific definition of a litmus test.

Definition 3.3 (Framework sketch)**.** *A* framework sketch *is a pair* $(\varphi, \textsc{Enc})$*, where:*

- φ *is a relational formula containing zero or more expression holes. The relations in* φ *are partitioned into sets* S *and* E*, where relations in* S *characterize the syntax of a concurrent program, and relations in* E *characterize an execution of that program.*
- $\textsc{Enc}$ *is a function that takes as input a concurrent program* P*, and returns a pair* $(U, \mathcal{V})$ *of a relational universe* U *and set of relation declarations* $\mathcal{V}$*, such that every relation in* $S \cup E$ *is bound by* $\mathcal{V}$*.*

We say that a framework sketch allows *a concurrent program* P *if there exists an interpretation* I *such that* $I \models (\!|U; \mathcal{V}; \varphi|\!)$*, where* $(U, \mathcal{V}) = \textsc{Enc}(P)$*. Otherwise, the sketch* forbids *P.*

Definition 3.4 (Memory model)**.** *A* memory model M *is a framework sketch* $(\varphi_M, \textsc{Enc})$ *in which* φ_M *contains no expression holes. We say that* M belongs *to a framework sketch* $F = (\varphi_F, \textsc{Enc})$*, written* $M \in F$*, if and only if* φ_M *can be obtained from* φ_F *by substituting every hole* $h = \mathcal{G}(N, T, d, k)$ *in* φ_F *with a relational expression* $e \in h$*.*

By not mandating a specific definition of a concurrent program P, MemSynth allows framework sketches to define instruction sets and other program structures (e.g., control flow) that are relevant to a given class of memory models. For example, a language memory model would include release/acquire operations [20]; an architectural model, such as F_{Alglave} below (Section 3.3.2), would include fences for a specific architecture (e.g., `mfence` on x86 or `sync` and `lwsync` on PowerPC); and a GPU memory model would include scopes on fence operations [4]. MemSynth requires only that the framework sketch separate the relations S defining a litmus test from the relations E defining an execution of that test, so that it can support a variety of automated reasoning queries in a framework-agnostic way (described in Section 3.4).

3.3.2 $F_{Alglave}$

This section illustrates a framework sketch based on an axiomatic framework by Alglave et al. [7]. We call the corresponding framework sketch F_{Alglave}. We use F_{Alglave} for most of our experiments, although in Section 3.6.2 we construct a second framework sketch based on a different framework.

3.3.2.1 *Litmus Tests*

In F_{Alglave}, a *litmus test* is a small multi-threaded program together with a candidate outcome, expressed as a constraint on the program's final state. For example, the Intel Software Developer's Manual [71] includes the following litmus test to illustrate a surprising behavior allowed by the x86 memory model, where reads may be reordered with earlier writes:

Test x86/3

Thread 1	**Thread 2**
1: X ← 1	3: Y ← 1
2: r1 ← Y	4: r2 ← X

Outcome: $\mathtt{r1} = 0 \wedge \mathtt{r2} = 0$

x86: allowed

We assume that all memory locations (denoted by capital letters) and registers (denoted by r1, r2, etc.) initially hold the value 0 unless stated otherwise. The instruction X ← 1 means that 1 is written to the memory location X, and r1 ← Y means that the value at memory location Y is read into register r1. The outcome is a conjunction of equalities that specify final values of memory (optional) and registers (mandatory).

Given a litmus test, F_{Alglave}'s encoding function $\textsc{Enc}_A$ constructs a universe of *memory events* (i.e., read, write, and fence instructions), *locations*, *threads*, and *values* that appear in the test (Definition 3.5). It also constructs the relations S that encode the syntax of the test, including, for example, unary relations (such as Read) for the types of each instruction of the test. The contents of the syntax relations S are known statically (i.e., the values observed by each read are known from the test's outcome predicate, and we do not handle control dependencies) and extracted automatically from the test.

Definition 3.5 (Litmus test)**.** *A* litmus test *in $F_{Alglave}$ is a small concurrent program together with a postcondition constraint. Given a litmus test T, $F_{Alglave}$'s encoding function $\textsc{Enc}_A(T)$ returns a finite universe of discourse U and a set of relation declarations S over U, defined as follows:*

- *Every relation declaration in S takes the form $r :_k [R, R]$. That is, $I(r) = R$ for all interpretations I, and we say that r is* constant*.*
- *Unary relations* Event*,* Thread*,* Location*, and* Value *partition the universe U into memory events, threads, locations, and values.* Value *always includes the distinguished value* 0*.* Event *is partitioned by* Read*,* Write*,* Fence*, and* LWFence *relations, which contain reads, writes, heavyweight fences, and lightweight fences, respectively.*
- *The* thd *relation is a function from* Event *to* Thread*.*
- loc *and* val *map each event $e \in$ Read $+$ Write to the* Location *and* Value*, respectively, that they read or write.*
- *The program order relation* po *is a strict partial order over* Event *(i.e., irreflexive, transitive, and asymmetric); if $(e_1, e_2) \in$ po, then events e_1 and e_2 share a thread (i.e., e_1.thd $= e_2$.thd) and event e_1 executes before event e_2.*
- *The dependencies relation* dep *is a subset of* po*; if $(e_1, e_2) \in$ dep then event e_2 depends on event e_1.*

- *The final value relation* final *is a partial function from* Location *to* Value*, specifying constraints on the final state of memory imposed by the test's candidate outcome.*

Example 3.6. Consider the test x86/3 above. $\textsc{Enc}_A(\texttt{x86/3})$ defines a universe $U = E \cup L \cup T \cup V$ with four events $E = \{e_1, e_2, e_3, e_4\}$, two locations $L = \{\mathsf{X}, \mathsf{Y}\}$, two threads $T = \{t_1, t_2\}$, and two values $V = \{0, 1\}$. Its relations $\mathcal{V}$ are:

$\mathsf{Read} = \{\langle e_2 \rangle, \langle e_4 \rangle\}$ $\quad$ $\mathsf{Write} = \{\langle e_1 \rangle, \langle e_3 \rangle\}$
$\mathsf{Fence} = \{\}$ $\quad$ $\mathsf{Thread} = \{\langle t_1 \rangle, \langle t_2 \rangle\}$
$\mathsf{LWFence} = \{\}$ $\quad$ $\mathsf{Location} = \{\langle \mathsf{X} \rangle, \langle \mathsf{Y} \rangle\}$
$\mathsf{Value} = \{\langle 0 \rangle, \langle 1 \rangle\}$ $\quad$ $\mathsf{dep} = \{\}$
$\mathsf{po} = \{\langle e_1, e_2 \rangle, \langle e_3, e_4 \rangle\}$ $\quad$ $\mathsf{final} = \{\}$
$\mathsf{thd} = \{\langle e_1, t_1 \rangle, \langle e_2, t_1 \rangle, \langle e_3, t_2 \rangle, \langle e_4, t_2 \rangle\}$
$\mathsf{loc} = \{\langle e_1, \mathsf{X} \rangle, \langle e_2, \mathsf{Y} \rangle, \langle e_3, \mathsf{Y} \rangle, \langle e_4, \mathsf{X} \rangle\}$
$\mathsf{val} = \{\langle e_1, 1 \rangle, \langle e_2, 0 \rangle, \langle e_3, 1 \rangle, \langle e_4, 0 \rangle\}$

3.3.2.2 *Executions*

F_{Alglave} uses two relations, rf and ws, to define the execution of a litmus test (Definition 3.7). The *reads-from* relation rf maps each write event to the reads that observe it: if $(w, r) \in \mathsf{rf}$, then w and r are a write and a read, respectively, to the same address and with the same value. The *write serialization* relation ws places a total order on all writes to the same location. The encoding function $\textsc{Enc}_A$ returns $\{\mathsf{rf}, \mathsf{ws}\}$ as the set of execution relations E for a litmus test T, and it specifies bounds on their contents by automatically extracting them from T.

Definition 3.7 (F_{Alglave} Execution)**.** *In $F_{Alglave}$, an* execution *E of a litmus test T declares two relations:*

- *The* reads-from *relation* rf *is a subset of* $\mathsf{Write} \to \mathsf{Read}$*, such that if* $(w, r) \in \mathsf{rf}$ *then (1)* $w.\mathsf{loc} = r.\mathsf{loc}$ *and* $w.\mathsf{val} = r.\mathsf{val}$*, and (2) for all* $w' \in \mathsf{Write}$*, if* $w' \neq w$ *then* $(w', r) \notin \mathsf{rf}$*.*
- *The* write serialization *relation* ws *is a subset of* $\mathsf{Write} \to \mathsf{Write}$*, such that if* $(w_1, w_2) \in \mathsf{ws}$ *then* $w_1.\mathsf{loc} = w_2.\mathsf{loc}$*, and for every memory location* $l_i \in \mathsf{Location}$*, the relation* $\{(w_1, w_2) \in \mathsf{ws} \mid w_1.\mathsf{loc} = l_i\}$ *is a total order.*

3.3.2.3 *Memory Model*

F_{Alglave} defines a memory model as a relational formula φ_A that constructs a happens-before order and checks its acyclicity. F_{Alglave}'s memory model definition is parametric—many different memory models can be defined within the same framework. This freedom is exposed through three relations $\langle \mathsf{ppo}, \mathsf{grf}, \mathsf{fences} \rangle$ that define the allowed intra-thread reorderings, inter-thread reorderings, and reorderings across fences, respectively. Fig. 3.3 shows examples of these relations for the common sequential consistency (SC) and total store order (TSO) models. The F_{Alglave} formula φ_A replaces these three relations with expression holes for use in synthesis.

$$\mathsf{ppo}_{SC} \triangleq \mathsf{po} \qquad \mathsf{grf}_{SC} \triangleq \mathsf{rf} \qquad \mathsf{fences}_{SC} \triangleq \emptyset$$

(a) Sequential consistency

$$\mathsf{ppo}_{TSO} \triangleq \mathsf{po} - (\mathsf{Write} \rightarrow \mathsf{Read}) \qquad \mathsf{grf}_{TSO} \triangleq \mathsf{rf} - (\mathsf{thd}.\sim\mathsf{thd}) \qquad \mathsf{fences}_{TSO} \triangleq \emptyset$$

(b) Total store order

Figure 3.3: Examples of common memory models defined by hand in the F_{Alglave} framework.

Preserved Program Order. The *preserved program order* relation ppo defines which thread-local reorderings are allowed by a memory model. Given the program order relation po of a litmus test, $\mathsf{ppo} \subseteq \mathsf{po}$ specifies the program-order edges in po that cannot be reordered. In Fig. 3.3, sequential consistency allows no thread-local reordering, while total store order (TSO) allows writes to be reordered beyond later reads by excluding write-to-read edges from ppo.

Global Reads-From. The *global reads-from* relation grf defines which inter-thread communications create ordering requirements between events. Given the reads-from relation rf from an execution (Definition 3.7), grf specifies the edges in rf that must be globally ordered. In Fig. 3.3, sequential consistency allows no reordering, and so every edge in rf creates an ordering obligation. On the other hand, total store order (TSO) allows threads to read their own writes early, and so if a read observes a write on the same thread, it should not create an ordering obligation for other threads.

Fences. The *fences* relation fences defines which events are ordered by a memory fence. For example, the x86 architecture has an `mfence` instruction that serializes all reads and writes issued prior to it. The TSO example in Fig. 3.3 already includes fences in ppo, and so fences is still empty. But some relaxed memory models, such as PowerPC and ARM, also have a notion of fence *cumulativity* [69], in which fence operations create orderings between events on other threads; F_{Alglave} uses fences to model cumulativity. The rules for cumulativity are subtle, but MemSynth correctly synthesizes them for PowerPC in under 12 seconds, as we show in Section 3.6.1.

Axioms. Given the definitions of ppo, grf, and fences, F_{Alglave} uses the axioms in Fig. 3.4 to specify the framework sketch's formula φ_A. The axioms follow Alglave et al. [7], with two changes for better solving performance. First, we omit initialization write events (events that initialize each memory location to 0) in favor of an Init axiom. Second, we use an explicit Final axiom to encode outcome constraints on memory locations, rather than simulating all possible memory states as Alglave et al.'s herd tool does [9].

The first five axioms in Fig. 3.4b define well-formedness of an execution E. The Execution axiom applies the rules in Definition 3.7 to the rf and ws relations. The *initialization* axiom Init states that reads absent from the reads-from relation rf observe the initial value 0. The *uniprocessor* axiom Uniproc requires executions to respect coherence at each memory location. The *thin-air* axiom Thin prevents ex-

$$\mathsf{fr} \triangleq (\sim\mathsf{rf}.\mathsf{ws}) + \{\langle r, w\rangle : \mathsf{Read} \to \mathsf{Write} \mid (\text{no } \mathsf{rf}.r) \text{ and } (r.\mathsf{loc} = w.\mathsf{loc})\}$$
$$\mathsf{ghb} \triangleq \mathsf{ppo} + \mathsf{ws} + \mathsf{fr} + \mathsf{grf} + \mathsf{fences}$$

(a) Auxiliary relations

$$\begin{aligned}
\mathsf{Execution} \triangleq\ & \mathsf{rf} \text{ in } (\mathsf{Write} \to \mathsf{Read}) \mathbin{\&} (\mathsf{loc}.\sim\mathsf{loc}) \mathbin{\&} (\mathsf{val}.\sim\mathsf{val}) \\
& \text{and no } (\mathsf{rf}.\sim\mathsf{rf} - \mathsf{iden}) \\
& \text{and } \mathsf{ws} \text{ in } (\mathsf{Write} \to \mathsf{Write}) \mathbin{\&} \mathsf{loc}.\sim\mathsf{loc} \\
& \text{and no } \mathsf{iden} \mathbin{\&} \mathsf{ws} \\
& \text{and } \mathsf{ws}.\mathsf{ws} \text{ in } \mathsf{ws} \\
& \text{and all } a : \mathsf{Write}.\ \text{all } b : \mathsf{Write}. \\
& \qquad (\text{not } (a = b) \text{ and } a.\mathsf{loc} = b.\mathsf{loc}) \\
& \qquad\quad \text{implies } (\langle a, b\rangle \text{ in } \mathsf{ws} \text{ or } \langle b, a\rangle \text{ in } \mathsf{ws}) \\
\mathsf{Init} \triangleq\ & \text{all } r : \mathsf{Read}.\ (\text{no } \mathsf{rf}.r) \text{ implies } r.\mathsf{val} = 0 \\
\mathsf{Uniproc} \triangleq\ & \text{no } \hat{}(\mathsf{rf} + \mathsf{ws} + \mathsf{fr} + (\mathsf{po} \mathbin{\&} \mathsf{loc}.\sim\mathsf{loc})) \mathbin{\&} \mathsf{iden} \\
\mathsf{Thin} \triangleq\ & \text{no } \hat{}(\mathsf{rf} + \mathsf{dep}) \mathbin{\&} \mathsf{iden} \\
\mathsf{Final} \triangleq\ & \text{all } w : \mathsf{Write}.\ (w \text{ in } (\mathsf{univ}.\mathsf{ws} - \mathsf{ws}.\mathsf{univ}) \text{ and some } (w.\mathsf{loc}).\mathsf{final}) \\
& \qquad\qquad \text{implies } w.\mathsf{val} = w.\mathsf{loc}.\mathsf{final} \\
\mathsf{Acyclic} \triangleq\ & \text{no } \hat{}\mathsf{ghb} \mathbin{\&} \mathsf{iden} \\
\mathsf{Valid} \triangleq\ & \mathsf{Execution} \text{ and } \mathsf{Init} \text{ and } \mathsf{Uniproc} \text{ and } \mathsf{Thin} \text{ and } \mathsf{Final} \text{ and } \mathsf{Acyclic}
\end{aligned}$$

(b) Axioms

Figure 3.4: The axioms of the F_{Alglave} framework extend those of Alglave et al. [7], with changes to remove initialization write events and support outcomes for memory locations.

ecutions that create values out of thin air (i.e., involve cyclic dependencies). Lastly, the *final value* axiom Final imposes the constraints defined by the final relation.

To define whether an execution is *allowed*, F_{Alglave} constructs a *global happens-before* order ghb reflecting the orderings between events induced by the memory model. The Valid axiom allows a test if there exists some valid execution for which the global happens-before relation is acyclic (i.e., no event is transitively reachable from itself). That is, F_{Alglave}'s framework sketch $(\varphi_A, \textsc{Enc}_A)$ defines $\varphi_A \triangleq \mathsf{Valid}$.

3.4 MEMORY MODEL QUERIES

MemSynth is designed to efficiently answer four queries about memory models from a given framework sketch:

Verification determines whether a litmus test is allowed or forbidden by a memory model;

Synthesis searches for a memory model that produces desired outcomes on a set of litmus tests;

Equivalence determines whether two memory models are equivalent (within finite bounds); and

Ambiguity decides whether a memory model uniquely explains the outcomes of a set of litmus tests, and if not, synthesizes a disambiguating test.

This section defines the MemSynth queries and explains their utility in building and refining memory model specifications. Section 3.5 shows how to implement these queries to scale to hundreds of litmus tests and large specifications.

3.4.1 *Verification*

The verification query, determining whether a memory model allows a litmus test, is well-studied in the literature [7, 87, 90, 129, 139]. Given a litmus test T and memory model $M = (\varphi, \textsc{Enc})$ (Definition 3.4), the verification query checks satisfiability of the formula

$$\exists I. [\![(\!| U; \mathcal{V}; \varphi |\!)]\!] I$$

where $(U, \mathcal{V}) = \textsc{Enc}(T)$. If this formula is satisfiable, then M allows the test T (Definition 3.3). Otherwise, M forbids T. The verification query involves a straightforward satisfiability check that can be discharged with any relational solver, including MemSynth.

3.4.2 *Synthesis*

The synthesis query searches a framework sketch for a memory model that is consistent with the desired outcomes for a set of litmus tests. Given a set $\mathcal{T}_P$ of tests that should be allowed, a set $\mathcal{T}_N$ of tests that should be forbidden, and a framework

sketch $F = (\varphi, \textsc{Enc})$, the synthesis task is to find a memory model $(\varphi_M, \textsc{Enc}) \in F$ that allows all tests in $\mathcal{T}_P$ and forbids all tests in $\mathcal{T}_N$. This query amounts to solving the formula

$$\begin{aligned}\exists(\varphi_M, \textsc{Enc}) \in F.\ & \bigwedge_{T \in \mathcal{T}_P} \exists I.\ [\![(\!| U_T; \mathcal{V}_T; \varphi_M |\!)]\!] I \\ & \wedge \bigwedge_{T \in \mathcal{T}_N} \forall I.\ \neg [\![(\!| U_T; \mathcal{V}_T; \varphi_M |\!)]\!] I\end{aligned} \tag{1}$$

where $(U_T, \mathcal{V}_T) = \textsc{Enc}(T)$.

The synthesis query involves higher-order universal quantification over the non-constant relations in $\mathcal{V}_T$ for forbidden tests $\mathcal{T}_N$ The recent Alloy* solver [99] supports finite model finding for relational formulas with higher-order quantifiers, and so could in principle solve the synthesis query. In practice, however, these queries are intractable for Alloy* because its language lacks crucial constructs for precisely specifying the size and shape of the search space: expression holes and bounds on the contents of declared relations. These limitations motivated our embedding of bounded relational logic in Rosette (Section 3.2). In Section 3.5.2, we present an algorithm for solving synthesis queries that scales to complex framework sketches and many litmus tests.

3.4.3 *Equivalence*

MemSynth can compare two memory models M_A and M_B from a framework F for equivalence. If they are not equivalent, MemSynth generates a *distinguishing litmus test* T_D on which they disagree (i.e., one model allows T_D while the other forbids it). As with existing work on generating distinguishing tests [89, 136], the equivalence check is bounded, proving two models equivalent only up to a bound on the size of the distinguishing test. These bounds are defined by a *symbolic* litmus test (Definition 3.8), in which some syntax relations S are not constant (in contrast to, e.g., Definition 3.5). A symbolic litmus test thus defines a set of concurrent programs rather than only one such program.

Definition 3.8 (Symbolic litmus test)**.** *A* symbolic litmus test $T_S = (\!| U; \mathcal{V}; f |\!)$ *for a framework sketch* $F = (\varphi, \textit{Enc})$ *is a relational specification in which*

- $\mathcal{V}$ *binds the relations* $S \cup E$ *in* φ *(as in Definition 3.3).*
- *The formula* f *is a* well-formedness *predicate for the litmus test, in which the only relations are those in* S*.*

Given a symbolic litmus test T_S and two memory models $M_A = (\varphi_A, \textsc{Enc})$ and $M_B = (\varphi_B, \textsc{Enc})$, the equivalence query solves for a distinguishing litmus test by checking the satisfiability of two formulas:

$$\begin{aligned}\exists I_T.\ & [\![T_S]\!] I_T \wedge \exists I. [\![(\!| U; \mathcal{V}; \varphi_A |\!)]\!] (I_T \cup I) \\ & \wedge \forall I. \neg [\![(\!| U; \mathcal{V}; \varphi_B |\!)]\!] (I_T \cup I)\end{aligned}$$

to find a test on which M_A is weaker than M_B (i.e., M_A allows a test that M_B forbids), and similarly the second formula

$$\exists I_T.\ [\![T_S]\!]I_T \wedge \exists I.[\![(\!|U; \mathcal{V}; \varphi_B|\!)]\!](I_T \cup I) \\ \wedge \forall I.\neg[\![(\!|U; \mathcal{V}; \varphi_A|\!)]\!](I_T \cup I)$$

for a test on which M_A is stronger than M_B. The symbolic litmus test $T_S = (\!|U; \mathcal{V}; f|\!)$ includes a well-formedness predicate f, a relational formula that ensures the resulting test is a syntactically valid program. If either formula is satisfiable, then $T_D = \textsc{Eval}(T_S, I_T)$ is a litmus test that distinguishes the two models M_A and M_B.[2] If both formulas are unsatisfiable, then M_A and M_B are equivalent on all valid tests in the search space defined by T_S.

3.4.4 *Ambiguity*

The ambiguity query checks whether a memory model M is the only one within a framework sketch that gives the desired outcomes on a set of allowed ($\mathcal{T}_P$) and forbidden ($\mathcal{T}_N$) litmus tests. To do so, the query attempts to synthesize a second memory model M_S and a distinguishing litmus test T_D such that M_S and M disagree on T_D but agree on all tests in $\mathcal{T}_P$ and $\mathcal{T}_N$. If such a model and test exist, the set of given tests is ambiguous: there are two semantically distinct memory models that both explain the input tests $\mathcal{T}_P \cup \mathcal{T}_N$.

Given a memory model $M = (\varphi_M, \textsc{Enc})$, a framework sketch $F = (\varphi, \textsc{Enc})$, a symbolic litmus test $T_S = (\!|U_S; \mathcal{V}_S; f|\!)$, and sets of allowed and forbidden tests $\mathcal{T}_P$ and $\mathcal{T}_N$, detecting ambiguity involves checking the satisfiability of a formula that combines synthesis and equivalence:

$$\begin{aligned}\exists I_T.\exists(\varphi_S, \textsc{Enc}) \in F.\ [\![T_S]\!]I_T \wedge &\bigwedge_{T\in\mathcal{T}_P} \exists I.\ [\![(\!|U_T; \mathcal{V}_T; \varphi_S|\!)]\!]I \\ &\wedge \bigwedge_{T\in\mathcal{T}_N} \forall I.\ \neg[\![(\!|U_T; \mathcal{V}_T; \varphi_S|\!)]\!]I \\ &\wedge \exists I.[\![(\!|U_S; \mathcal{V}_S; \varphi_S|\!)]\!](I_T \cup I) \\ &\wedge \forall I.\neg[\![(\!|U_M; \mathcal{V}_M; \varphi_M|\!)]\!](I_T \cup I)\end{aligned}$$

where $(U_T, \mathcal{V}_T) = \textsc{Enc}(T)$, and a second formula that swaps M_S and M in the final two conjuncts (akin to the two equivalence formulas). If either formula is satisfiable, then $M_S = (\varphi_S, \textsc{Enc})$ is a second memory model that produces the desired outcomes on all tests in $\mathcal{T}_P$ and $\mathcal{T}_N$, and $T_D = \textsc{Eval}(T_S, I_T)$ is a litmus test that distinguishes M and M_S. If both formulas are unsatisfiable, then M is the only memory model that produces the desired outcomes. This uniqueness result is with respect to two bounds: the finite search space defined by the framework sketch F, and the finite search space for the symbolic litmus test T_S.

The ambiguity query identifies missing tests from the input sets, and so can form the basis of a refinement loop to guide the development of a memory model

2 $\textsc{Eval}(T_S, I_T)$ substitutes each variable v in T_S with the value $I(v)$.

```
1 function Verify(M = (φ_M, Enc), T)
2     (U, V) ← Enc(T)
3     I ← Instantiate(V)
4     φ ← Interpret(φ_M, I)
5     return Solve(φ) = SAT
```

Figure 3.5: MemSynth's verification procedure Verify takes as input a memory model M and litmus test T and determines whether M allows T.

```
1 function Enc_A(T)
2     (U, V) ← EncSyntax(T)              ▷ Encode relations in Definition 3.5
3     I ← Instantiate(V)                 ▷ Make an interpretation from V
4     B_u^rf ← Interpret((Write → Read) & (loc.~loc) & (val.~val), I)
5     B_u^ws ← Interpret((Write → Write) & (loc.~loc), I)        ▷ Fig. 3.4
6     return (U, V ∪ {rf :2 [∅, B_u^rf], ws :2 [∅, B_u^ws]})
```

Figure 3.6: The $\textsc{Enc}_A$ procedure computes relational bounds for an execution E in the F_{Alglave} framework.

specification. For example, if we take $\mathcal{T}_P$ to contain only the test `x86/3` from Section 3.3.2, and $\mathcal{T}_N$ to be empty, then many distinct memory models within F_{Alglave} produce the desired outcomes (TSO, RMO, PowerPC, etc.). If we take M to be one such model, the ambiguity query will identify a second model that also allows test `x86/3`, and produce a new distinguishing litmus test T_D to resolve the ambiguity. By deciding the desired outcome for T_D and adding it to the appropriate set ($\mathcal{T}_P$ or $\mathcal{T}_N$), we can repeat the synthesis process to refine the memory model M. The user can decide on the desired outcome for T_D by inspecting documentation, executing the test on hardware, consulting with system architects, or otherwise.

3.5 REASONING ENGINE

This section presents MemSynth's engine for answering the queries in Section 3.4. We show the algorithms to implement these queries, and describe key optimizations to make them scale to real-world memory models.

3.5.1 *Verification*

The verification query (Section 3.4.1) determines whether a memory model M allows a litmus test T. The Verify procedure in Fig. 3.5 takes as input a memory model $M = (\varphi, \textsc{Enc})$ and litmus test T, and returns true iff M allows T. The Verify procedure first encodes the litmus test as a finite universe U and set of relation declarations $\mathcal{V}$ using the memory model's Enc function (Definition 3.3). Given these bounds, it then checks the satisfiability of the relational specification $(\!|U; \mathcal{V}; \varphi|\!)$. The implementation of Verify is only four lines of code, demonstrating the utility of our relational DSL for reasoning about memory models.

Bounds Compaction. Fig. 3.6 shows an example implementation of the ENC function. The ENC_A procedure computes bounds for the relations in a F_{Alglave} execution (Definition 3.7). A naive bound that includes every tuple of the appropriate arity is sound, but tighter bounds can significantly improve performance, since the difference between the upper and lower bounds for each free relation defines the size of the search space for the solver query. For F_{Alglave}, an execution consists of two relations rf and ws that specify a reads-from and write serialization order, respectively. ENC_A computes upper bounds for each relation from the Execution axiom in Fig. 3.4. The rf relation contains only tuples (w, r) where w is a write, r is a read, and both w and r access the same location with the same value. Likewise, the ws relation contains only tuples (w_1, w_2) where both entries are writes to the same location. Compared to naive upper bounds, this more compact search space improve verification time by an average of 27× on the PowerPC tests discussed in Section 3.6.1.

3.5.2 *Synthesis*

The synthesis query (Section 3.4.2) generates a memory model that gives the desired outcomes on a set of litmus tests. The space of candidate solutions is defined by a framework sketch $F = (\varphi, \text{ENC})$ (Definition 3.3), which contains expression holes that define a candidate space of memory models.

Our synthesis procedure, SYNTHESIZE (Fig. 3.7), takes as input a framework sketch $F = (\varphi, \text{ENC})$, a set of allowed litmus tests $\mathcal{T}_P$, and a set of forbidden litmus tests $\mathcal{T}_N$. Given these inputs, it uses our relational DSL (embedded in Rosette) to generate and solve quantified formulas using an off-the-shelf SMT solver [53]. Because MemSynth represents relations as matrices of boolean values, these formulas quantify over boolean variables. We found the Z3 SMT solver [53] to be extremely effective at discharging these formulas—an average of 2–5× faster than our own specialized implementation of counterexample-guided inductive synthesis [122].

SYNTHESIZE does not try to find a correct model for all tests in $\mathcal{T}_P$ and $\mathcal{T}_N$ at once, since this would require encoding every test against the framework sketch predicate φ. Instead, tests are added to the synthesis query incrementally. The order in which tests are added influences synthesis performance; we use a simple heuristic that adds tests in increasing order of size, which optimizes for small search spaces. This incrementalization reduces the size of the synthesis query substantially: in Section 3.6.1, we show that only 16 of 768 tests were added to the query when synthesizing a model for PowerPC.

The SYNTHESIZE procedure is sound, and it is complete with respect to the input sketch: if a correct model exists within the input sketch, SYNTHESIZE will return a solution.

Theorem 3.9 (Soundness)**.** *If* $\textsc{Synthesize}(F, \mathcal{T}_P, \mathcal{T}_N)$ *returns a memory model M, then M satisfies Eq.* (1)*.*

Proof. $\textsc{Synthesize}(F, \mathcal{T}_P, \mathcal{T}_N)$ can return a memory model M only from line 14. To reach this point, T must be $\perp$, which happens only if there is a candidate model

```
1  function Synthesize(F = (φ, Enc), T_P, T_N)
2      S ← new IncrementalSMTSolver()
3      T_U ← {}                                          ▷ Set of used tests
4      φ_M ← false                                       ▷ Model that forbids all outcomes
5      T ← NextTest(φ_M, T_P, T_N, T_U)                  ▷ Choose an initial test
6      while T ≠ ⊥ do
7          AddTest(S, F, T, T_P)                         ▷ Add encoding of T to S
8          T_U ← T_U ∪ T
9          I_b ← Solve(S)                                ▷ Boolean interpretation or UNSAT
10         if I_b = UNSAT then                           ▷ No model exists
11             return UNSAT
12         φ_M ← Eval(φ, I_b)                            ▷ Use I_b to fill the holes in φ
13         T ← NextTest(φ_M, Enc, T_P, T_N, T_U)         ▷ Choose the next test
14     return (φ_M, Enc)    ▷ M gives the expected outcome on all tests in T_P ∪ T_N
```

(a) Main synthesis routine

```
1  function AddTest(S, F = (φ, Enc), T, T_P)
2      (U, V) ← Enc(T)
3      I ← Instantiate(V)                                ▷ Symbolic relational interpretation I
4      ϕ ← Interpret(φ, I)                               ▷ Boolean encoding
5      if T ∈ T_P then
6          Assert(S, ϕ)                                  ▷ Add an allowed test
7      else
8          X ← Symbolics(I)                              ▷ All symbolic booleans in I
9          Assert(S, ∀X. ¬ϕ)                             ▷ Add a forbidden test
```

(b) Test evaluation

```
1  function NextTest(φ_M, Enc, T_P, T_N, T_U)
2      for T ∈ (T_P ∪ T_N) \ T_U do                      ▷ Iterate over unused tests
3          if Verify((φ_M, Enc), T) ≠ (T ∈ T_P) then
4              return T                                  ▷ M gives the wrong outcome on T
5      return ⊥                    ▷ M gives the expected outcome on all unused tests
```

(c) Test selection

Figure 3.7: MemSynth's synthesis procedure Synthesize takes as input a memory model sketch $\mathcal{M}$, a set $\mathcal{T}_P$ of allowed litmus tests, and a set $\mathcal{T}_N$ of forbidden litmus tests, and returns a memory model that produces the given outcomes on all tests.

M for which NEXTTEST returns $\perp$. NEXTTEST returns $\perp$ only if M is correct for every test in the set $(\mathcal{T}_P \cup \mathcal{T}_N) \setminus \mathcal{T}_U$; that is, M satisfies:

$$\bigwedge_{T \in \mathcal{T}_P \setminus \mathcal{T}_U} \exists I.\ [\![\text{Allow}(M, T, \text{EXEC}(T))]\!] I$$
$$\wedge \bigwedge_{T \in \mathcal{T}_N \setminus \mathcal{T}_U} \forall I.\ \neg [\![\text{Allow}(M, T, \text{EXEC}(T))]\!] I$$

So M satisfies Eq. (1) for all tests other than those in $\mathcal{T}_U$.

Because the solver S is incremental, and is never reset, a candidate model M returned from the call to SOLVE on line 9 satisfies all constraints ever added to S. Only ADDTEST adds constraints to S, and it is invoked once for each test added to $\mathcal{T}_U$. For any $T \in \mathcal{T}_U$, if $T \in \mathcal{T}_P$ then the constraint

$$\exists I.\ [\![\text{Allow}(F, T, \text{EXEC}(T))]\!] I$$

is added to S at line 6 of ADDTEST. Otherwise, $T \in \mathcal{T}_N$, and the constraint

$$\forall I.\ \neg [\![\text{Allow}(F, T, \text{EXEC}(T))]\!] I$$

is added to S by lines 8–9 of ADDTEST. Observe that the holes in F are also encoded using symbolic boolean values, and these values appear in the encodings for both allowed and forbidden tests. Therefore, whenever SOLVE is invoked by SYNTHESIZE, the state of S is:

$$\bigwedge_{T \in \mathcal{T}_P \cap \mathcal{T}_U} \exists I.\ [\![\text{Allow}(F, T, \text{EXEC}(T))]\!] I$$
$$\wedge \bigwedge_{T \in \mathcal{T}_N \cap \mathcal{T}_U} \forall I.\ \neg [\![\text{Allow}(F, T, \text{EXEC}(T))]\!] I$$

So any satisfying boolean interpretation I_b returned by SOLVE includes a memory model $M = \text{EVAL}(F, I_b)$ that satisfies Eq. (1) for all tests in $\mathcal{T}_U$. Combining the two results, we have that if SYNTHESIZE returns M then M satisfies Eq. (1). □

Theorem 3.10 (Termination)**.** *SYNTHESIZE$(F, \mathcal{T}_P, \mathcal{T}_N)$ terminates when $\mathcal{T}_P$ and $\mathcal{T}_N$ are finite sets.*

Proof. The formulas sent to the solver S fall into the effectively propositional fragment of first-order logic, which is decidable by SMT solvers such as Z3, and so calls to SOLVE will always terminate. SYNTHESIZE can always terminate early if NEXTTEST returns $\perp$ because the current candidate M is correct. In the worst case, since $\mathcal{T}_P$ and $\mathcal{T}_N$ are finite, and each loop iteration in SYNTHESIZE grows $\mathcal{T}_U$ by exactly one test, NEXTTEST will eventually be invoked with $\mathcal{T}_U = \mathcal{T}_P \cup \mathcal{T}_N$, and will therefore return $\perp$ (since the domain of its loop is empty), terminating the SYNTHESIZE loop. □

Theorem 3.11 (Completeness)**.** *If there exists a model M in the framework sketch F that satisfies Eq.* (1)*, and $\mathcal{T}_P$ and $\mathcal{T}_N$ are finite sets, then SYNTHESIZE$(F, \mathcal{T}_P, \mathcal{T}_N)$ will return a model.*

Proof. Suppose there is such a model M. By Theorem 3.10, we know that SYNTHESIZE terminates, so we need only show that it terminates while returning a model. Suppose for a contradiction that SYNTHESIZE instead returns UNSAT. This happens only if the synthesis query on line 9 is unsatisfiable. From the proof of Theorem 3.9, whenever line 9 is invoked, the state of the synthesizer S is of the form

$$\bigwedge_{T \in \mathcal{T}_P \cap \mathcal{T}_U} \exists I.\ [\![\text{Allow}(\mathcal{M}, T, \text{EXEC}(T))]\!] I$$
$$\wedge \bigwedge_{T \in \mathcal{T}_N \cap \mathcal{T}_U} \forall I.\ \neg [\![\text{Allow}(\mathcal{M}, T, \text{EXEC}(T))]\!] I$$

But note that Eq. (1) implies this formula, since each conjunction is a subset of the tests in Eq. (1). So if this formula is unsatisfiable, so is Eq. (1), which is a contradiction. □

3.5.3 *Equivalence*

MemSynth can determine if two memory models are equivalent (up to given bounds) by searching for a litmus test on which they disagree. Our equivalence-checking procedure $\text{COMPARE}(M_A, M_B, T_S)$ takes as input two memory models M_A and M_B, and a designer-provided symbolic litmus test T_S (Definition 3.8). Given these inputs, it returns either a litmus test T such that $\text{VERIFY}(M_A, T) \neq \text{VERIFY}(M_B, T)$, or $\perp$ if no such test exists within the bounds of T_S. To search for a distinguishing test T, COMPARE solves the two quantified boolean equivalence formulas shown in Section 3.4.3 using the Z3 SMT solver (as with SYNTHESIZE), with two additional optimizations described next.

Symmetry Breaking. For most framework sketches, a naive specification of a symbolic litmus test will define a search space that contains many redundant candidate tests. For example, after checking a test T in F_{Alglave}, there is no need to also check a test T' that differs from T by a permutation of the used memory locations (e.g., T' swaps all instances of $\mathtt{x}$ and $\mathtt{y}$ in the loc relation of T). To improve query performance, our definition of T_S for F_{Alglave} applies lex-leader symmetry breaking [50] to rule out tests that differ only by a permutation of threads, addresses, or values, similar to existing work [89]. The well-formedness predicate f for T_S also adds assertions to rule out other uninteresting litmus tests, such as tests that refer to a memory location exactly once, which has no visible effect on inter-thread memory reorderings. These optimizations reduce the run time of equivalence queries by 2–10×, and generalize beyond F_{Alglave}.

Concretization with Metasketches. As another critical optimization, we express the symbolic litmus test T_S using a metasketch (Chapter 4), which decomposes T_S into a set of *partially concretized* symbolic tests. In particular, T_S describes the set of all litmus tests with up to k threads and up to n instructions per thread. The corresponding metasketch describes the same search space using a *set* of symbolic tests of the form $T_S^{\langle k,(t_1,\ldots,t_k),(w_1,\ldots,w_k)\rangle}$, each of which encodes the space of all litmus

tests with a concrete number of threads (k), instructions per thread ($t_1, \ldots, t_k$), and writes per thread ($w_1, \ldots, w_k$). For example, the set $T_S^{\langle 2,(2,3),(1,2)\rangle}$ contains all tests with two threads, with two instructions (one of which is a write) on the first thread, and three instructions (two of which are writes) on the second thread. This concretization enables each symbolic litmus test in the metasketch to use more compact bounds (e.g., the thread relation `thd` becomes entirely concrete), which reduces the search space exponentially. Without this optimization, the equivalence queries in Section 3.6.3 are up to two orders of magnitude slower.

3.5.4 *Ambiguity*

The final MemSynth query checks whether a memory model is unique for a set of allowed tests $\mathcal{T}_P$ and forbidden tests $\mathcal{T}_N$, and if not, synthesizes a disambiguating litmus test. The ambiguity procedure $\textsc{Disambiguate}(M, \mathcal{T}_P, \mathcal{T}_N, F, T_S)$ takes as input a memory model M, sets of allowed tests $\mathcal{T}_P$ and forbidden tests $\mathcal{T}_N$, a framework sketch $F = (\varphi, \textsc{Enc})$, and a symbolic litmus test T_S. It returns a new memory model M_S and test T_D, such that for all $T \in \mathcal{T}_P \cup \mathcal{T}_N$, $\textsc{Verify}(M, T) = \textsc{Verify}(M_S, T)$, but $\textsc{Verify}(M, T_D) \neq \textsc{Verify}(M_S, T_D)$. In other words, the set of tests $\mathcal{T}_P \cup \mathcal{T}_N$ is ambiguous, because both M and M_S satisfy every test in the set. Since the ambiguity query involves synthesizing a memory model M_S and litmus test T_D, the implementation of Disambiguate extends Synthesize (Fig. 3.7) and benefits from the same optimizations as Compare, i.e., metasketches and symmetry breaking. Metasketches also enable solving in parallel [27], which Disambiguate exploits to gain up to 3× speedup on 8 threads in our experiments.

3.5.5 *Discussion*

Limitations. As with other tools based on syntax-guided synthesis [122], MemSynth's results are inherently bounded. Both framework sketches (for synthesis) and symbolic litmus tests (for equivalence and ambiguity) define large but finite search spaces, which MemSynth explores exhaustively with an SMT solver. While incomplete, such bounded reasoning provides useful results on real-world problems, as Section 3.6 shows; in most of those experiments, increasing the bounds yielded no meaningful difference in the results.

MemSynth's synthesis queries also face the potential for overfitting, like other example-based synthesis tools. The relational DSL (Section 3.2) reduces this risk by not including operators prone to overfitting (e.g., if-then-else expressions), and by offering control over the size of the search space for expression holes. The Compare and Disambiguate queries can also exploit metasketch support for cost functions [27] to minimize the size of the synthesized tests.

Integration. MemSynth queries read and write litmus tests in the common Herd format [9], allowing them to integrate with existing tools for memory models. MemSynth's relational DSL can also export models to Alloy* specifications, which are used by some memory model tools [129, 136]. MemSynth's relational DSL (Section 3.2) is similar to the `cat` language [3] for specifying memory models,

and so MemSynth models could be exported for use by that toolchain. But `cat` includes fixpoint operations, while our DSL does not, so importing `cat` models into MemSynth would require more work (e.g., bounded unwinding of fixpoints [136]).

3.6 CASE STUDIES

To demonstrate that MemSynth is an effective approach to reasoning about memory models, we sought to answer three research questions:

- Can MemSynth scale to real-world memory models such as PowerPC and x86?
- Does MemSynth provide a basis for rapidly building useful automated memory model tools?
- Does MemSynth outperform existing relational solvers and memory model tools?

Methodology and Code. Experiments in this section were performed on a quad-core Intel Core i7-7700K CPU at 4.8 GHz, with 16 GB of RAM. We used Rosette [126, 127] version 2.2 and Z3 [53] version 4.5.0. Both MemSynth and our relational DSL, Ocelot, are open source and available from `http://memsynth.uwplse.org`, together with the synthesized models and tests from this section and a virtual machine artifact for reproducing the results.

3.6.1 *Can MemSynth scale to real-world memory models such as PowerPC and x86?*

This section uses MemSynth to synthesize specifications for the PowerPC [69] and x86 [71] memory models. The results (summarized in Fig. 3.8) show that MemSynth scales to complex real-world models, and that its queries can aid in the design of memory model specifications by identifying ambiguities and redundancies in tests and documentation.

3.6.1.1 *Synthesizing a PowerPC Model*

The PowerPC architecture is well-known for relaxed memory behaviors that have proven difficult to formalize. Existing formalization efforts have identified subtle mis-specifications [5, 7, 91], making an automated process particularly appealing. To synthesize a specification for PowerPC, MemSynth uses a set of 768 litmus tests from Alglave et al. [6, 7], which they generated with their diy tool [8]. These tests vary from 6–24 instructions across 2–5 threads, and while they examine most aspects of the PowerPC memory model, they are not intended to be exhaustive. We use the Alglave et al. [7] model to decide whether each test should be allowed, although we could use hardware observations instead, as discussed later.

We employ F_{Alglave} as the basis for the synthesis process. The framework sketch contains expression holes for the ppo, grf, and fences relations. All three holes use a grammar containing all relational expressions e in Fig. 3.1 other than set comprehension and closure. For the barrier expression fences, we provide a sketch

	Input Tests			Framework Sketch		
Arch.	$\lvert\mathcal{T}_P\rvert$	$\lvert\mathcal{T}_N\rvert$	Time	ppo/grf Depth	fences Depth	State Space
PPC	163	605	12 s	4	4	2^{1406}
x86	2	8	2 s	4	0	2^{624}

(a) Synthesis results

			Symbolic Litmus Test		
Arch.	New Tests	Time	Num. Threads	Num. Events	State Space
PPC	9	110 min	2–4	2–6	2^{165}
x86	4	67 min	2–4	2–6	2^{114}

(b) Ambiguity results

Figure 3.8: Results of real-world memory model synthesis and ambiguity experiments for PowerPC and x86. We describe the framework sketches and symbolic litmus tests both in terms of their parameters (e.g., expression hole depth) and the number of candidate solutions they contain (i.e., their state space). The ambiguity results (b) for a given architecture use the same framework sketch as the synthesis results (a) for that architecture.

of the form $\mathsf{fences} \triangleq \mathsf{F}_{\mathsf{Fence}} + \mathsf{F}_{\mathsf{LWFence}}$, where $\mathsf{F}_{\mathsf{Fence}}$ and $\mathsf{F}_{\mathsf{LWFence}}$ are expression holes containing Fence and LWFence, respectively, as terminals. This sketch expresses the high-level insight that PowerPC features two kinds of cumulative barriers (heavyweight `sync` fences and lightweight `lwsync` fences) that do not interact.

Synthesis. MemSynth synthesizes a model, which we call PPC_0, that agrees with Alglave et al.'s hand-written model on all 768 tests. The synthesis takes 12 seconds, and due to its heuristics for test ordering, the incremental synthesis algorithm (Fig. 3.7) uses only 16 of the 768 tests.

Ambiguity. While the 768 tests described above cover much of the semantics of PowerPC, they do not identify a unique model. To resolve this ambiguity, we apply MemSynth's DISAMBIGUATE query (Section 3.4.4) to enlarge the set until it identifies a single model. We use the Alglave et al. model as an oracle to decide the correct outcome for the generated distinguishing tests.

MemSynth finds 9 new tests to add to the set. The tests deal with the semantics of PowerPC barriers; for example:

Test `ppc/ambig/3`

Thread 1	**Thread 2**
1: `r1` ← `B`	4: `r2` ← `A`
2: `lwsync`	5: `lwsync`
3: `A` ← `1`	6: `B` ← `1`

Outcome: $\texttt{r1} = 1 \wedge \texttt{r2} = 1$

PowerPC: forbidden

After adding the 9 tests, the new synthesized model PPC_1 is equivalent to the Alglave et al. [7] model on all tests up to 6 instructions across 4 threads, and is the only model (within our sketch) that produces the given outcomes on all tests.

Discussion. MemSynth is complementary to test-generation tools such as `diy` [7]: these tools can seed the synthesis process with initial tests, and MemSynth can then identify ambiguities and synthesize new tests to resolve them. While our experiments use the hand-written model of Alglave et al. [7] as an oracle, we could instead determine litmus test outcomes by manually consulting documentation or by hardware experiments. For example, Alglave et al. [6] also ran their 768 tests on PowerPC hardware and observed whether each behavior occurred. MemSynth is able use the results of these experiments as an oracle, and synthesizes a new model PPC_H in 13 seconds. The resulting model is not equivalent to PPC_0 (MemSynth synthesizes a distinguishing test with its equivalence query in 6 seconds) because some allowed outcomes were not observed on the hardware.

3.6.1.2 *x86 Ambiguity and Redundancy*

The x86 architecture specifies a variant of *total store ordering* (TSO) as its memory model. The x86 TSO memory model is defined in the Intel Software Developer's Manual [71] with prose and a set of 10 litmus tests. Though TSO is one of the simplest memory models, formalizing the subtleties of its x86 variant has been challenging [31, 32, 114, 119].

We used MemSynth to synthesize a specification of the x86 memory model. To do so, we extended F_{Alglave} with support for atomic operations (adding a new unary Atomic relation to Definition 3.5 to model x86's `xchg` instruction) and the `mfence` full memory fence (populating the Fence relation in Definition 3.5). MemSynth synthesizes a formalization TSO_0 that is correct on the Intel manual's 10 litmus tests in under two seconds. Fig. 3.9 shows the framework sketch F_{Alglave} and the synthesized model TSO_0.

Ambiguity. But MemSynth's DISAMBIGUATE query (Section 3.4.4) determines that another weaker memory model, TSO_1, also satisfies all 10 tests, while disagreeing with TSO_0 on a new distinguishing test:

```
(define (hole depth arity non-terms terms)
  ...)  ; Expression hole (Section 3.2.2)

(define (F_Alglave ppo grf fences)
  ...)  ; Axioms from Fig. 3.4

; Common components of memory model specifications
(define (SameAddr X) (& (-> X X) (join loc (~ loc))))
(define rfi (& rf (join thd (~ thd))))
(define rfe (- rf (join thd (~ thd))))

; Expression holes for F_Alglave model (Section 3.3.2)
(define ppo
  (hole 4 2 (list + - -> & SameAddr)
            (list po dep Event Read Write Fence Atomic)))
(define grf (hole 4 2 (list + - -> & SameAddr)
                      (list rf rfi rfe none univ)))
; x86 fences are not cumulative
(define fences (-> none none))

; Final sketch
(define x86-sketch (F_Alglave ppo grf fences))
```

(a) Framework sketch F_{Alglave}

```
; Before disambiguation
(define ppo_0
  (& po (- (-> Event (+ Write Read))
           (-> (- Write Atomic) Read))))
(define grf_0 (- rf (join thd (~ thd))))
(define TSO_0 (F_Alglave ppo_0 grf_0 fences))

; After resolving 4 ambiguities
(define ppo_4 (- po (-> (- Write Atomic) Read)))
(define grf_4 (- rf (join thd (~ thd))))
(define TSO_4 (F_Alglave ppo_4 grf_4 fences))
```

(b) Synthesized models TSO_0 and TSO_4

Figure 3.9: The framework sketch F_{Alglave} for synthesizing a memory model for the x86 architecture (a), and synthesized models TSO_0 and TSO_4 before and after resolving ambiguities (b). The expression holes for ppo and grf define a search space of size 2^{624}, as described in Fig. 3.8. The fences relation is empty because x86 fences are not cumulative.

Test `x86/ambig/1`

Thread 1	**Thread 2**
1: `r1 ← A`	3: `B ← 1`
2: `r2 ← B`	4: `xchg(A, r3)`

Initially: $\texttt{r3} = 1$

Outcome: $\texttt{r1} = 1 \wedge \texttt{r2} = 0$

This test is a variant of the manual's example 8-1 [71], but with an atomic exchange instead of a plain write to `A`. The documentation indicates that x86 should forbid this outcome, as TSO_0 does but TSO_1 does not.

Repeating the ambiguity query after adding `x86/ambig/1` finds 3 more distinguishing tests that further examine the semantics of atomic operations and `mfence`. According to the documentation, the resulting tests should also be forbidden. After adding these tests to the synthesis process, MemSynth is able to prove that a new synthesized model TSO_4 is unique, up to the bounds in Fig. 3.8 on the size of the model specification and distinguishing litmus test.

The TSO_4 model in Fig. 3.9b correctly captures the intent of the x86 TSO model, and is similar to both Fig. 3.3 and Alglave [2]. The synthesized ppo_4 allows writes to be reordered past later reads, while grf_4 allows a thread to read its own writes early. TSO_4 also models the semantics of the `xchg` and `mfence` instructions, both of which are included in ppo_4 and so prevent reordering.[3] We further validated the synthesized model by comparing it to the x86-TSO model of Sewell et al. [119] on the 24 litmus tests in their paper [104]; TSO_4 agrees with x86-TSO on all such tests. The 4 distinguishing tests synthesized by MemSynth are either single-fenced variants of Sewell et al.'s amd5 litmus test, or `xchg`-based variants of other Intel tests, similar to their n8 test.

Potential Redundancy. In the paper on their earlier x86-CC formalization of the x86 memory model, Sarkar et al. [114] write that "P8 may be redundant," where P8 is a principle from the Intel manual about which reorderings are allowed:

> "§8.2.3.9: Loads and stores are not reordered with locked instructions." [71]

The manual section describing this principle includes two litmus tests demonstrating forbidden reorderings. We found that if we omit these two tests from the synthesis process, the ambiguity experiment above re-discovers them, suggesting they are needed to uniquely identify x86's memory model.

3.6.2 *Does MemSynth provide a basis for rapidly building useful automated memory model tools?*

While previous sections build on the F_{Alglave} framework sketch, based on the Alglave et al. [7] framework, MemSynth's engine generalizes to other framework

3 We model Atomic as a subset of Write, so the expression `(- Write Atomic)` allows only plain writes to be reordered.

sketches. In this section, we present F_{MH}, a framework sketch constructed from a framework developed by Mador-Haim, Alur, and Martin [89, 90]. The implementation took only two days of work by this dissertation's author. Moreover, we use MemSynth to automatically rectify a discrepancy between our implementation and the paper's results that we could not resolve by hand.

3.6.2.1 *The F_{MH} Framework*

Mador-Haim, Alur, and Martin's memory model framework [89, 90] was developed to *contrast* memory model specifications by generating a distinguishing litmus test on which two models disagree (as MemSynth's equivalence query does). A memory model is defined by a "must-not-reorder" function $F(x, y)$ that determines whether two instructions x and y can be reordered. The framework places syntactic restrictions on F such that it admits only 90 models. The authors prove that the size of litmus test needed to distinguish models in this set is bounded, and that only 82 of the 90 models are semantically distinct.

3.6.2.2 *Repairing the Framework*

After implementing F_{MH}, we found that our results differed from those in the original paper. The paper states there should be 82 distinct models, but our implementation found only 12 distinct models. Moreover, the paper identifies the following as a distinguishing litmus test (i.e., some models allow it while others forbid it):

Test `mh/L2`

Thread 1	**Thread 2**
1: `X` ⟵ `1`	3: `r1` ⟵ `X`
2: `X` ⟵ `2`	4: `r2` ⟵ `X`

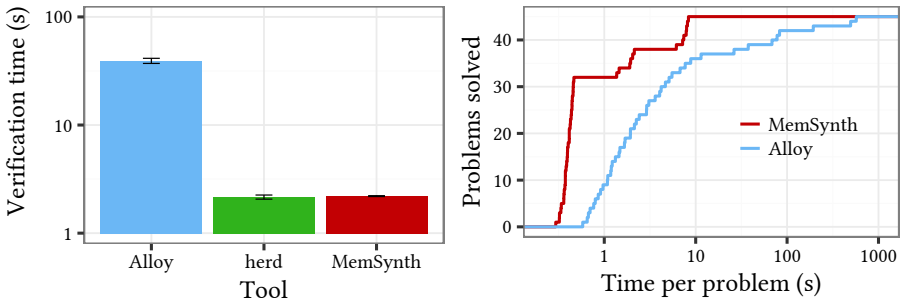
Outcome: $\texttt{r1} = 2 \wedge \texttt{r2} = 0$

Yet our implementation reported this test (which contains a load-load coherence violation allowed by SPARC's RMO model) to be disallowed by all 90 memory models.

Our manual investigation implicated one of the paper's axioms for happens-before relations:

> 5. **Ignore local**: If x is after y in program order, then x cannot happen before y.

Omitting this axiom from our implementation gave 86 distinct models, not 82 as expected, and so we hypothesized that the axiom was necessary but too strong. Since the paper correctly reports that `mh/L2` is allowed by RMO, we believe the paper's results are correct but this axiom was misprinted in the paper. However, the paper's authors were unable to provide their implementation for us to compare against [13].



(a) Verification query

(b) Equivalence query

Figure 3.10: Performance comparisons between MemSynth and existing tools for (a) verification and (b) equivalence.

We first tried to fix the axiom by hand, but despite several attempts, a correct fix eluded us: our closest results identified 78 or 86 distinct models rather than 82. Instead, we used MemSynth's relational logic DSL to synthesize a repair. In relational logic, axiom 5 is written as "no $(\sim\mathsf{po})$ & hb", where hb is the happens-before relation for an execution. To repair the axiom, we replaced ~po with an expression hole of depth 3, and synthesized a completion that gave the correct outcomes on the 9 litmus tests from the original paper on both TSO and RMO memory models. We were able to synthesize the following repair in 4 seconds:

$$\text{no } (\sim((\mathsf{po} - \mathsf{rf}) \,\&\, (\mathsf{Write} \to \mathsf{Read}))) \,\&\, \mathsf{hb}$$

In prose:

> 5a. **Ignore local**: If x is after y in program order, x is a read, y is a write, and x does not read the value written by y, then x cannot happen before y.

The repaired axiom allows reads to see local writes early without affecting the happens-before relation. We believe it is intended to allow models such as TSO to observe their own writes early by ignoring the happens-before order. With the repaired axiom, our pairwise comparison results produce 82 distinct models, identical to the original paper.

3.6.3 *Does MemSynth outperform existing relational solvers and memory model tools?*

This section compares MemSynth to existing relational engines and memory model tools on verification, equivalence, and synthesis queries.

Verification. Fig. 3.10a shows the time for MemSynth, Alloy (`v4.2_2015-02-22`) [72, 128], and herd (v7.43) [9] to verify 768 PowerPC litmus tests from Section 3.6.1. The Alloy results use the PPC_1 specification synthesized by MemSynth, while herd (configured in "speed check" mode) already supports PowerPC. The results show that MemSynth outperforms Alloy by 10×, and is comparable to herd's custom decision procedure for memory models.

Equivalence. We used MemSynth and Alloy* (v0.2) [99] to perform a pairwise comparison of 10 different synthesized PowerPC models. Both MemSynth and Alloy* used a symbolic litmus test with up to 2 threads and 6 instructions. Fig. 3.10b shows that MemSynth outperforms Alloy* on most of these queries: MemSynth can solve 3× more queries in under one second, and the hardest problem takes 8 s for MemSynth versus 10 min for Alloy*. With symmetry breaking and concretization (which cause the large steps in the MemSynth line in Fig. 3.10b) disabled, MemSynth could not solve any of the comparisons in under an hour.

Synthesis. The synthesis query (Section 3.4.2) requires higher-order quantification, and so we compared MemSynth to Alloy* [99]. Because Alloy* does not support expression holes (Section 3.2), we designed a framework sketch $\mathcal{M}$ that simply chooses between hard-coded memory models. When given $\mathcal{M} = \{\text{SC}, \text{TSO}\}$, both MemSynth and Alloy* return in under a second. However, when given $\mathcal{M} = \{\text{SC}, \text{TSO}, \text{PSO}\}$, MemSynth still returns in under a second, but Alloy* times out after one hour. This result suggests Alloy* would not be able to synthesize models from complex framework sketches.

3.7 RELATED WORK

MemSynth is, to our knowledge, the first tool to provide synthesis and other higher-order queries for memory model specifications. It builds on existing work in formalizing and reasoning about memory models, which this section reviews.

Formalization. Few architectures formalize their memory models (with the exception of SPARC [134] and Alpha [48]), and so this task has fallen to researchers. A notable success is the x86-TSO model [119], which formalizes the memory model of the x86 architecture. This model was refined through several papers [103, 114], which revealed ambiguities in the x86 documentation. In Section 3.6.1.2, MemSynth's DISAMBIGUATE query automatically identified more such ambiguities.

Another effort has developed several formalizations of the PowerPC architecture [5, 7, 9, 91, 113]. The PowerPC memory model allows many more reorderings than x86, and features cumulative barriers to restore stronger behavior. The specification for PowerPC is complex, and several ambiguities in the PowerPC manual [69] required detailed experimentation to resolve. The PowerPC formalization effort also developed a suite of memory model experimentation tools, which we use in Section 3.6.1 and Section 3.6.3.

Formalization efforts have also brought clarity to emerging programming language memory models, particularly C11 and C++11 [19, 20]. These efforts have helped check that the target models provide basic guarantees about important classes of programs—for example, that all data-race-free programs have sequentially consistent memory ordering [1]. Like hardware memory models, language memory models are also relational, and some (e.g., the Java Memory Model [92]) have already been formalized [129] in bounded relational logic. We therefore be-

lieve MemSynth could also be effective for language models, with appropriate design of a framework sketch.

Frameworks. Recent work has developed generic memory model frameworks that can be instantiated with different architectures. The Nemos framework [139] offers axiomatic specifications for a variety of models, such as causal consistency, but (to our knowledge) cannot express microprocessor models such as TSO. Alglave, Maranget, and Tautschnig [2, 7, 9] developed an axiomatic framework for microprocessor memory models. It admits models for complex architectures such as PowerPC, and is the basis for our F_{Alglave} framework sketch (Section 3.3.2) and most experiments in Section 3.6. Mador-Haim, Alur, and Martin [90] developed a framework for store-atomic memory models, which we implement in Section 3.6.2. It captures common models such as TSO, but is restricted enough to prove upper bounds on the size of distinguishing litmus tests.

Automated Reasoning. One common application of formal memory models is inserting synchronization instructions that restore sequential consistency in a concurrent program. Alglave et al. [7] address this problem for PowerPC with a specification of the platform's barrier semantics, including cumulativity; we automatically synthesize this specification in Section 3.6.1. Another common application is verification of concurrent code under relaxed memory models, and several tools have been developed for this purpose (e.g., [52, 54]). All of them rely on formal specifications of memory models that can be synthesized with MemSynth.

MemSAT [129] is an automated tool that implements the verification query of Section 3.4.1 for axiomatic memory model specifications. MemSAT found several discrepancies in the formalization of the Java Memory Model [92]. MemSynth is similar to MemSAT in its use of relational logic, but focuses on hardware models and offers richer automated reasoning queries including synthesis. Wickerson et al. [136] use Alloy* [99] to implement a tool for automatically comparing memory consistency models, similar to MemSynth's equivalence query. They show results for both processor and language memory models, but their tool does not support MemSynth's synthesis and ambiguity queries, and it is unclear how to adapt their quantifier elimination strategy ("deadness") to specification synthesis. Lustig et al. [88] use Alloy [72] to synthesize suites of litmus tests that examine a set of predefined memory ordering relaxations, which together compose a design space we could use as a framework sketch.

3.8 CONCLUSION

MemSynth is a synthesis-aided system for reasoning about axiomatic specifications of memory consistency models. As opposed to existing memory model tools that perform verification of hand-written models, MemSynth can synthesize memory model formalizations based on a framework sketch provided by a designer. MemSynth's expressive specification language builds on an optimized bounded relational logic engine, which serves as a platform for developing novel automated reasoning queries. We showed that MemSynth can synthesize specifi-

cations for complex architectures, refine those specifications by identifying ambiguities, and support rapid development of memory model tools that outperform hand-crafted versions. As new parallel architectures continue to emerge, MemSynth can help formalize their memory models rapidly and precisely.

4

METASKETCHES

Many advanced programming tools—for both end-users and expert developers—rely on program synthesis to automatically generate implementations from high-level specifications. These tools often need to employ tricky, custom-built synthesis algorithms because they require synthesized programs to be not only correct, but also *optimal* with respect to a desired cost metric, such as program size. Finding these optimal solutions efficiently requires domain-specific search strategies, but existing synthesizers hard-code the strategy, making them difficult to reuse.

This chapter presents *metasketches*, a general framework for specifying and solving optimal synthesis problems.[1] Metasketches make the search strategy a part of the problem definition by specifying a fragmentation of the search space into an ordered set of classic sketches. Two search algorithms cooperate to effectively solve metasketches. A global optimizing search coordinates the activities of local searches, informing them of the costs of potentially-optimal solutions as they explore different regions of the candidate space in parallel. The local searches execute an incremental form of counterexample-guided inductive synthesis to incorporate information sent from the global search. Synapse is an implementation of these algorithms that effectively solves optimal synthesis problems with a variety of different cost functions. In addition, metasketches can be used to accelerate classic (non-optimal) synthesis by explicitly controlling the search strategy, and Synapse solves classic synthesis problems that state-of-the-art tools cannot. Metasketches have been used to accelerate several synthesis tools, including for chemical reaction networks [39] and education [33].

4.1 OVERVIEW

Program synthesis is the classic problem of automatically producing an implementation from a high-level correctness specification. Recent research efforts have addressed this problem successfully for a variety of application domains, from browser layout [97] to executable biology [79]. But for many applications, such as synthesis-aided compilation [107, 115] or end-user programming [59, 66], it is not enough to produce *any* correct program. These applications require the synthesized implementation to also be *optimal* with respect to a desired cost function—for example, the number of instructions or the sum of their latencies.

Optimal synthesis involves producing a program that is both correct with respect to a (logical) specification and optimal with respect to a cost function. Existing tools for optimal synthesis are highly specialized, employing custom search strategies to quickly find the best solution in a large space of candidate programs. For example, a superoptimizer [75, 115] finds the least expensive instruction se-

1 This chapter was first published as the paper *Optimizing Synthesis with Metasketches*, by James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze, at POPL 2016 [27].

quence (according to a cost model) equivalent to a given reference implementation. To make this task tractable, a superoptimizer must be able to focus its search on candidate programs cheaper than the currently-optimal solution. Building such a tool on top of existing synthesizers is impractical, because they provide no means for the tool to guide or control the search strategy. Instead, tool developers are forced to implement their own synthesis engines from scratch, giving up the potential to benefit from advances in general synthesis technology.

Metasketches are a general framework for specifying and solving optimal synthesis problems. A metasketch is an ordered set of *sketches* [122], together with a *cost function* to minimize and a *gradient function* to direct the search. A sketch is a syntactic template that defines a finite space of candidate programs. The union of the (possibly overlapping) sketches describes the candidate space of the metasketch. The ordered set of sketches, together with the cost and gradient functions, expresses a high-level search strategy: the sketches fragment the candidate space into regions that can be explored in parallel, while the sketch ordering and the two functions focus the search toward cheaper regions of the space. Because a metasketch consists of a set of classic sketches, solving a metasketch reduces to solving a set of classic synthesis problems, and so tools built with metasketches benefit from progress in the underlying synthesis techniques.

To solve an optimal synthesis problem expressed as a metasketch, we employ two cooperating algorithms: a global optimizing search over the entire candidate space, and many parallel instances of a local combinatorial search over the individual sketches in the metasketch. The local search algorithm implements an incremental form of counterexample-guided inductive synthesis (CEGIS). The global search drives this incremental local exploration in two ways. First, it uses the gradient function to select which sketches to explore locally whenever a satisfying solution (and therefore a tighter upper bound on the cost) is found. The gradient function is simple: given a numerical cost, it returns the set of all sketches from the metasketch that (may) contain a cheaper candidate program. Second, the global search communicates the cost of discovered solutions to all running local searches, which integrate these results to prune their own candidate spaces. The search process proceeds until an optimal solution is found or the global search space is exhausted. This search strategy is highly effective, solving both classic and optimal synthesis problems that cannot be solved by existing techniques.

In addition to enabling efficient search, metasketches also bring new expressive power to syntax-guided synthesis. By representing the search space as a set of sketches, a metasketch can describe candidate spaces such as "the set of all programs, of any length, that are in static single assignment (SSA) form and that contain no unused variables." A space of this form cannot be expressed with a single sketch (because it is infinite) nor can it be expressed with a context-free grammar (because neither the SSA nor the used-variable constraints are context-free). A set-of-sketches space description also supports an effective new form of encoding optimization, which we call *structure constraints*. These constraints take the form of assertions within an individual sketch, which rule out candidates that are semantically equivalent to cheaper programs from other sketches. The resulting

metasketch avoids redundant work in the local searches, thus accelerating the global synthesis process.

We have implemented our optimal synthesis approach in a tool called SYNAPSE, built on top of the ROSETTE language [126, 127]. We have used SYNAPSE to develop and solve metasketches for a variety of optimal synthesis problems, from superoptimization to fixed-point approximation of computational kernels. Our experiments show that SYNAPSE is not only effective at solving optimal synthesis problems, but that it can also solve standard synthesis benchmarks [10] that are intractable for state-of-the-art syntax-guided synthesizers (due to their large, monolithic search spaces that we fragment with metasketches). The fragmented search space exposed by a metasketch also allows SYNAPSE to realize significant parallel speedup for large synthesis problems. We find that for many optimal synthesis problems, the search algorithm spends most of its time *proving* optimality of the final candidate solution, suggesting that the search can quickly output intermediate results that will *likely* be optimal. Finally, we show that SYNAPSE can effectively reason about a variety of cost functions through several examples: fitting a linear model to a training data set, optimizing worst-case execution time of a program, and training a small neural network.

4.2 OPTIMAL SYNTAX-GUIDED SYNTHESIS

This section briefly reviews syntax-guided synthesis [11, 122] and formalizes the problem of optimal syntax-guided synthesis. We also introduce a small Scheme-like synthesis language, SYN, that will be used to present (optimal) synthesis examples throughout this chapter. Our approach is independent of SYN, however, and can be applied to any language that supports basic sketching constructs (such as Sketch [122] or ROSETTE [126]).

Synthesis. The program synthesis problem is to automatically discover a program P that implements a desired specification φ. Programs are written in a language $\mathcal{L}$, and specifications in a decidable theory $\mathcal{T}$ (or a decidable combination of theories $\mathcal{T}_i$). We assign a deterministic semantics $[\![P]\!]$ to each program $P \in \mathcal{L}$. For a set of programs $\mathcal{S} \subseteq \mathcal{L}$, we write $[\![\mathcal{S}]\!]$ to denote the set $\{[\![P]\!] \mid P \in \mathcal{S}\}$. A specification is a formula $\varphi(x, [\![P]\!](x))$ in the theory $\mathcal{T}$ that relates program inputs to outputs. Given a specification φ, the program synthesis task is to find a program $P \in \mathcal{L}$ such that the formula $\forall x.\, \varphi(x, [\![P]\!](x))$ is valid modulo $\mathcal{T}$.

Programs and Specifications. For examples in this chapter, we take the language $\mathcal{L}$ to be SYN, a subset of core Scheme [111] shown in Figure 4.1. SYN expressions are constructed from booleans, signed finite-precision integers, lambda terms, applications, conditionals, and sequential let-binding expressions. The language also includes the usual built-in procedures for operating on booleans and integers. We take the specification theory $\mathcal{T}$ to be the quantifier-free theory of fixed-width bitvectors. For convenience, specifications can be expressed as assertions in SYN

expressions $e ::= l \mid x \mid$ (**lambda** (x …) e) $\mid$ (e e …) $\mid$
(**if** e e e) $\mid$ (**let*** ([x e] …) e) $\mid$ (**assert** e)
$l ::=$ **true** $\mid$ **false** $\mid$ integer literal
$x ::=$ identifier $\mid$ = $\mid$ > $\mid$ + $\mid$ - $\mid$ * $\mid$ / $\mid$ & $\mid$ …
definitions $d ::=$ (**define** x e)
forms $f ::= d \mid e$
programs $p := f \mid p\,f$

Figure 4.1: Syntax of programs in the simple Scheme-like language Syn we use for examples.

programs in the standard manner.[2] An assertion succeeds if the value of the expression argument is not **false**. The semantics of Syn is standard [111], except that all built-in integer operators also accept boolean arguments, treating **true** as 1 and **false** as 0.

Example 1. Suppose that we are trying to synthesize a Syn implementation of the max function. In the theory of bitvectors, the specification for max is straightforward:

$$\varphi_{\max}(\langle x, y\rangle, [\![P]\!](x,y)) \equiv [\![P]\!](x,y) = \mathrm{ite}(x > y, x, y)$$

There are many programs in Syn that meet this specification, such as the following SSA-style implementation:[3]

```
(define (max1 i1 i2)
  (let* ([o1 (> i1 i2)]
         [o2 (if o1 i1 i2)])
    o2))
```

A program synthesizer should return `max1` or another correct implementation from Syn.

Syntax-Guided Synthesis. Syntax-guided synthesis is a form of program synthesis that restricts the search for P to a space of candidate implementations $\mathcal{C} \subseteq \mathcal{L}$ defined by a syntactic template [11]. This restriction makes the search more tractable, and it enables the programmer to describe the desired implementation using a mix of syntactic and semantic constraints.

Syntactic constraints commonly take the form of a context-free grammar [11] or a *sketch* [122]. A sketch is a partial implementation of a program, with missing expressions called *holes* to be discovered by the synthesizer. Holes are constrained to admit expressions from a finite set of choices—for example, a hole could be replaced with a 32-bit integer constant or with an expression obtained from a finite unrolling of a context-free grammar. Unlike context-free grammars, sketches can

2 In particular, Syn assertions can be reduced to formulas in the theory of bitvectors for all finite Syn programs, using existing methods [126].

3 We write (**define** (x y …) e) to abbreviate (**define** x (**lambda** (y …) e)).

express only finite candidate spaces $\mathcal{C}$. In return, however, they provide the programmer with more control over the shape of the search space, as well as the ability to express syntactic constraints that are not context-free.

Sketches. To enable sketching in SYN, we add a hole construct:

$$\text{expressions } e ::= \ldots \mid (\texttt{??}\ e \ldots)$$

The hole construct can be used in one of two ways. When it is applied to no expressions, (`??`), it represents a placeholder for an integer constant. Otherwise, it is a placeholder that selects from among the provided expressions.

We call a program in SYN a sketch if it contains holes. A sketch $S \in \mathcal{L}$ defines a set of candidate programs $\overline{S}$, which is the set of all possible programs produced by replacing the holes H in P with concrete expressions. We can define the synthesis problem in terms of completing the holes: given a sketch S, the program synthesis task is to find a completion $\vec{h}$ for the holes H in S such that the formula $\forall x.\, \varphi(x, [\![S[H := \vec{h}]]\!](x))$ is valid modulo $\mathcal{T}$. We abuse notation to write $[\![S]\!]$ for the set of semantics $[\![\overline{S}]\!]$ of all possible programs produced by a sketch.

Example 2. Sketches allow programmers to capture domain insights that can make synthesis more tractable. For example, a SYN sketch for `max` might specify that the last operation is always an **`if`**:

```
(define (max1-sketch i1 i2)
  (let* ([o1 ((?? > >= = < <=) (?? i1 i2) (?? i1 i2))]
         [o2 (if o1 i1 i2)])
    o2))
```

The advantage of a sketch is that the synthesizer need only discover an assignment to the holes that satisfies the specification φ, without having to explore all possible programs in SYN.

Optimal Syntax-Guided Synthesis. Even with syntactic constraints, there is rarely a unique solution to a given synthesis problem. Our simple `max1-sketch`, for example, has four correct solutions, and the synthesizer is free to return any one of them. But for many applications, some solutions are more desirable than others due to requirements such as program size, execution time, or memory or register usage. For these applications, the synthesis task becomes one of optimization rather than search.

We define the *optimal syntax-guided synthesis* problem as a generalization of syntax-guided synthesis. The optimal program synthesis problem is the task of searching a space of candidate programs $\mathcal{C}$ for a lowest-cost implementation P that satisfies the given specification φ. The search is performed with respect to a cost function κ, which assigns a numeric cost to each program $P \in \mathcal{L}$.

Definition 4.1 (Optimal Syntax-Guided Synthesis)**.** *Let $\mathcal{L}$ be a programming language, and $\mathcal{T}$ a decidable theory. Given a specification formula $\varphi(x, [\![P]\!](x))$ in $\mathcal{T}$, a cost function $\kappa : \mathcal{L} \to \mathbb{R}$, and a search space $\mathcal{C} \subseteq \mathcal{L}$ of candidate programs, the* optimal (syntax-guided) synthesis *problem is to find a program $P \in \mathcal{C}$ such that the formula $\forall x.\, \varphi(x, [\![P]\!](x))$ is valid modulo $\mathcal{T}$, and $\kappa(P)$ is minimal among all such programs.*

Note that when the cost function κ is constant, optimal synthesis reduces to syntax-guided synthesis.

To ensure that the optimal synthesis problem remains decidable, we must place restrictions on the cost function κ. Existing optimal synthesis techniques often require κ to reason only about program syntax. For our synthesis approach (Section 4.4), it is sufficient to require that the evaluation of κ on a program P be reducible to a term in a decidable theory $\mathcal{T}$. This allows us to encode cost functions that reason not only about program syntax but also about program semantics. For example, in Section 4.5.6, we demonstrate a simplified worst-case execution time metasketch, which reasons about feasible paths through the program's control flow.

Example 3. Optimal synthesis chooses among multiple correct candidate programs by minimizing a given cost function. Different cost functions will produce different optimal solutions. For example, suppose we want to find an implementation $P \in \textsc{Syn}$ of our $\varphi_{\max}$ specification that minimizes the sum of operation costs:

programs	$\kappa(p)$	$= \sum_{f \in p} \kappa(f)$
definitions	$\kappa\,((\textbf{define}\ x\ e))$	$= \kappa(e)$
expressions	$\kappa\,((\textbf{if}\ e_1\ e_2\ e_3))$	$= 1 + \kappa(e_1) + \kappa(e_2) + \kappa(e_3)$
	$\kappa\,((\textbf{let*}\ ([x_i e_i] \ldots)\ e))$	$= \kappa(e) + \sum_i \kappa(e_i)$
	$\kappa\,((\textbf{lambda}\ (x \ldots)\ e))$	$= \kappa(e)$
	$\kappa\,((\textbf{assert}\ e))$	$= 0$
	$\kappa\,((e\ e_i \ldots))$	$= \kappa(e) + \sum_i \kappa(e_i)$
	$\kappa\,(x)$	$= 1$ if x is a built-in operator
		$= 0$ otherwise

The program `max1` from Example 1 has a cost of 2, and it is an optimal solution under the cost function κ. However, suppose that we are targeting an environment where branches are expensive and to be avoided. We can update the cost function to penalize branches as follows:

$$\text{expressions} \quad \kappa\,((\textbf{if}\ e_1\ e_2\ e_3)) = 8 + \kappa(e_1) + \kappa(e_2) + \kappa(e_3)$$

Under this cost function, the `max1` solution has a cost of 9. The new optimal solution has a cost of 4 and implements an arithmetic manipulation for the maximum of two (finite precision) integers:

```
(define (max2 i1 i2)
  (let* ([o1 (- i2 i1)]
         [o2 (<= i1 i2)]
         [o3 (* o1 o2)]
         [o4 (+ i1 o3)])
    o4))
```

Given $\varphi_{\max}$, $\mathcal{C} = \textsc{Syn}$, and our new cost function, an optimal synthesizer should return `max2`, or another correct program with the same cost as `max2`.

4.3 METASKETCHES

This section introduces *metasketches* (Def. 4.2), a new abstraction for specifying and solving optimal synthesis problems. Metasketches generalize sketches, enabling a description of an infinite space of candidate programs with a countable, ordered set of finite sketches. This representation permits fine-grained control over the shape of the candidate space, which is critical for effective search. A metasketch additionally provides a means of assigning cost to programs and of directing the search toward lower-cost regions of the candidate space. This section defines metasketches, describes their properties, and illustrates their utility for capturing insights that enable efficient search. Section 4.4 presents a synthesis algorithm that exploits the search strategy exposed by metasketches.

4.3.1 *The Metasketch Abstraction*

A metasketch consists of three components: (1) a *space* of candidate programs, represented as a countable, ordered set of finite sketches; (2) a *cost* function from programs to numeric cost values; and (3) a *gradient* function from each cost value c to a set of sketches (i.e., a subspace) that may contain a program with a lower cost than c. The space component provides a way to fragment a monolithic search space into a set of finite regions that can be explored independently of one another. Because each region is described by a sketch, the programmer gains the ability (as we show later) to use context-sensitive *structure constraints* to reduce overlap between the regions—thus making both the global and local search tasks easier. The cost and gradient functions provide a way to navigate the local and global search spaces in a cost-sensitive way. Together, these components enable the programmer to easily convey key problem-specific insights to a generic search algorithm.

Definition 4.2 (Metasketch)**.** *A* metasketch *is a tuple* $m = \langle \mathcal{S}, \kappa, g \rangle$*, where:*

- *The* space $\mathcal{S} \subseteq \mathcal{L}$ *is a countable set of sketches in* $\mathcal{L}$*, equipped with a total ordering relation* $\preceq$*.*
- *The* cost function $\kappa : \mathcal{L} \to \mathbb{R}$ *assigns a cost to each program in the language* $\mathcal{L}$*.*
- *The* gradient function $g : \mathbb{R} \to 2^{\mathcal{S}}$ *returns an overapproximation of the set of sketches in* $\mathcal{S}$ *that contain programs with lower cost than a given value* $c \in \mathbb{R}$*:*

$$g(c) \supseteq \{S \in \mathcal{S} \mid \exists \vec{h}.\, \kappa(S[H := \vec{h}]) < c\} \tag{2}$$

Given a specification φ, a metasketch $m = \langle \mathcal{S}, \kappa, g \rangle$ defines an instance of the optimal program synthesis problem (Def. 4.1), in which the search space $\mathcal{C}$ is the union $\bigcup_{S \in \mathcal{S}} \overline{S}$ of the search spaces of each sketch in the set $\mathcal{S}$, and the cost function is given by κ.

4.3.2 *Properties of Metasketches*

Metasketches bring new expressive power to (optimal) syntax-guided synthesis in two ways. First, they can express richer candidate spaces than either sketches or context-free grammars alone: unlike a classic sketch, a metasketch can capture an infinite space of candidate programs, and unlike a context-free grammar, it can express syntactic constraints on the search space that are context-sensitive (see Section 4.3.3 for examples). Second, unlike other forms of syntactic templates, metasketches describe both a search space and a *search strategy*.

In particular, by specifying the candidate space as an ordered set of sketches, the programmer provides a decomposition of the problem into independent parts, as well as an order in which those parts should be explored. At one extreme, if each sketch $S \in \mathcal{S}$ is a concrete program with no holes, the metasketch is an implementation of brute force search. Ordering these programs according to AST depth yields a bottom-up brute force search, a common synthesis technique [10, 130]. At the other extreme, if $\mathcal{S}$ contains only a single finite sketch, the programmer is choosing to solve the problem monolithically with the underlying synthesizer's search strategy, such as reduction to SMT [122, 127]. But there are many other search strategies between these two extremes that are also easily captured with a metasketch—for example, adaptive concretization [73] randomly replaces some holes in a sketch with concrete values, thus creating a family of sketches of roughly the same complexity (as measured by the number of holes) that are solved independently.

While the space component of a metasketch expresses a search strategy, the gradient function expresses an *optimization strategy*—essentially, a cost-based filter for the optimal synthesizer to apply to the global search space once it finds some solution. For commonly used cost functions (such as program length or the sum of instruction latencies), it is easy to provide a function that precisely determines whether a finite sketch contains a program with a cost lower than a given value. Of course, this may be difficult or impossible for more complex cost functions. For this reason, a metasketch only requires g to be an overapproximation (Equation 2): it can return sketches that have no solution with cost less than c, but to be sound, must not filter out a sketch that does have such a solution. As a degenerate case, the gradient function $g(c) = \mathcal{S}$ is a trivial overapproximation that filters out no sketches. Using a trivial gradient will not affect the correctness of the search, nor will it affect its optimality over finite spaces, but as we discuss in Section 4.4, a more constrained gradient is required to guarantee optimality over infinite spaces.

4.3.3 *Examples*

We illustrate the process of creating metasketches for two optimal synthesis problems: superoptimization [68, 75, 94, 115, 133] and approximate computing [38, 57, 100]. Superoptimization is the problem of finding the optimal sequence of instructions that implements a given specification. Approximate computing allows small calculation errors in programs (such as image processing kernels) that result

$$
\begin{aligned}
\mathcal{S} &= \{S_i \mid i \in \mathbb{N}^+\} \\
\preceq &= \{\langle S_i, S_j\rangle \mid i \le j\} \\
\kappa(P) &= i \text{ for } P \in S_i \\
g(c) &= \{S_i \mid i < c\}
\end{aligned}
$$

$$
\begin{aligned}
S_i = \ & \texttt{(lambda (}x \ldots\texttt{)} \\
& \quad \texttt{(let* ([o}_1 \texttt{ (}expr\ x \ldots\texttt{)]} \\
& \qquad \ldots \\
& \qquad \texttt{[o}_i \texttt{ (}expr\ x \ldots\ \texttt{o}_1 \ \ldots\ \texttt{o}_{i-1}\texttt{)])} \\
& \quad \texttt{o}_i\texttt{))}
\end{aligned}
$$

$$
\begin{aligned}
(expr\ e \ldots) = \ & \texttt{(?? ((?? - ~) (?? } e \ldots\texttt{))} \\
& \quad \texttt{((?? + - * \& ...) (?? } e \ldots\texttt{) (?? } e \ldots\texttt{))} \\
& \quad \texttt{(if (?? } e \ldots\texttt{) (?? } e \ldots\texttt{) (?? } e \ldots\texttt{)))}
\end{aligned}
$$

Figure 4.2: A basic metasketch for superoptimization. This formulation defines the search space to consist of all Syn programs in SSA form. The sketches are ordered according to size. The cost function κ : Syn $\to \mathbb{N}$ measures the number of conditionals and applications of built-in operators.

in lower energy expenditure or execution time. A common formulation of the approximate computing task boils down to an optimal synthesis problem, in which the cost function encodes the desired performance metric, and the specification constrains the synthesized program to be sufficiently accurate with respect to the reference program. We show three simple metasketches for these applications. Despite their simplicity, however, our metasketches capture enough insight to enable optimal synthesis of superoptimization and approximation benchmarks that cannot be solved—even ignoring optimality—with existing techniques (Section 4.5).

Superoptimization Figure 4.2 shows a basic superoptimization metasketch, designed to find a shortest program in SSA form using a given set of operators (in our case, all built-in Syn operators). The space of all SSA programs cannot be expressed with a context-free grammar, since SSA constraints are context-sensitive. However, it is easily expressed as a set of sketches. Our metasketch assumes a cost function κ that measures the number of conditionals and applications of built-in operators (that is, the original cost function from Example 3). The search space consists of all finite SSA sketches S_i with i defined variables (whose names are canonical), and the sketches are ordered according to how many defined variables they contain. For our cost function, the number of defined variables in a sketch completely determines the cost of all programs in that sketch, which is reflected in the gradient function g.

The metasketch in Figure 4.2 encodes a simple iterative deepening strategy for superoptimization, in which smaller search spaces (corresponding to shorter programs) are explored first. However, this encoding is inefficient: a sketch of size i includes programs that can be trivially reduced to a shorter program by dead code elimination. As a result, any search over the space defined by the sketch S_i will

explore programs that are also covered by sketches $S_j \preceq S_i$. To reduce overlap between sketches, we can amend our encoding of S_i to force all defined variables to be used at least once. To do so, we simply lift the choice of the k^{th} variable's input arguments into fresh variables $\vec{v_k}$, and add, for each o_j, the following assertion to the body of the `let*` expression: $\bigvee_{k>j, v \in \vec{v_k}} o_j = v$. We call such assertions (that only constrain the holes) *structure constraints*. In this case, the structure constraints remove from a sketch of length i (a large class of) programs that are also found in shorter sketches. We have used the resulting metasketch to solve existing synthesis benchmarks orders-of-magnitude faster than other symbolic synthesis techniques, and in many cases as fast as the winner of the 2014 syntax-guided synthesis competition [10].

Adaptive Superoptimization Superoptimization (e.g., [75, 115]) often involves richer cost functions than program length. For example, we may want to use a static cost model of operator latencies, defined by modifying the cost function κ from Example 3 to assign different weights to built-in operators and to conditionals:

$$\kappa\,((\textbf{if}\ e_1\ e_2\ e_3)) = c_{if} + \kappa(e_1) + \kappa(e_2) + \kappa(e_3)$$

$$\kappa\,(x) \qquad\qquad = c_x \text{ if } x \text{ is a built-in operator}$$

To obtain a gradient for this cost function, we simply change g from Figure 4.2 to $g(c) = \{S_i \mid i * c_{\min} < c\}$, where $c_{\min}$ is the lowest operator cost according to κ. The new metasketch continues to encode length-based iterative deepening, but given its cost function, we can refine the search strategy by adding a second dimension to the sketches: the set of operators that may appear in the program.

In particular, to bias the search toward exploring cheaper sub-spaces first, we sort the available operators according to cost, and then create a set of sketches $S_{i,j}$ where i is the length of the sketch, as in Figure 4.2, and j specifies the prefix of the sorted operators used to build expressions. The ordering on sketches now becomes the lexicographical order over $\langle i, j \rangle$ (although other orders are possible as well), and we add one more structure constraint to $S_{i,j}$ that forces the j^{th} operator to define at least one of the i variables, thus reducing overlap between sketches along the instruction-set dimension. The resulting adaptive superoptimization metasketch enables us to find optimal, fast approximations of image processing kernels that cannot be found tractably with other approximation techniques.

Piecewise Polynomial Approximation Typical targets for approximate computing include small computational kernels that are invoked many times by an outer loop. These kernels often perform expensive floating-point arithmetic using transcendental functions. For example, the following C code shows a kernel that computes the inverse kinematics of a robotic arm with two joints:

```
void inversek2j(float x, float y, float* th1, float* th2) {
  *th2 = acos(((x*x) + (y*y) - 0.5) / 0.5);
  *th1 = asin((y * (0.5 + 0.5*cos(*th2)) - 0.5*x*sin(*th2))
               / (x*x + y*y));
}
```

Existing techniques [57] for approximating kernels such as `inversek2j` rely on hardware-accelerated neural networks, limiting their usability by requiring custom-

$$
\begin{aligned}
\mathcal{S} &= \{S_{k,n} \mid k \in \mathbb{N}^+, n \in \mathbb{N}\} \\
\preceq &= \{\langle S_{k,n}, S_{u,v}\rangle \mid k < u \vee (k = u \wedge n \leq v)\} \\
\kappa(P) &= a * k + b * n \text{ for } P \in S_{k,n} \\
g(c) &= \{S_{k,n} \mid a * k + b * n < c\}
\end{aligned}
$$

$S_{1,n}$ =
```
(lambda (x_1 ... x_m)
  (poly_n x_1 ... x_m))
```

$S_{k>1,n}$ =
```
(lambda (x_1 ... x_m)
  (if (bnd x_1 ... x_m)
      (poly_n x_1 ... x_m) ; 1st piece
      (…
        (if (bnd x_1 ... x_m)
            (poly_n x_1 ... x_m)
            (poly_n x_1 ... x_m) ; kth piece
            )…)))
```

$(bnd\ x_1 \ldots x_m)$ =
```
(and (< x_1 (??)) ... (< x_n (??)))
```

$(poly_n\ x_1 \ldots x_m)$ =
```
(+ (* (??) (expt x_1 n)) ...
   (* (??) (expt x_m n)) ...
   (* (??) (expt x_1 1)) ...
   (* (??) (expt x_m 1)) ...
   (??))
```

Figure 4.3: A basic metasketch for piecewise polynomial approximation. The search space consists of all SYN programs that implement a piecewise polynomial function with k pieces and the maximum degree of n. The sketches are ordered lexicographically by $\langle k, n\rangle$. The cost function κ : SYN $\to \mathbb{Z}$ is a linear combination of k and n.

designed hardware. We present a metasketch that implements the first software-based technique for approximating these kernels successfully, using only conditionals and fixed-point addition and multiplication.

Our metasketch, shown in Figure 4.3, implements a piecewise polynomial approximation of a mathematical function. The candidate space is decomposed along two dimensions: the number k of pieces and the degree n of each polynomial. The sketch $S_{k,n}$ contains $k - 1$ unknown branching conditions defining k pieces, and each branch contains a polynomial of degree n with unknown coefficients. Because our optimal synthesis problem involves approximating a function over a set of points sampled from its domain, the cost κ minimizes a linear combination of pieces and degree, in order to prevent overfitting to the sample set.

As in the case of (adaptive) superoptimization, we can reduce overlap between the sketches in Figure 4.3 with structure constraints. In particular, we force some piece in every $S_{k,n}$ to include an n^{th} degree term with a non-zero coefficient, and we force the branching conditions to differ in at least one term. In other words, our constraints prevent sketches of size $\langle k, n\rangle$ from containing (a class of) programs that belong to smaller sketches after constant propagation and dead code elimination. Finally, we also break symmetries in the $S_{k,n}$ search space by ordering the constants in the branching conditions, so that the i^{th} bound for the ar-

gument x_i is no greater than its $i + 1^{\text{st}}$ bound. Our optimal synthesis algorithm solves the resulting metasketch for several standard approximation benchmarks, including `inversek2j`, for which it finds an approximation with 16% error and 35× speedup.

4.4 OPTIMAL SYNTHESIS ALGORITHM

Our synthesis approach takes advantage of the metasketch abstraction by layering a global search atop individual local searches running in parallel. The global search executes the high-level strategy encoded in a metasketch, and coordinates the activities of local searches to satisfy the optimality requirement. Local searches execute an incremental form of counterexample-guided inductive synthesis (CEGIS) [121], which can accept additional constraints during the inductive synthesis loop. This section presents the global and local search algorithms, characterizes their properties, and describes performance-oriented implementation details.

4.4.1 *Global Search*

To solve a metasketch $m = \langle \mathcal{S}, \kappa, g \rangle$, the global search coordinates individual solvers operating on sketches drawn from the space $\mathcal{S}$. This coordination takes two forms. First, the global search uses the ordered set $\mathcal{S}$ and the gradient function g to select which sketches to send to individual solvers. The total order $\preceq$ on $\mathcal{S}$ defines the order in which to search sketches; the search order can significantly change the performance of the synthesis procedure, as Section 4.4.4 discusses. The gradient function g filters $\mathcal{S}$ once a satisfying solution is found to only search sketches with potentially cheaper solutions. Second, the global search receives candidate solutions from the individual solvers as they execute. The global search broadcasts information about these candidates to all local solvers, focusing their search efforts on cheaper solutions.

Figure 4.4 shows the global search algorithm SYNTHESIZE. The global search runs τ local solvers in parallel, each executing $\exists\forall$SOLVEASYNC on a logical encoding of the synthesis problem for a particular sketch S from $\mathcal{S}$. Our algorithm assumes the existence of a procedure (as provided by, for example, Sketch [122] or ROSETTE [126]) that can encode the application of an arbitrary program from a sketch as a term in the theory $\mathcal{T}$. A local solver that completes a search with the sketch S returns a tuple $\langle S, \mathit{result}, h \rangle$ of results to the global search on line 12. The variable result is SAT if there is a completion of the sketch S that satisfies φ, or UNSAT otherwise. If result is SAT, the program $S[H := h]$ is a completion of S that satisfies φ; if result is UNSAT, h is $\perp$.

If a local solver produces a new solution that is the best seen so far (line 15), the global search filters $\mathcal{S}$ with the gradient function g (line 17). The gradient function restricts $\mathcal{S}$ to include only sketches that may contain programs cheaper than the new best cost c^*. The global search then announces c^* to all currently running solvers as a new constraint in theory $\mathcal{T}$. Consequently, the application of κ to an arbitrary program from S needs to be reducible to a term in $\mathcal{T}$ (as mentioned in Section 4.2). The local solvers use this constraint to prune their candidate space

```
1  global τ                                                          ▷ Number of parallel threads

2  function Synthesize(φ, m = ⟨S, κ, g⟩)
3      V ← ∅                                                         ▷ Completed sketches
4      P*, c* ← ⊥, ∞                                                 ▷ Optimal program and cost
5      R ← Take(S, τ)                                                ▷ Remove first τ sketches from S
6      for all S ∈ R do
7          η ← VarsForHoles(S)                                       ▷ Logical variables for holes
8          x ← VarsForInputs(S)                                      ▷ Logical variables for inputs
9          ψ ← λe.λa.φ(a, ToSMT(S[H := e](a)))
10         ∃∀SolveAsync(S, ∃η.∀x.ψ(η, x))                            ▷ Start solving S
11     while R ≠ ∅ do
12         S, result, h ← WaitForResult(R)
13         P ← S[H := h]
14         t ← 0                                                     ▷ Number of new sketches to launch
15         if result = SAT ∧ κ(P) < c* then                          ▷ New optimal solution
16             P*, c* ← P, κ(P)
17             S ← g(c*)
18             for all S ∈ R do
19                 if S ∈ S then                                     ▷ Allow S to continue
20                     φ ← λe.λa.ToSMT(κ(S[H := e])) < c*
21                     SendConstraint(S, φ)
22                 else                                              ▷ Prune S
23                     KillSolver(S)
24                     R ← R \ {S}; V ← V ∪ {S}; t ← t + 1
25         else if result = UNSAT then                               ▷ No (more) solutions to S
26             KillSolver(S)
27             R ← R \ {S}; V ← V ∪ {S}; t ← t + 1
28         if t > 0 then
29             N ← Take(S \ (R ∪ V), t)
30             for all S ∈ N do
31                 η ← VarsForHoles(S)
32                 x ← VarsForInputs(S)
33                 ψ ← λe.λa.φ(a, ToSMT(S[H := e](a))) ∧
                           ToSMT(κ(S[H := e])) < c*
34                 ∃∀SolveAsync(S, ∃η.∀x.ψ(η, x))
35             R ← R ∪ N
36     return P*
```

Figure 4.4: The global optimal synthesis algorithm Synthesize takes as input a specification φ and a metasketch $\langle \mathcal{S}, \kappa, g \rangle$, and finds a program $P \in \bigcup_{S \in \mathcal{S}} \overline{S}$ that satisfies φ and minimizes κ. The synthesis runs τ local solvers ($\exists\forall$SolveAsync), each executing in parallel on its own thread, and coordinates their search activities by sharing constraints.

to include only programs cheaper than c^*. As an optimization, the global search can also kill local solvers that are no longer in the set $\mathcal{S}$ after applying the gradient function. This is sound because if a gradient function $g(c^*)$ filters out a sketch S, then by Equation (2), the sketch S has no solutions with cost less than c^*. Therefore, when the local search for S receives the new constraint that its cost be less than c^*, it will return UNSAT.

If a local solver produces no solution (line 25), it is marked as completed and new solvers are launched on new sketches from $\mathcal{S}$. In addition to the specification φ, these new solvers take an additional constraint that requires their solutions to be cheaper than the best known solution so far.

4.4.2 *Local Searches*

The global search invokes a local search procedure ∃∀SolveAsync (Figure 4.5) on individual sketches S from the space $\mathcal{S}$ of the metasketch. The local search implements an incremental version of the CEGIS [122] algorithm for solving $\exists\eta.\forall x.\psi(\eta, x)$ synthesis queries—that is, for finding a binding h for η that makes the formula $\forall x.\psi(\eta := h, x)$ valid. The classic CEGIS algorithm uses one (incremental) solver instance, called the synthesizer, to search for an h that is correct for a set Z of values for x, by checking the satisfiability of the formula

$$\exists\eta. \bigwedge_{z \in Z} \psi(\eta, x := z).$$

If the synthesis formula is satisfiable, another solver instance, called the verifier, checks the satisfiability of the formula $\exists x.\neg\psi(\eta := h, x)$, looking for a value of x, called a counterexample, that invalidates the candidate solution h. If such a value exists, it is added to Z. This loop repeats until either the verifier returns UNSAT, indicating that h is valid, or the synthesizer returns UNSAT, indicating that there are no candidates left.

The ∃∀SolveAsync algorithm extends CEGIS to accept additional constraints inside of the CEGIS loop. After each iteration of the CEGIS loop, our incremental algorithm can accept a new constraint $\psi'(\eta, x)$ (line 22) to obtain a new synthesis problem $\exists\eta.\forall x.\psi(\eta, x) \wedge \psi'(\eta, x)$. This constraint is added to the set of constraints seen so far (line 24), and asserted to the synthesizer for each counterexample collected so far (line 23). The synthesizer then searchers for a model of the new problem (line 9), and if the result is a valid solution (line 11), ∃∀SolveAsync emits that solution to its output channel (line 17). As in classic CEGIS, an invalid solution leads to a counterexample, which is added to Z (lines 12–14). If there is no model for the problem, the search terminates (lines 19–20). When the algorithm satisfies all the constraints it has received so far, it blocks until it receives new constraints.

4.4.3 *Characterization*

We now show that the global search Synthesize is sound (it returns only correct programs), complete (it returns a correct program if one exists), and optimal (it returns the cheapest correct program). To start, we prove soundness and complete-

```
1  function ∃∀SolveAsync(id, ∃η. ∀x. ψ(η, x))
2      G_S ← new IncrementalSMTSolver()
3      y ← ValuesFor(x)                                  ▷ Arbitrary initial binding for x
4      Z ← {y}                                           ▷ CEGIS counterexamples
5      Ψ ← {ψ}                                           ▷ All constraints
6      Assert(G_S, ψ(η, x := y))                         ▷ Assert that ψ holds for y
7      while true do
8          block ← True
9          result, h ← Solve(G_S)                        ▷ Solve for η
10         if result = SAT then                          ▷ h is a candidate model
11             result, z ← Verify(Ψ, η := h, x)
12             if result = SAT then                      ▷ Candidate is incorrect
13                 Assert(G_S, ⋀_{ϕ∈Ψ} ϕ(η, x := z))
14                 Z ← Z ∪ {z}                           ▷ z is a counterexample
15                 block ← False                         ▷ Do not wait for more constraints
16             else                                      ▷ Candidate is valid; send to global search
17                 SendResult(id, SAT, h)
18         else                                          ▷ Ψ is not valid
19             SendResult(id, UNSAT, ⊥)
20             return
21         if block ∨ a constraint has been received then
22             ϕ ← ReceiveConstraint()
23             Assert(G_S, ⋀_{z∈Z} ϕ(η, x := z))
24             Ψ ← Ψ ∪ {ϕ}

25 function Verify(Ψ, η := h, x)
26     G_V ← new SMTSolver()
27     Assert(G_V, ⋁_{ϕ∈Ψ} ¬ϕ(η := h, x))
28     return Solve(G_V)                                 ▷ Solve for x
```

Figure 4.5: The local synthesis algorithm ∃∀SolveAsync takes as input a constraint ψ over a list of existentially quantified variables η and universally quantified variables x. The algorithm is an incremental form of counterexample-guided inductive synthesis (CEGIS) that accepts new constraints within the CEGIS loop. These new constraints are conjoined to ψ in the order in which they are received. ∃∀SolveAsync emits models for η that make the resulting conjunctions valid. The search terminates if the current conjunction becomes unsatisfiable.

ness of ∃∀SolveAsync. Next, we use the soundness of ∃∀SolveAsync to establish the soundness of Synthesize. We then observe that Synthesize is not guaranteed to terminate on an arbitrary metasketch with an infinite search space. To prove completeness and optimality, we therefore introduce *compact metasketches* (Def. 4.8), which place a simple compactness requirement on the gradient function g. This requirement is true of all metasketches with finite search spaces, and, in practice, is easy to satisfy for infinite search spaces as well. But Synthesize is still useful for non-compact metasketches: because it will always *discover* a solution if one exists (a property we call *online completeness*), an implementation that emits intermediate results (on line 16 of Figure 4.4) can be used to find the best solution within a given time budget.

Local Search. The global search invokes ∃∀SolveAsync on the ∃∀ synthesis query for a given sketch S with respect to the correctness specification φ. To prove that the global search is sound, we need to show simply that no solution produced by ∃∀SolveAsync violates φ (Lemma 4.3). Completeness of ∃∀SolveAsync is more subtle, however. Unlike classic CEGIS, the incremental CEGIS explicitly filters out some solutions satisfying φ by receiving additional constraints. We prove completeness of ∃∀SolveAsync in the case where it has received a set of constraints Ψ, but then receives no further constraints until it sends a result (Lemma 4.5). This completeness result is sufficient for proving completeness of the global search on compact metasketches.

Lemma 4.3 (Soundness of ∃∀SolveAsync)**.** *Let $\exists\eta.\ \forall x.\ \psi(\eta, x)$ be the problem with which ∃∀SolveAsync is initialized. If the algorithm emits a result of the form $\langle \mathrm{id}, SAT, h\rangle$, then $\forall x.\ \psi(\eta := h, x)$ is valid modulo $\mathcal{T}$.*

Proof. If ∃∀SolveAsync sends a result $\langle id, \mathrm{SAT}, h\rangle$ from line 17 in Figure 4.5, then the verification on line 11 must have returned UNSAT. By the definition of Verify, this means that $\nexists z.\ \bigvee_{\phi\in\Psi} \neg\phi(\eta := h, z)$, and therefore that $\forall z.\ \bigwedge_{\phi\in\Psi} \phi(\eta := h, z)$. Since $\psi \in \Psi$ and additional constraints in Ψ can only rule out solutions, we have that $\forall x.\ \psi(\eta := h, x)$ is valid modulo $\mathcal{T}$. □

Lemma 4.4 (∃∀SolveAsync loop invariant)**.** *At line 8 of Figure 4.5, the state of the incremental solver G_S is the assertion $\bigwedge_{\phi\in\Psi}\bigwedge_{z\in Z}\phi(\eta, x := z)$.*

Proof. By induction on loop iterations. On loop entry, $\Psi = \{\psi\}$ and $Z = \{y\}$, and the only assertion is $\psi(\eta, x := y)$. Now suppose the state is $\bigwedge_{\phi\in\Psi}\bigwedge_{z\in Z}\phi(\eta, x := z)$ at the start of the current iteration. The iteration can add only a single new counterexample z'; if it does, it will assert $\bigwedge_{\phi\in\Psi}\phi(\eta, x := z')$ (line 13), and set $Z' = Z \cup \{z'\}$; if not, it will set $Z' = Z$. (line 14) Then the iteration can add a single new constraint ϕ'; if it does, it will assert $\bigwedge_{z\in Z'}\phi'(\eta, x := z)$ (line 23) and set $\Psi' = \Psi \cup \{\phi'\}$ (line 24); if not, it will set $\Psi' = \Psi$. Therefore, at the start of the next iteration, the assertion store contains $\bigwedge_{\phi\in\Psi'}\bigwedge_{z\in Z'}\phi(\eta, x := z)$. □

Lemma 4.5 (Completeness of ∃∀SolveAsync)**.** *Let $\exists\eta.\ \forall x.\ \psi(\eta, x)$ be the problem with which ∃∀SolveAsync is initialized. Suppose that ∃∀SolveAsync receives constraints $\phi_1, \ldots, \phi_k$, such that $\Psi = \{\psi, \phi_1, \ldots, \phi_k\}$. If no more constraints are re-*

ceived, and there exists some assignment h such that $\forall x.\ \bigwedge_{\phi\in\Psi}\phi(\eta := h, x)$ is valid modulo $\mathcal{T}$, then $\exists\forall$SolveAsync will eventually send a SAT result.

Proof. Suppose we are at line 8 of Figure 4.5, when the previous iteration of the loop received the last message ϕ_k. Let Z' be the set Z of counterexamples at this point. We now have a set of specifications Ψ that will not change again. By Lemma 4.4, the accumulated state of G_S is the assertion $\bigwedge_{\phi\in\Psi}\bigwedge_{z\in Z'}\phi(\eta, x := z)$. From this point, the algorithm reduces to classic CEGIS, which is sound and complete on bounded input domains. □

Global Search. The correctness of the global search depends on the soundness and completeness of the local search. We first show that Synthesize is sound for the classic synthesis problem.

Theorem 4.6 (Soundness of Synthesize)**.** *Let $m = \langle\mathcal{S}, \kappa, g\rangle$ be a metasketch and φ a specification. If Synthesize(φ, m) returns a program P, then P is a solution to the classic synthesis problem; that is, $\forall x.\ \varphi(x, [\![P]\!](x))$ is valid modulo $\mathcal{T}$.*

Proof. Follows immediately from Lemma 4.3. □

As stated, the Synthesize procedure is not complete: it is not guaranteed to return a solution if one exists, because it is not guaranteed to terminate. The issue is that both the space $\mathcal{S}$ and the sets returned by the gradient function g may be countably infinite. However, we can show that Synthesize will always *discover* a solution if one exists; that is, Synthesize is a semi-decision procedure. We call this property *online completeness*.

Theorem 4.7 (Online completeness of Synthesize)**.** *Let $m = \langle\mathcal{S}, \kappa, g\rangle$ be a metasketch and φ a specification. Suppose that there exists a program $P \in \bigcup_{S\in\mathcal{S}}\overline{S}$ in the search space defined by the metasketch such that $\forall x.\ \varphi(x, [\![P]\!](x))$ is valid modulo $\mathcal{T}$. Then at some point during execution, the call to WaitForResult on line 12 returns a tuple with $result = SAT$.*

Proof. Because $P \in \bigcup_{S\in\mathcal{S}}\overline{S}$, there exists a sketch $S \in \mathcal{S}$ such that $P \in \overline{S}$. Then there are three possibilities for how Synthesize treats the sketch S, all of which guarantee that the global search receives a SAT message:

- S is launched by a call to $\exists\forall$SolveAsync which then receives no constraint messages from the global search. Then by Lemma 4.5, because S is satisfiable, the global search eventually receives a SAT message with sketch S.
- S is launched by a call to $\exists\forall$SolveAsync and receives at least one constraint message from the global search. But constraint messages are only sent from line 21 of Figure 4.4, which is only reachable when WaitForResult receives a SAT message.
- If S is never launched, it must have been removed from $\mathcal{S}$. This removal can only happen on line 23 of Figure 4.4, which is only reachable when WaitForResult receives a SAT message.

□

Online completeness is useful because an implementation of SYNTHESIZE could emit intermediate results while continuing its search. As results in Section 4.5.4 show, SYNTHESIZE spends most of its execution time proving optimality of a candidate program, and so emitting intermediate results can make synthesis much faster at the expense of a weaker optimality guarantee. Online completeness ensures that SYNTHESIZE will always emit an intermediate solution if any solutions exist.

Compact Metasketches The global search is not guaranteed to terminate on an arbitrary metasketch, as explained above. To guarantee termination, we introduce an additional *compactness* constraint on the gradient function of a metasketch. This constraint is sufficient to prove that SYNTHESIZE is complete and optimal.

Definition 4.8 (Compact Metasketch)**.** *A* compact metasketch *is a metasketch* $m = \langle \mathcal{S}, \kappa, g \rangle$ *satisfying Definition 4.2 with the additional property that for all* $c \in \mathbb{R}$, $g(c)$ *is finite.*

Theorem 4.9 (Completeness of SYNTHESIZE)**.** *Let* $m = \langle \mathcal{S}, \kappa, g \rangle$ *be a compact metasketch and* φ *a specification. Suppose that there exists a program* $P' \in \bigcup_{S \in \mathcal{S}} \overline{S}$ *in the search space defined by the metasketch such that* $\forall x.\, \varphi(x, [\![P']\!](x))$ *is valid modulo* $\mathcal{T}$*. Then there exists a program* P *such that* SYNTHESIZE(φ, m) *returns* P*.*

Proof. By Theorem 4.7, there is at least one sketch S and program P such that WAITFORRESULT will return the message $\langle \text{SAT}, S, P \rangle$. Let this be the first such SAT message. Then $c^* = \infty$, and so line 17 will set $\mathcal{S}' = g(\kappa(P))$. Since m is a compact metasketch, $\mathcal{S}'$ is finite, and since line 17 is only called when a new cost is smaller than c^*, no new sketches can be added to $\mathcal{S}'$. Therefore there are only finitely many sketches remaining to explore. Each sketch has only finitely many solutions and, whenever a sketch returns a SAT message, it either receives a new constraint ruling that solution out (if the solution it returned has cost $\kappa(P) < c^*$), or a constraint ruling that solution out is already waiting on its queue (if $\kappa(P) \geq c^*$). Therefore, local solvers can only return finitely many more solutions, after which they will return UNSAT and be added to $\mathcal{V}$. Eventually, the set $\mathcal{S} \setminus (\mathcal{R} \cup \mathcal{V})$ of unexplored sketches will be empty, the running sketches will return UNSAT, and SYNTHESIZE will return a program. □

Theorem 4.10 (Optimality of SYNTHESIZE)**.** *Let* $m = \langle \mathcal{S}, \kappa, g \rangle$ *be a compact metasketch and* φ *a specification. Suppose that* SYNTHESIZE(φ, m) *returns a program* P *with cost* c*. Then* P *is an optimal program: there is no program* $P' \in \bigcup_{S \in \mathcal{S}} \overline{S}$ *such that* $\forall x.\, \varphi(x, [\![P']\!](x))$ *is valid modulo* $\mathcal{T}$*, and* $\kappa(P') < c$*.*

Proof. We proceed by contradiction. Suppose SYNTHESIZE returns P with $\kappa(P) = c$, but there is a sketch $S' \in \mathcal{S}$ that contains a correct program P' with $\kappa(P') = c' < c$. Since the assignment to c^* is guarded by line 15, c^* can only decrease, and by assumption, will never be smaller than c. By the definition of the gradient function, the sketch S' is never filtered out by line 17, since $c' < c \leq c^*$. Hence, when SYNTHESIZE receives a SAT message with a sketch S and cost c, either a

local search for S' is still running, or it has not started. In both cases, the local search for S' will receive the constraint $\text{ToSMT}(\kappa(S[H := e])) < c$ (at line 21 or at line 34), and it will receive no further constraints (since we assumed that SYNTHESIZE returns a program with cost c). By Lemma 4.5, the local search will run to completion and will be satisfiable, returning a correct solution P' with cost $\kappa(P') < c$. This solution will be received by SYNTHESIZE on line 12, contradicting the assumption that SYNTHESIZE returns P. □

4.4.4 *Implementation*

We implemented our optimal synthesis approach in a new tool we call SYNAPSE, built on top of the Rosette language (Chapter 2). Here we highlight some implementation details.

Sharing Counterexamples. The incremental CEGIS algorithm in Figure 4.5 can receive new constraints after each iteration. But the algorithm can be extended to also receive other messages. SYNAPSE exchanges CEGIS counterexamples between different local solvers in an effort to speed up each search. When a local solver sends a SAT or UNSAT message, it also includes the set Z of counterexamples it used to generate that result. The global search broadcasts the new counterexamples it receives to all running solvers, and maintains a set of all counterexamples that it provides to new local solvers. This optimization is sound because it does not affect the VERIFY check in ∃∀SOLVEASYNC. SYNAPSE uses the shared counterexamples only to accelerate the VERIFY check in ∃∀SOLVEASYNC, by first checking that the assertion on line 27 is not trivially invalidated by any of the existing counterexamples. This optimizations allows solvers to reduce the number of solver queries. We measure the effect of this optimization in Section 4.5.5.

Timeouts. While individual sketches are finite and therefore local searches will terminate, the queries made by local searches can take too long to be practical. We control this effect by adding a timeout parameter to ∃∀SOLVEASYNC. Once the timeout expires, the local solver sends a timeout message to the global search. The global search treats a timed-out search in the same way as an unsatisfiable one: it kills the local solver and launches the next sketch.

Timeouts weaken the optimality guarantee that SYNAPSE provides. A solution output by SYNAPSE is only guaranteed to be optimal among those sketches that did not time out. In practice, the metasketches we designed were unlikely to contain cheaper solutions in sketches that timed out, and extending the time out by an order of magnitude did not change our results.

Search Order. The completeness of SYNAPSE does not depend on the order $\preceq$ of the set of sketches $\mathcal{S}$ in a metasketch. The only requirement is that the order is total (as Definition 4.2 states), so that for every sketch $S \in \mathcal{S}$, SYNTHESIZE eventually either tries to solve that sketch, or prunes it by finding a cheaper solution.

However, the search order can have a significant effect on performance. In the example metasketch designs in Section 4.3.3, we were careful to select a search

order $\preceq$ that preferred simpler sketches to more complex ones. This order avoids wasted work on complex sketches that are likely to time out. It also best exploits the counterexample sharing optimization described above, as smaller sketches quickly generate a set of counterexamples that later local searches can use. We found the Cantor and Szudzik orders [81, 124] to be particularly effective.

4.5 EVALUATION

To demonstrate that SYNAPSE effectively solves optimal synthesis problems expressed as metasketches, we evaluated it on four sets of benchmarks drawn from existing work. We sought to answer the following questions:

1. Is SYNAPSE a practical approach to solving different kinds of synthesis problems? In particular, can it solve optimal synthesis problems? Do metasketches also enable more effective classic synthesis compared to existing syntax-guided synthesizers?
2. Does the fragmentation of the search space by a metasketch translate into parallel speedup?
3. Is *online completeness* empirically useful? What proportion of SYNAPSE's run time is spent finding an optimal solution versus proving its optimality?
4. How beneficial are our optimizations at the level of metasketches (realized through structure constraints) and within the implementation (realized through counterexample sharing)?
5. Can SYNAPSE reason about dynamic cost functions; that is, cost functions that execute the synthesized program?

This section presents our benchmarks, experiments, and results. The results provide affirmative answers to all five questions. SYNAPSE, our benchmarks, and our experimental data are available online[4] and have been artifact evaluated.

4.5.1 *Benchmarks*

Table 4.1 shows the benchmarks used in our evaluation. The benchmark problems come from two sources: the 2014 and 2015 syntax-guided synthesis (SyGuS) competitions [10], and common approximate computing benchmarks [57]. We selected 67 problems from four categories of the SyGuS competition, ranging in difficulty from easy (i.e., solvable by most solvers) to hard (i.e., unsolvable by most solvers). The approximate computing benchmarks consist of 7 programs that cannot be approximated with existing software-based techniques. We developed metasketches for each set of problems. Section 4.3.3 described some of these metasketches, and we describe the rest below.

4 `http://synapse.uwplse.org`

Table 4.1: The benchmarks used in our evaluation. For each benchmark suite, we wrote a metasketch whose set of sketches together form the relevant search space, as described in Section 4.3.3.

Benchmark Suite	Problems	Source	Metasketch	Cost Function
Array Search	14	SyGuS'14 [10]	Array programs	Expression depth
Search a sorted array of size n for a given element and return its index				
Conditional Integer Arithmetic (CIA)	13	SyGuS'15 [12]	Integer programs	Expression depth
Integer programs that use complex branching structure				
Hacker's Delight d0	20	SyGuS'14 [10]	Superoptimization	Program length
Bit-manipulating programs, with sketches in the metasketch containing only the minimal set of bitvector operators necessary to implement the reference program.				
Hacker's Delight d5	20	SyGuS'14 [10]	Superoptimization	Program length
Bit-manipulating programs, with sketches in the metasketch containing all operators from the theory of bitvectors.				
Parrot	3	Parrot [57]	Adaptive superopt.	Static cost model
Approximate computing kernels: kmeans and sobel ($\times 2$ convolution matrices)				
Parrot (polynomial)	4	Parrot [57]	Piecewise polynomial	Pieces + Degree
Approximate computing kernels: fft ($\times 2$ outputs) and inversek2j ($\times 2$ outputs)				

Hacker's Delight. The first category contains 20 bit-manipulating problems, used as superoptimization benchmarks in previous synthesis work [68, 115], appearing in two different difficulties, d0 and d5. The metasketch for a d0 problem includes only the bitvector operators that appear in the reference solution for that problem. The metasketch for d5 problems includes all bitvector operators.

Array Search. The second category contains 14 array search problems from the SyGuS competition [10]. The problem arraysearch-n is to synthesize a program that returns the index of a search key in a sorted array of size n, or zero if the key is not present in the array. The most efficient solution to these problems implements binary search. The metasketch for array search problems is an infinite set of sketches generated by two mutually recursive functions that encode SyGuS grammars for integer and boolean expressions. Each sketch in this metasketch is parameterized by the depth of the deepest integer and boolean expressions, respectively.

Conditional Integer Arithmetic (CIA). The third category contains 13 conditional integer arithmetic problems[5] new to the 2015 syntax-guided synthesis competition [12]. Each problem involves synthesizing a program from a grammar that includes the program inputs; constants 0, 1, and 3; integer addition and subtraction; and the `qm` operation

```
(define (qm a b)
  (if (< a 0) b a))
```

5 The conditional integer arithmetic benchmarks are labeled `qm` in the SyGuS competition dataset.

The metasketch for CIA problems is an infinite set of sketches generated by this grammar, with one sketch per depth of production from the grammar. Several of the CIA benchmarks were unsolved by any solver in the SyGuS competition; we present only those solved by at least one SyGuS solver or by Synapse.

Parrot. The fourth category contains 7 problems drawn from the approximate computing literature [57]. The specification for these problems allows the synthesized program to differ from the reference program by a given application-specific quality bound. We use two metasketches for the Parrot benchmarks: piecewise polynomial approximation (for benchmarks that use transcendental functions) and adaptive superoptimization (for all other benchmarks).

Methodology. We performed all experiments on an 18-core Intel Xeon E5-2666 CPU at 2.9 GHz, with 60 GB of RAM. For SyGuS benchmarks, we timed out each metasketch after one hour, for consistency with the SyGuS competition setup [10]. For Parrot benchmarks, we did not use a timeout for any metasketch. In both cases, individual sketches within a metasketch were timed out after 15 minutes. Section 4.4.4 describes the effect of timeouts on Synapse's optimality guarantee; we found that extending the individual sketch timeout by an order of magnitude did not discover cheaper solutions for any problem. All timing results are wall-clock times for the entire Synapse execution. Where speedups are presented, they are aggregated over all benchmarks in a category before being normalized to the relevant baseline [120].

4.5.2 *Is Synapse a practical approach to solving different kinds of synthesis problems?*

To evaluate the effectiveness of Synapse as a generic synthesis engine, we applied it to all of our benchmarks in sequential mode—that is, running only a single local search at a time. This gives a baseline for comparison against existing syntax-guided synthesis solvers, which are single-threaded. Figure 4.6 shows the sequential solving performance of Synapse on our benchmarks.

For the Hacker's Delight benchmarks at difficulty d0, Synapse solves all 20 problems. The performance is competitive with results from the syntax-guided synthesis (SyGuS) competition [10], showing that metasketches do not introduce additional overhead for easy problems. However, Synapse also solves problem 20, which none of the SyGuS solvers could solve in either 2014 or 2015.

For Hacker's Delight benchmarks at difficulty 5, Synapse is able to solve 18 of the 20 problems within a one hour timeout. This result is better than other SMT-based SyGuS solvers: the symbolic solver in 2014 [11, 68] times out on all 20 problems, the Sketch-based [122] solver in 2014 solves only problems 1–8, and the CVC4-based solver that won the 2015 competition [112] cannot solve problems 14 or 15. The winner of the SyGuS competition in 2014 used an enumerative brute force strategy, and solved the same 18 problems that Synapse solves in comparable time (same order of magnitude).

Synapse solves all Array Search problems. In comparison, the best SyGuS solver on these problems in 2014 was the Sketch-based [122] solver, which could solve

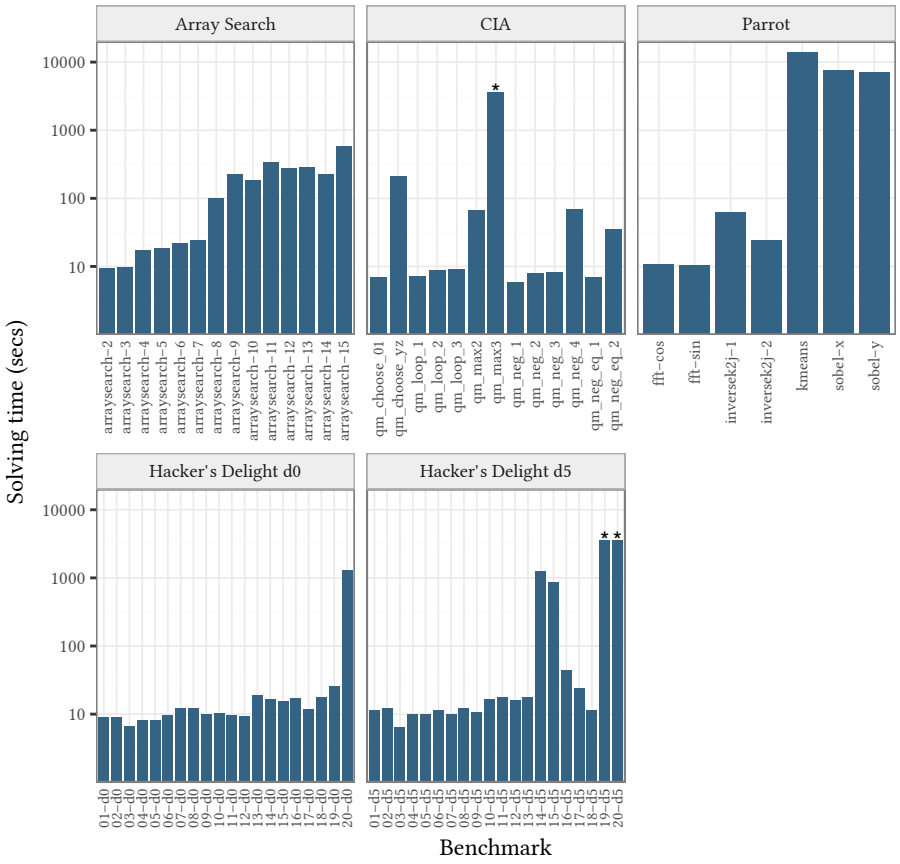


Figure 4.6: Sequential solving performance for all benchmarks. Asterisks indicate benchmarks that timed out after one hour.

these problems only up to length 7. The enumerative solver, which won the syntax-guided synthesis competition, could only solve lengths 2 and 3. In 2015, the CVC4-based solver [112] also solved all Array Search problems. However, its solutions were highly non-optimal: for arraysearch-15, Synapse produces the expected binary search solution with AST depth 5 and size 349 bytes, while CVC4 produces a solution with AST depth 45 and size 7.1 MB.

Synapse is also able to solve all seven Parrot problems. We attempted to solve the Parrot benchmarks using SyGuS solvers, Sketch [122], and the Stoke stochastic superoptimizer [115] without success. We encoded the adaptive superoptimization Parrot problems in the SyGuS benchmark format and in Sketch. Only the CVC4-based SyGuS solver [112] and Sketch produced solutions for these problems, but the solutions failed to meet the specification. The other publicly-available SyGuS solvers did not return solutions in 4 hours, which is the maximum time taken by Synapse to solve any Parrot benchmark. The piecewise polynomial Parrot problems are not expressible in the SyGuS format, because they require synthesizing (arbitrary numeric) constants. Sketch supports synthesis of constants, but it was unable to solve any of these problems in 4 hours. Implementing the benchmarks in C and passing them to Stoke also resulted in no solutions.

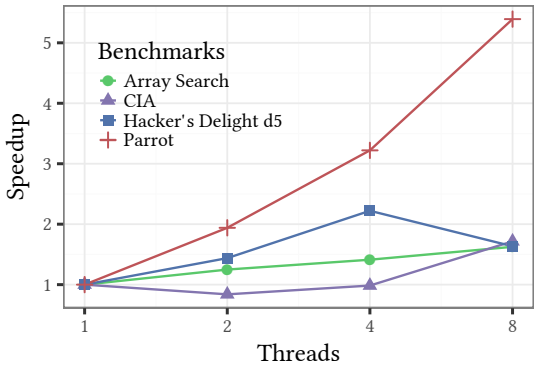


Figure 4.7: Parallel solving performance for all benchmarks except Hacker's Delight d0 (which are too small to benefit). Parrot sees substantial parallel speedup. Hacker's Delight d5 sees speedup up to four threads, but then a single local search dominates solving time. Array Search and Conditional Integer Arithmetic benchmarks see minimal speedup because most solving time is spent on a single local search.

4.5.3 *Does the fragmentation of the search space by a metasketch translate into parallel speedup?*

To evaluate the benefits of coarse-grained parallelism exposed by metasketches, we applied Synapse to our benchmarks using 2, 4, and 8 threads. Figure 4.7 shows the resulting parallel solving performance. We omit Hacker's Delight at difficulty d0 because these small benchmarks do not benefit from parallelization. Results are speedups in total execution time aggregated over all benchmarks in a category, excluding benchmarks that timed out at any number of threads.

Synapse realizes substantial parallel speedups for the Parrot problems, which are the hardest synthesis problems in our benchmark suite. These speedups are similar to or better than recent work in parallel program synthesis [73]. For Hacker's Delight problems, the parallel speedups are significant up to four threads, but eventually a single local search (which executes sequentially) becomes the bottleneck. For Array Search and Conditional Integer Arithmetic problems, parallel speedups are minimal, because most sketches early in the search order are quickly found to be unsatisfiable, so solving time is again dominated by a single sequential local search.

4.5.4 *Is* online completeness *empirically useful?*

Unlike a classic program synthesizer, which can terminate as soon as it discovers a solution, an optimal program synthesizer such as Synapse must also prove the optimality of a candidate solution. Metasketches provide an abstraction that allows this search to terminate despite exploring an infinite space of candidate programs. But the proof of optimality can still consume a significant portion of the search time: the gradient function of a metasketch returns sketches that *may* con-

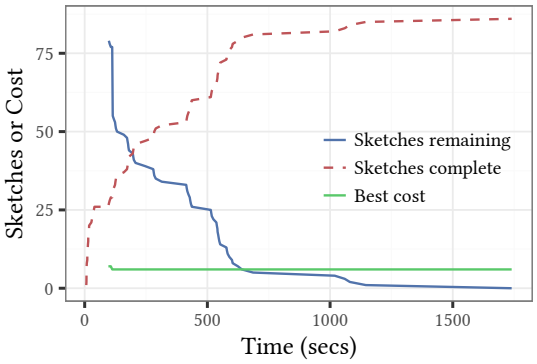


Figure 4.8: Search progress for the sobel-y benchmark on 4 threads. When solutions are discovered, the size of the remaining search space to explore drops significantly. SYNAPSE finds the optimal solution after 112 s, but must spend an additional 1,628 s proving it optimal.

tain cheaper candidate programs, and so the search can spend significant amounts of time searching sketches that do not contain cheaper solutions.

Figure 4.8 shows the progress over time of a search for the sobel-y Parrot benchmark. The x-axis is the time since starting the search, and the y-axis plots both the number of sketches completed and remaining in the search, and the cost of the best solution so far. Note that the number of sketches remaining is infinite before a first solution is found. The search discovers the optimal solution after 112 s with cost 6. However, the gradient function returns 56 sketches that may contain solutions of lower cost. The search spends another 1,628 s exploring each of these sketches to prove they do not contain such solutions. The slope of the sketches-remaining line in Figure 4.8 shows that many of these sketches can be quickly pruned, due to the added constraint they receive from the global search that their solutions must be cheaper than 6. For some sketches, however, the local search is unable to quickly deduce unsatisfiability despite this added constraint. These sketches dominate the search time.

4.5.5 *How beneficial are our metasketch and implementation optimizations?*

SYNAPSE admits two optimizations beyond existing CEGIS-based solvers, as Section 4.4.4 describes. First, the global search can exchange counterexamples between local searches, which can improve their performance by reducing the number of calls to the verifier. Second, a metasketch can impose structure constraints on the individual local searches, which can rule out some semantically-equivalent programs from being considered by multiple searches.

Figure 4.9 shows the effect of these optimizations for a single-threaded search. Results are speedups in total execution time aggregated over all benchmarks in a category, excluding benchmarks that timed out in any configuration. Both Hacker's Delight and Array Search benchmarks see minimal benefits from the optimizations, because they consist mainly of sketches that are easily proven unsatisfiable. The Parrot problems benefit significantly (50%) from structure constraints, be-

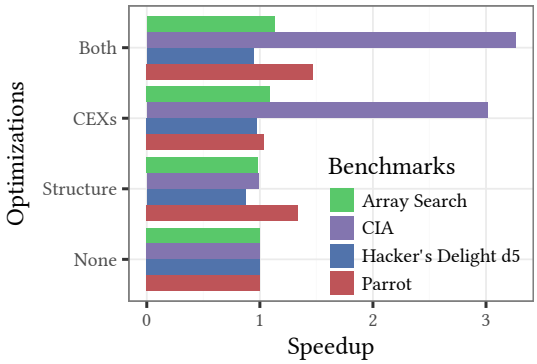


Figure 4.9: Effect of optimizations on Synapse's performance for a single-threaded search. Synapse can exchange counterexamples (CEXs) between local searches, and can impose structure constraints that prevent different local searches traversing some semantically-equivalent programs.

cause the sketches in the Parrot metasketches contain significant semantic overlap. The Conditional Integer Arithmetic benchmarks, on the other hand, benefit significantly (3×) from counterexample exchange.

4.5.6 *Can Synapse reason about dynamic cost functions?*

Metasketches place only very general restrictions on cost functions: the application of the cost function to a program must reduce to a term in a decidable theory (as discussed in Section 4.2). This restriction allows for static cost functions, such as static instruction cost models, but also for dynamic cost functions that execute the synthesized program to establish its cost. We illustrate Synapse's support for a variety of dynamic cost functions with three small examples.

Least-Squares Regression. Least squares regression fits a model function f to a data set $\{x_i, y_i\}$ by minimizing the objective function $\sum_{i=1}^{n}(y_i - f(x_i))^2$. We implemented a modified version of the piecewise polynomial metasketch presented in Section 4.3.3 to perform least-squares regression. Each sketch in this metasketch is a piecewise polynomial with a fixed number of pieces and fixed degree. We defined the cost function to be the least-squares objective function, which is dynamic because it requires evaluating the synthesized program f at each x_i in the data set. The metasketch uses the trivial gradient function $g(c) = \mathcal{S}$, and because this metasketch is not compact (Def. 4.8), we provided a finite set $\mathcal{S}$ of sketches. We used as a data set 30 samples of the polynomial

$$p(x) = x^3 - 8x^2 + x - 9$$

from the interval $x \in [-1, 10]$ with added Gaussian noise ($\sigma = 5$). Synapse synthesized the polynomial

$$q(x) = x^3 - 8x^2 + x - 7$$

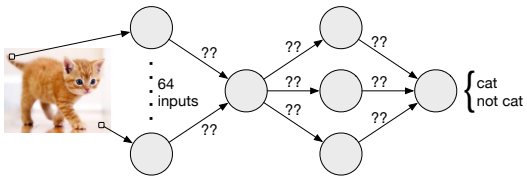


Figure 4.10: The sketch for a neural network is an SSA-form implementation of its evaluation function, with holes for each weight. In this example, the input nodes are the grayscale values of each pixel in the input image, and the output is a binary classification.

in 30 seconds as the optimal (integer) solution to this problem. SYNAPSE also explored the other sketches in the metasketch, which correspond to other possible models for the data (including linear, quadratic, and quartic functions), but correctly found the cubic function to be the best fit.

Worst-Case Execution Time. The superoptimization metasketches in Section 4.3.3 used static measures of program performance. The metasketch abstraction also supports dynamic measures, such as worst case execution time. To illustrate the effects of dynamic and static measures on the results of optimal synthesis, we selected problem 13 from the Hacker's Delight benchmark suite, which implements the sign function for 32-bit integers. We created two metasketches for this benchmark: $\langle \mathcal{S}, \kappa_s, g \rangle$ and $\langle \mathcal{S}, \kappa_d, g \rangle$, where $\mathcal{S}$ is a finite set of sketches and g is the trivial gradient $g(c) = \mathcal{S}$. Our sketches were drawn from a subset of the SYN grammar that includes conditional expressions and a minimal set of operators needed to implement benchmark 13. We defined κ_s to be an additive static cost function (as in Example 3 of Section 4.2) that assigns cost 2 to conditional expressions and cost 1 to all other expressions, including constants and variables. On the other hand, κ_d is a dynamic cost function that measures the worst-case execution time (over all feasible paths) using the same costs for SYN expressions as κ_s. We applied SYNAPSE to both metasketches and obtained two different optimal programs.

The static metasketch $\langle \mathcal{S}, \kappa_s, g \rangle$ produces the reference implementation for benchmark 13 as the optimal solution (with cost 8):

```
(define (sgn-s x)
  (| (>>> (- x) 31) (>> x 31)))
```

The dynamic metasketch $\langle \mathcal{S}, \kappa_d, g \rangle$, on the other hand, produces a different optimal solution (also with cost 8):

```
(define (sgn-d x)
  (if (< 0 x)
    (>>> -1 31) ; 1
    (>> x 31)))
```

The `sgn-d` function has cost 11 under the static cost function κ_s, and so is not an optimal solution to the static metasketch. SYNAPSE finds each solution in a few seconds.

Neural Networks. We developed a simple metasketch to train a feed-forward neural network classifier from a training set. The neural network metasketch $\langle \mathcal{S}, \kappa, g \rangle$ contains a sketch S_t in $\mathcal{S}$ for each of a (finite) set of neural network topologies t. A sketch S_t is an implementation of a feed-forward neural network in the SSA language of Figure 4.2 using fixed-point arithmetic and rectifier activation functions [102]. Each weight in the neural network is a hole in the sketch, as shown in Figure 4.10. The cost function κ simply counts the number of misclassified examples. Notably, we did not have to explicitly encode a training algorithm such as backpropagation.

We used this metasketch to train a simple image classifier to distinguish between cats and other images. We used 40 training examples (20 cats, 20 not-cats) from the CIFAR-10 dataset [80], resized to 8×8 pixels and converted to grayscale. SYNAPSE synthesized a neural network in 35 mins that achieved 95% recognition accuracy on the 40 examples. The synthesized network has 64 input nodes (one per pixel in the input image), a hidden layer of one node, a second hidden layer of three nodes, and a single output node. In training this network, SYNAPSE also explored the rest of the metasketch, consisting of all topologies with at most 2 layers and 4 nodes per layer (a total of 20 topologies).

Of course, this example is no advance in machine learning: we do not have anywhere near enough training examples, the recognition accuracy is measured on the training set, and the training time is many orders of magnitude slower than backpropagation. Rather, this example demonstrates that the underlying synthesizer can discover a training strategy given only an implementation of forward evaluation and the error function to minimize. It also demonstrates that SYNAPSE can handle large, under-constrained programs: the SSA-form program for the synthesized neural network contains 284 instructions, and some other topologies in the metasketch consist of over 1500 instructions.

4.6 RELATED WORK

Program synthesis is well studied in the literature. Some program synthesizers, and some applications of synthesis, implicitly or explicitly optimize an objective function. This section reviews related work on program synthesis, domain-specific synthesizers, and on optimal or quantitative synthesis.

Program Synthesis. Our work builds on recent advances in syntax-guided synthesis (e.g., [11, 68, 74, 115, 127, 130]), which are based on counterexample-guided search [122]. In addition to a correctness specification, a syntax-guided synthesizer takes as input a space of candidate programs, defined by a syntactic template, and searches it for a program (if any) that satisfies the specification. Existing approaches employ a variety of search procedures, including bottom-up enumeration [130], symbolic solving [68, 74, 122, 127], and stochastic search [115]. The recently developed syntax-guided synthesis (SyGuS) framework [11] unifies these approaches, providing a common language for expressing synthesis problems, a suite of standard benchmarks, and a set of search procedures for solving SyGuS problems. We drew several of our benchmarks from the SyGuS framework.

A number of SyGuS solvers implicitly minimize (or nearly minimize) a fixed objective function. A bottom-up enumerative solver [130] implicitly minimizes program length: shorter solutions will be discovered before longer ones. Some symbolic synthesis algorithms [68] use a fixed library of components, which is expanded only when the problem is unsatisfiable, implicitly directing the search toward simpler programs. In both cases, however, the optimization is implicit, and not easily extended to different cost functions. The winner of the 2015 SyGuS competition builds support for refutation-based synthesis into the CVC4 SMT solver [112]. While the refutation approach quickly produces correct solutions, those solutions are often extremely long. Extending the refutation approach with support for optimization would be non trivial.

Domain-Specific Synthesizers. Optimality is desirable in a number of synthesis applications, and so domain-specific synthesizers often implement an optimization strategy. Chlorophyll [107] is a synthesis-aided compiler for a low-power spatial architecture that performs modular superoptimization. The superoptimizer executes a binary search over programs given a cost model: it uses counterexample-guided inductive synthesis (CEGIS) to synthesize a program of cost k, and if one exists, to synthesize a program of cost $k/2$, and so on. To make the superoptimization scale to real-world programs, the process uses "sliding windows" to break a program into smaller pieces. Metasketches also give structure to the search, but our synthesis approach can provide whole-program optimality guarantees that the sliding windows technique cannot, and can reason about more general cost functions.

Feser, Chaudhuri, and Dillig [59] present a synthesis algorithm for producing data structure transformations from input-output examples. The algorithm guarantees optimality of the generated transformation with respect to an additive cost function over program syntax, which is defined similarly to the cost function in Example 3 of Section 4.2. In contrast, our approach is generic: it is applicable to a broad range of synthesis problems, and it can optimize a variety of cost functions, as long as their semantics is expressible in a decidable theory.

McSynth [123] is a synthesizer that generates machine code instructions from semantic specifications of their behavior. While McSynth alone does not consider optimality, the authors note that it could be extended to generate optimal solutions with a naive algorithm that generates *every* solution to the synthesis problem and returns the one among them with minimum cost. Metasketches provide considerably more structure to the optimal synthesis process, and can accommodate an unbounded space of sketches (and therefore solutions).

Optimal Synthesis. Recent work has considered optimality in synthesis, with various forms of ranking and weighting formulations [67, 96, 109], and in SMT problems in general. Chaudhuri, Clochard, and Solar-Lezama [42], for example, propose a smoothed proof search technique for synthesizing parameter holes in a program while optimizing a quantitative objective. Smoothed proof search reduces optimal synthesis to a sequence of optimization problems that can be solved numerically to satisfy the specification in the limit. The use of numeri-

cal optimization allows this technique to perform probabilistic reasoning, which Synapse does not support. However, the technique's sketching language is less expressive than a metasketch, allowing only linear operations on holes. The optimality guarantee also holds only over a single monolithic sketch, which restricts synthesis to a finite set of candidate programs; in contrast, a metasketch can represent an unbounded set of candidate programs.

Symba [85] is an SMT-based optimization algorithm for objective functions in the theory of linear real arithmetic (LRA). Symba optimizes the objective function by maintaining an under-approximation of the maximal cost, and using the SMT solver to generate new models for the specification that violate that under-approximation (i.e., are more optimal). This approach builds on a line of work on optimization for SMT problems [45, 116]. Unlike Symba, our approach supports non-linear cost functions (in, for example, the theory of bitvectors) and can optimize over an unbounded space of candidate programs. However, Symba is able to detect when the cost function it is maximizing has no upper bound, whereas a (compact) metasketch must explicitly rule out this possibility. Integrating Symba with our approach is a promising direction for future work.

4.7 CONCLUSION

Metasketches are a general framework for specifying and solving optimal synthesis problems. A metasketch fragments the search space of a synthesis problem into an ordered set of sketches, provides a cost function to optimize, and specifies a gradient function to direct the search toward cheaper regions of the space. This three-part abstraction enables the programmer to succinctly express both the desired (optimal) synthesis problem and a high-level strategy for solving it. By making the search strategy programmable, metasketches enable rapid creation of competitive (optimal) synthesis tools, ranging from superoptimization to approximation of computational kernels. Moreover, metasketches bring new expressive power to syntax-guided synthesis, including sketching of unbounded search spaces and use of dynamic cost functions that reason about program semantics. Our synthesis approach, implemented in Synapse, exploits the structure of metasketches to search for solutions in parallel, employing effective generic and problem-specific optimizations. Our results demonstrate that custom search strategies expressed via metasketches make it possible for Synapse to solve hard synthesis problems, both optimal and classic, which cannot be solved with general state-of-the-art synthesis algorithms.

5 SYMBOLIC PROFILING

Solver-aided tools rely on *symbolic evaluation* to reduce programming tasks, such as verification and synthesis, to satisfiability queries. Many reusable symbolic evaluation engines are now available as part of solver-aided languages and frameworks, which have made it possible for a broad population of programmers to create and apply solver-aided tools to new domains. But to achieve results for real-world problems, programmers still need to write code that makes effective use of the underlying engine, and understand where their code needs careful design to elicit the best performance. This task is made difficult by the all-paths execution model of symbolic evaluators, which defies both human intuition and standard profiling techniques.

This chapter presents *symbolic profiling*, a new approach to identifying and diagnosing performance bottlenecks in programs under symbolic evaluation.[1] To help with diagnosis, we develop a catalog of common performance anti-patterns in solver-aided code. To locate these bottlenecks, we develop SymPro, a new profiling technique for symbolic evaluation. SymPro identifies bottlenecks by analyzing two implicit resources at the core of every symbolic evaluation engine: the *symbolic heap* and *symbolic evaluation graph*. These resources form a novel performance model of symbolic evaluation that is general (encompassing all forms of symbolic evaluation), explainable (providing programmers with a conceptual framework for understanding symbolic evaluation), and actionable (enabling precise localization of bottlenecks). Performant solver-aided code carefully manages the shape of these implicit structures; SymPro makes their evolution explicit to the programmer.

To evaluate SymPro, we implement profilers for the Rosette solver-aided language and the Jalangi program analysis framework. Applying SymPro to 15 published solver-aided tools, we discover 8 previously undiagnosed performance issues. Repairing these issues improves performance by orders of magnitude, and our patches were accepted by the tools' developers. We also conduct a small user study with Rosette programmers, finding that SymPro helps them both understand what the symbolic evaluator is doing and identify performance issues they could not otherwise locate. SymPro has also been integrated into other tools, including the Crucible symbolic evaluation engine developed by Galois [61].

5.1 OVERVIEW

Solver-aided tools have automated a wide range of programming tasks, from test generation to program verification and synthesis. Such tools work by reducing programming problems to satisfiability queries that are amenable to effective SAT

1 This chapter was first published as the paper *Finding Code That Explodes Under Symbolic Evaluation*, by James Bornholt and Emina Torlak, published at OOPSLA 2018 [24].

or SMT solving. This reduction is performed by the tool's *symbolic evaluator*, which encodes program semantics as logical constraints. Effective symbolic evaluation is thus key to effective solver-aided automation.

Building and applying solver-aided automation used to be the province of experts, who would invest years of work to obtain an efficient symbolic evaluator for a new application domain. This barrier to entry is now greatly reduced by the availability of solver-aided languages (e.g., [126, 131]) and frameworks (e.g., [30, 117]), which provide reusable symbolic evaluation engines for programmers to target. These platforms have made it possible for a broader population of programmers, from high-school students to professional developers, to rapidly create solver-aided tools for many new domains (e.g., Table 5.1).

But scaling solver-aided programs to real problems, either as a developer or a user, remains challenging. As with classic programming, writing code that performs well (under symbolic evaluation) requires the programmer to be able to *identify* and *diagnose* performance bottlenecks—which parts of the program are costly to evaluate (symbolically) and why. For example, if a program synthesis tool is timing out on a given task, the tool's user needs to know whether the bottleneck is in the problem specification or the solution sketch [122]. Similarly, if neither the specification nor the sketch is the bottleneck, then the tool's developer needs to know where and how to improve the interpreter that specifies the semantics of the tool's input language. Yet unlike classic runtimes, which employ an execution model that is familiar to programmers and amenable to time- and memory-based profiling, symbolic evaluators employ an unfamiliar execution model (i.e., evaluating *all* paths through a program) that defies standard profilers. As a result, programmers currently rely on hard-won intuition and ad hoc experimentation to diagnose and optimize solver-aided code.

This chapter presents *symbolic profiling*, a systematic new approach to identifying and diagnosing code that performs poorly under symbolic evaluation. Our contribution is three-fold. First, we develop SymPro, a new (and only) profiling technique that can identify root causes of performance bottlenecks in solver-aided code. Here, we use the term 'solver-aided code' to generically refer to any program (in any programming language) that is being evaluated symbolically to produce logical constraints. Second, to help programmers diagnose these bottlenecks, we develop a catalog of the most common programming anti-patterns for symbolic evaluation. Third, we conduct an extensive empirical evaluation of symbolic profiling, showing it to be an effective tool for finding and fixing performance problems in real applications.

Symbolic Profiling. What characterizes the behavior of programs under symbolic evaluation? The fundamental challenge for symbolic profiling is to answer this question with a performance model of symbolic evaluation that is *general*, *explainable*, and *actionable*. A general model applies to all solver-aided platforms and must therefore encompass all forms of symbolic evaluation, from symbolic execution [47, 77] to bounded model checking [21]. An explainable model provides a conceptual framework for programmers to understand what a symbolic evaluator is doing, without having to understand the details of its implementa-

tion. Finally, an actionable model enables profiling tools to precisely identify root causes of performance bottlenecks in symbolic evaluation.

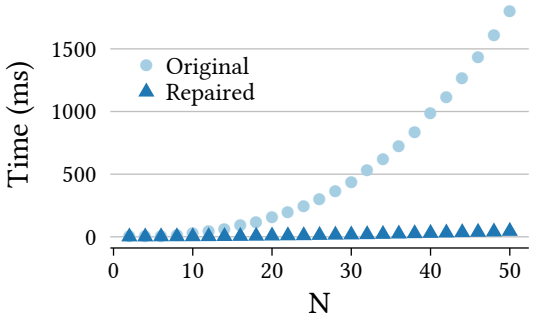
To illustrate the symbolic profiling challenge, consider applying a standard time-based profiler to the toy program in Fig. 5.1a. The program checks that the sum of any $n \leq N$ even integers is also even. Under symbolic evaluation for $N = 20$, the `take` call (line 5) takes an order of magnitude more time than `filter` (line 4), so a time-based profiler identifies `take` as the location to optimize. The source of the problem, however, is the call to `filter`, which generates $O(2^N)$ paths when applied to a symbolic list of length N (Fig. 5.1c). The `take` procedure, in contrast, generates $O(N)$ paths. A time-based profiler incorrectly blames `take` because it is evaluated 2^N times, once for each path generated by `filter`. The correct fix is to avoid calling `filter` (Fig. 5.1d). This repair location is missed not only by time-based profiling but also by models that rely on common concepts from the symbolic evaluation literature, such as path condition size or feasibility. For instance, a simple model that counts the total number of paths generated by a call will also blame `take`. It is, of course, possible to design more sophisticated time- and path-based models that can handle our toy example, and we examine such designs in Section 5.4.1, but they fall short on real code.

Symbolic profiling employs a new performance model of symbolic evaluation that is based on the following key insight: effective symbolic evaluation involves maximizing (opportunities for) concrete evaluation while minimizing path explosion. Classic concrete execution is thus a special,ideal case of symbolic evaluation, in which all operations are evaluated on concrete values along a single path of execution. Since this ideal cannot be achieved in the presence of symbolic values, symbolic evaluators choose which goal to prioritize at a given point by basing their evaluation strategy on either symbolic execution (SE) or bounded model checking (BMC). As illustrated in Fig. 5.4, SE maximizes concrete evaluation but suffers from path explosion, while BMC avoids path explosion but affords few opportunities for concrete evaluation. Performant solver-aided code elicits a practical balance between SE- and BMC-style evaluation in the underlying engine. The challenge for a symbolic profiler is therefore to help programmers identify the parts of their code that deviate most from concrete evaluation by generating excessive symbolic state or paths.

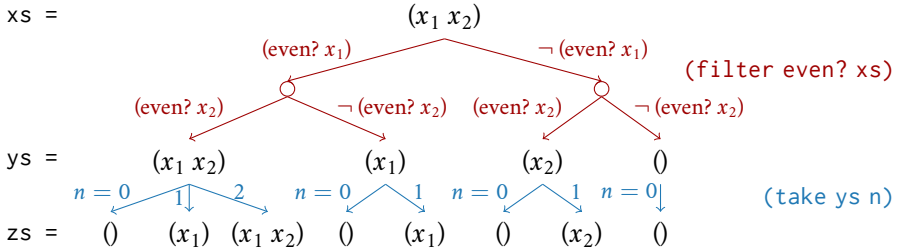
SymPro. We address this challenge with SymPro, a new profiling technique that tracks two abstract resources, the *symbolic heap* and the *symbolic evaluation graph*, which form our performance model. The symbolic heap consists of all symbolic values (constants, terms, etc.) created by the program, while the symbolic evaluation graph reflects the engine's evaluation strategy (which paths were explored individually, which were merged, etc.). In concrete execution, the symbolic heap is empty, and the evaluation graph consists of a single path. In symbolic evaluation, these resources evolve depending on the evaluation strategy. For example, the evaluation graph is a tree for SE engines, a DAG for BMC engines, and a mix of sub-trees and sub-DAGs for hybrid engines. The symbolic heap and graph are implicit in every forward symbolic evaluation engine, making our model general. They also capture the full spectrum of symbolic evaluation behaviors in an

```
1 (define (sum-of-even-integers-is-even N)
2   (define-symbolic* xs integer? [N]) ; xs = list of N symbolic ints
3   (define-symbolic* n integer?)      ; n = single symbolic integer
4   (define ys (filter even? xs))      ; ys = even integers from xs
5   (define zs (take ys n))            ; zs = first n elements of ys
6   (assert (even? (apply + zs))))     ; Check that sum of zs is even
```

(a) A program that checks that the sum of any $n \leq N$ even integers is even.



(b) The original program (a) performs poorly as N grows.



(c) The original program (a) creates $O(N2^N)$ paths (here, $N = 2$).

```
1 (define (sum-of-even-integers-is-even N)
2   (define-symbolic* xs integer? [N])
3   (define-symbolic* n integer?)
4   (define zs (take xs n))
5   (when (andmap even? zs)
6     (assert (even? (apply + zs)))))
```

(d) Repairing (a) to obtain asymptotically better performance.

Figure 5.1: A toy solver-aided program that performs poorly under symbolic evaluation.

implementation-independent way, making our model explainable and actionable. SymPro tracks the evolution of the symbolic heap and graph, identifying where new symbolic values are created, which values are frequently accessed, which values are eventually used in queries sent to a satisfiability solver, and how evaluation paths are merged at control-flow joins. It ranks procedure calls by these metrics to present the most expensive calls to the programmer. Given our motivating example from Fig. 5.1a, SymPro correctly identifies the call to `filter` as the bottleneck.

Anti-Patterns. To help the programmer diagnose the identified bottlenecks, we present a catalog of the most common performance anti-patterns in solver-aided code. These include *algorithmic*, *representational*, and *concreteness* problems. For example, the program in Fig. 5.1a suffers from *irregular representation*. It constructs a symbolic representation of $n \leq N$ even integers that describes $O(N2^N)$ concrete lists. The repaired program in Fig. 5.1d, in contrast, constructs a symbolic representation of $n \leq N$ integers that describes $O(N)$ concrete lists; this representation is then combined with a precondition (that all of its elements are even) before checking the desired property. We present a canonical example of each kind of anti-pattern, along with a repair the programmer could make.

Evaluation. We have implemented SymPro for the Rosette solver-aided language (Chapter 2). Our implementation is open-source and integrated into Rosette [125]. To evaluate the effectiveness of our profiler, we performed a literature survey of recent programming languages research, gathering 15 tools built using Rosette. Applying SymPro to these tools, we found 8 previously unknown bottlenecks. Repairing these bottlenecks improved the tools' performance by orders of magnitude (up to 290×), and several developers accepted our patches.

To demonstrate that SymPro profiles are *actionable*, we present detailed case studies on three of these Rosette-based tools, describing how a programmer can use SymPro to iteratively improve the performance of such a tool using language constructs afforded by Rosette and algorithmic changes guided by profile data. To show that SymPro is *explainable*, we conduct a small user study with Rosette programmers, showing that SymPro helps them identify performance bottlenecks in Rosette programs more efficiently than with standard (time-based) profiling tools. Finally, to show that symbolic profiling is *general*, we build a prototype symbolic profiler for Jalangi [117], a JavaScript program analysis framework with a symbolic execution pass [118], and show that it finds bottlenecks in JavaScript programs that a time profiler misses. As further evidence of generality, Galois, Inc. has integrated symbolic profiling into their Crucible symbolic evaluation engine [61], requiring only minimal changes to generate the input data for the symbolic profiling performance model.

5.2 EXAMPLE WORKFLOW

This section illustrates symbolic profiling on a small solver-aided program (Fig. 5.2). The performance bottleneck in this program is a recursive procedure (lines 18–

24) used by Uhler and Dave [131] to describe a common symbolic evaluation anti-pattern. We first show that time-based profiling fails to identify this procedure as the bottleneck, then apply symbolic profiling to identify, diagnose, and fix the issue.

A Small Solver-Aided Program. Our example program (Fig. 5.2) is written in Rosette, the solver-aided language introduced in Chapter 2. Programs written in Rosette behave like Racket programs when executed on concrete values, but Rosette lifts their semantics, via symbolic evaluation, to also operate on unknown *symbolic* values. These symbolic values are used to formulate *solver-aided queries*, such as searching for inputs on which a program violates its specification (verification), or searching for a program that meets a given specification (synthesis). The example implements a tool that verifies optimizations for a toy calculator language.

The `calculate` procedure (lines 4–16) defines the semantics of the calculator language with a simple recursive interpreter. A calculator program is a list of instructions that manipulate a single 4-bit storage cell, `acc`. An instruction consists of a 2-bit opcode and, optionally, a 4-bit argument. The language includes instructions for adding to, subtracting from, and squaring the value in the `acc` cell.

The toy calculator language is also equipped with procedures for optimizing calculator programs. One such procedure, `sub->add` (lines 25-29), takes as input a program and an index, and if the instruction at that index is a subtraction, `sub->add` replaces it with an equivalent addition instruction.

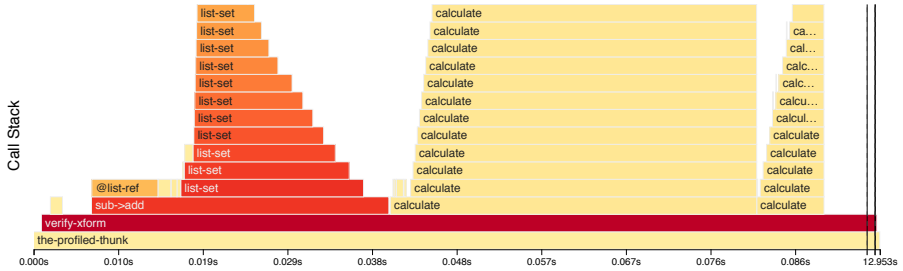
To check that these optimizations are correct, we implement a tiny verification tool, `verify-xform` (lines 31–41), using Rosette's **`verify`** query. The tool first constructs a symbolic calculator program P (lines 32–36) that represents all syntactically correct concrete programs of length N. This is done using Rosette's `(`**`define-symbolic*`** `id type)` form, which creates a fresh symbolic constant of the given `type` and binds it to the variable `id` every time the form is evaluated. Next, the tool applies the `(`**`verify`** `expr)` form to check that the input optimization `xform` preserves the semantics of P for all values of `acc` and the application index `idx`. This form searches for a concrete interpretation of the symbolic constants that violates an assertion encountered during the (symbolic) evaluation of `expr`. As expected, no such interpretation, or *counterexample*, exists for the `sub->add` optimization and programs of length $N \leq 5$.

Performance Bottlenecks. Verifying `sub->add` for larger values of N produces no counterexamples either, but the performance of the `verify-xform` tool begins to degrade, from less than a second for $N = 5$ to a dozen seconds for $N = 20$. While such degradation is inevitable given the computational complexity of the underlying satisfiability query, we have the (usual) practical goal of extracting as much performance as possible from our tool. With this goal in mind, we would like to find out what parts of the code in Fig. 5.2 are responsible for the degradation and how to improve them.

A first step in investigating the program's performance might be to use Racket's existing profiling support [18], or any other time-based profiler. The Racket profiler reports that most time is spent in `verify-xform`. It also cannot elide the in-

```
(define-values (Add Sub Sqr Nop) ; Calculator opcodes.
  (values (bv 0 2) (bv 1 2) (bv 2 2) (bv 3 2)))

(define (calculate prog [acc (bv 0 4)])
  (cond                       ; An interpreter for
    [(null? prog) acc]        ; calculator programs.
    [else                     ; A program is list of
     (define ins (car prog)) ; '(op) or '(op arg)
     (define op (car ins))    ; instructions that up-
     (calculate               ; date acc, where op is
      (cdr prog)              ; a 2-bit opcode and arg
      (cond                   ; is a 4-bit constant.
        [(eq? op Add) (bvadd acc (cadr ins))]
        [(eq? op Sub) (bvsub acc (cadr ins))]
        [(eq? op Sqr) (bvmul acc acc)]
        [else         acc]))]))

(define (list-set lst idx val) ; Functionally sets
  (match lst                   ; lst[idx] to val.
    [(cons x xs)
     (if (= idx 0)
         (cons val xs)
         (cons x (list-set xs (- idx 1) val)))]
    [_ lst]))


(define (sub->add prog idx)        ; Replaces Sub with
  (define ins (list-ref prog idx)) ; Add if possible.
  (if (eq? (car ins) Sub)
      (list-set prog idx (list Add (bvneg (cadr ins))))
      prog))

(define (verify-xform xform N) ; Verifies the given
  (define P                    ; transform for all
    (for/list ([i N])          ; programs of length N.
      (define-symbolic* op (bitvector 2))
      (define-symbolic* arg (bitvector 4))
      (if (eq? op Sqr) (list op) (list op arg))))
  (define-symbolic* acc (bitvector 4))
  (define-symbolic* idx integer?)
  (define xP (xform P idx))
  (verify ;  ∀ acc, idx, P. P(acc) = xform(P, idx)(acc)
   (assert (eq? (calculate P acc) (calculate xP acc)))))
```

Figure 5.2: A toy verifier in the Rosette solver-aided language. The performance bottleneck is the `list-set` procedure, originally used by Uhler and Dave to illustrate a symbolic evaluation anti-pattern they encountered when programming in the Smten solver-aided language [131].



Function	Score	Time (ms)	Term Count	Unused Terms	Union Size	Merge Cases
list-set xform.rkt:22 20 calls	3.3	19	2520	60	420	1700
verify-xform xform.rkt:36 1 call	1.4	12868	186	40	40	40
calculate xform.rkt:8 40 calls	1.3	41	380	180	0	280

Figure 5.3: Output of the SymPro symbolic profiler when run on the program in Fig. 5.2. Time spent in solver calls is collapsed into the dashed lines. SymPro identifies `list-set` as the performance bottleneck, ranking it highest in the *Score* column and highlighting it red.

ternal implementation details of Rosette from the profile, and so reports many spurious function calls that do not exist in the program as written. But even a Rosette-aware time-based profiler (essentially, the *Time* column in Fig. 5.3) reports `verify-xform` as the hot spot in this program, with the majority of the execution time spent directly in this procedure—specifically, calling an SMT solver from within the **`verify`** form. While useful, this information is not *actionable*, since it does not tell us where to attempt optimizations. When the solver is taking most of the time, what we want to know is the following: are there are any inefficiencies in the program that are causing the symbolic evaluator to emit a hard-to-solve (e.g., unnecessarily large) encoding?

Symbolic Profiling. To help answer this question, our symbolic profiler, SymPro, produces the output in Fig. 5.3 for $N = 20$. The table at the bottom of Fig. 5.3 identifies `list-set` as the main bottleneck, highlighting it in red and ranking it highest in the *Score* column. The score is computed (as Section 5.4 describes) from five statistics that quantify the effect of a procedure on the symbolic heap and evaluation graph:

- *Time* is the exclusive wall-clock time spent in a call;
- *Term Count* is the number of symbolic values created during a call;
- *Unused Terms* is the number of those values that do not appear in the query sent to the solver (i.e., the symbolic equivalent of garbage objects);
- *Union Size* is the sum of the out-degrees of all nodes added to the evaluation graph; and,
- *Merge Cases* is the sum of the in-degrees of those nodes.

The chart at the top of Fig. 5.3 visualizes the evolution of the call stack over time. Given this profile, it is easy to see that `list-set` has the largest effect on the heap and the evaluation graph, as well as the size of the final encoding, even though

both `verify-xform` and `calculate` are slower. Running the profiler on this benchmark has only minimal overhead: 4% slowdown and 19% additional memory.

Diagnosis and Repair. But why does `list-set` perform poorly under symbolic evaluation, and how can we repair it? The output in Fig. 5.3 shows that `list-set` creates many terms and performs many state merges. As noted by Uhler and Dave [131] and described in Section 5.3, the core issue is algorithmic. In particular, the recursive call to `list-set` is guarded by a short-circuiting condition `(= idx 0)` that is symbolic when `idx` is unknown. The symbolic evaluation engine must therefore explore both branches of this conditional, leading to quadratic growth in the symbolic representation (i.e., the term count in Fig. 5.3) of the output list:

```
> (define-symbolic* i integer?)
> (list-set '(1 2 3) i 4)
(list (ite (= 0 i) 4 1)
      (ite (= 0 i) 2 (ite (= 0 (- i 1)) 4 2))
      (ite (= 0 i) 3 (ite (= 0 (- i 1)) 3
                          (ite (= 0 (- i 2)) 4 3))))
```

The solution is to revise `list-set` to recurse unconditionally:

```
(define (list-set lst idx val)
  (match lst
    [(cons x xs)
     (cons (if (= idx 0) val x)
           (list-set xs (- idx 1) val))]
    [_ lst]))

> (list-set '(1 2 3) i 4)
(list (ite (= 0 i) 4 1)
      (ite (= 0 (- i 1)) 4 2)
      (ite (= 0 (- i 2)) 4 3))
```

With this revision, calls to `list-set` add at most $O(N)$ values to the symbolic heap, and the solving time for our verification query is cut in half for $N = 20$.

5.3 SYMBOLIC EVALUATION ANTI-PATTERNS

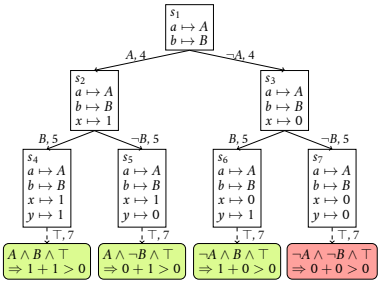
At the core of every symbolic evaluator is a strategy for reducing a program's semantics to constraints, and knowing what programming patterns are well or ill suited to an evaluator's strategy is the key to writing performant solver-aided code. This section presents three common *anti-patterns* that lead to poor performance under most evaluation strategies. We review the space of strategies first, followed by an illustration of each anti-pattern and a potential repair for it.

5.3.1 *Strategies for Reducing Programs to Constraints*

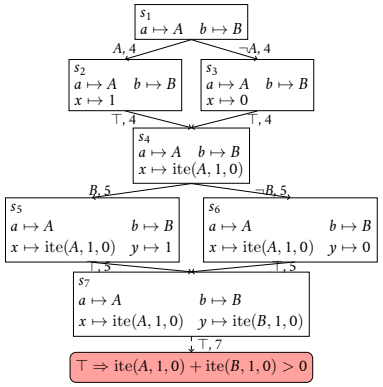
Symbolic evaluation engines rely on two basic strategies for reducing programs to constraints: *symbolic execution* (SE) [47, 77] and *bounded model checking* (BMC) [21]. There are engines that use just SE [36, 63, 64] or just BMC [17, 46, 138] or a hybrid

```
1 (define-symbolic* a boolean?)
2 (define-symbolic* b boolean?)

4 (define x (if a 1 0))
5 (define y (if b 1 0))

7 (assert (> (+ x y) 0))
```

(a) A program with a simple invalid assertion.



(b) Symbolic execution explores each control flow path through a program separately, resulting in a tree-shaped symbolic evaluation graph.



(c) Bounded model checking merges states from different paths at every control-flow join, resulting in a symbolic evaluation DAG.

Figure 5.4: An example of basic symbolic evaluation strategies. Symbolic execution and bounded model checking result in evaluation graphs of different shapes. Edge labels indicate the additional guard and the line of code that caused the transition.

of the two [62, 82, 118, 126]. We illustrate both SE and BMC on the program in Fig. 5.4a, and briefly review a hybrid approach [126]. For a more complete survey, we refer the reader to Torlak and Bodik [126] or Cadar and Sen [37].

Symbolic Execution. Symbolic execution (SE) reduces a program's semantics to constraints by evaluating and encoding individual paths through the program. Fig. 5.4b shows the *symbolic evaluation graph* (defined in Section 5.4) created by applying SE to the sample program in Fig. 5.4a. The nodes in the graph are program states, and the edges are transitions between states. Each edge is labeled with a guard and a program location that indicate where, and under what constraint, the transition is taken. The conjunction of all guards along a given path is called a *path condition*. The encoding of the program's semantics is the conjunction of the formulas $pc \Rightarrow \varphi$ at the leaves of the symbolic evaluation tree, where pc is the path condition and φ is the assertion at the end of that path. This encoding is worst-case exponential in program size, which is the key disadvantage of SE. The crucial advantage of SE is that it maximizes opportunities for concrete evaluation (e.g., line 7 is evaluated concretely along each path), leading to simpler and easier-to-solve queries.

Bounded Model Checking. Bounded model checking (BMC) avoids the exponential explosion of SE by merging program states at each control flow join, as shown in Fig. 5.4c. The resulting encoding of the program's semantics (i.e., the conjunction of the formulas at the leaves of the symbolic evaluation DAG) is polynomial in program size. The disadvantage of BMC, however, is the loss of opportunities for concrete evaluation. In our example, line 7 is evaluated symbolically, producing an encoding that requires reasoning about symbolic integers; the corresponding SE encoding, in contrast, uses only propositional logic. So while BMC encodings are compact, they are also harder to solve in practice.

Hybrid Approaches. Recent symbolic evaluation engines [118, 126] employ a hybrid of SE and BMC to offset the disadvantages of using either strategy on its own. These hybrid approaches generally prefer SE, applying BMC selectively to merge some paths and their corresponding states. For example, Rosette [126] performs BMC-style merging for values of the same primitive type; structural merging for values of the same shape (e.g., lists of the same length); and union-based merging (i.e., SE) for all other values:

```
1 (define-symbolic* b boolean?)

3 > (if b 1 0)            ; BMC-style merging
4 (ite b 1 0)
5 > (if b '(1 2) '(3 4)) ; structural merging
6 (list (ite b 1 3) (ite b 2 4))
7 > (if b 1 #f)           ; union-based merging (SE)
8 {[b 1] [(! b) #f]}
```

This evaluation strategy produces a compact encoding, like BMC, while creating more opportunities for concrete evaluation, like SE. But careful programming is still needed to achieve good performance, as we show next.

5.3.2 *Three Anti-Patterns in Solver-Aided Programs*

This section presents three kinds of *anti-patterns* in solver-aided code that lead to poor performance during symbolic evaluation. For each, we show an example of the issue and suggest potential repairs.

Algorithmic Mismatch. As observed by Uhler and Dave [131], small algorithmic changes can have a large impact on the efficiency of symbolic evaluation. Consider, for example, the `list-set` algorithm in Fig. 5.2 and the revised version presented in Section 5.2. The revised version is asymptotically better for engines that merge lists (e.g., [126, 131]). Yet the original version is asymptotically better when no merging of lists is performed (e.g., [118]). Such a *mismatch* between the algorithm and the underlying evaluation strategy can often be remedied with small changes to the algorithm's control flow. Symbolic profiling helps make these changes by informing the programmer of an algorithm's effect on the symbolic heap and the evaluation graph.

Irregular Representation. Poor choice of data structures is another frequent source of performance problems in solver-aided programs. Some common programming patterns, such as tree manipulations, require careful data structure design (see, e.g., [40]) to yield an effective symbolic encoding. In general, performance issues arise when the representation of a data type is *irregular* (e.g., a list of length one *or* two), increasing the number of paths that need to be evaluated to operate on a symbolic instance of that type.

To illustrate, consider the instruction data type for the calculator language from Fig. 5.2. Because an instruction is a list of the form `'(op)` or `'(op arg)`, applying `cadr` to a symbolic instruction at lines 13–14 involves evaluating two paths: one feasible (when the argument is present) and one infeasible (otherwise). Once the algorithmic mismatch in `list-set` is fixed, SymPro identifies this representational issue as the bottleneck by ranking `calculate` and `cadr` highest in the profile. Making the representation more regular—in our case, by replacing line 36 with `(list op arg)`—fixes the problem and leads to an additional 30% improvement in solving time. As this example illustrates, a less space-efficient but more uniform data representation is usually the better choice for symbolic evaluation.

Missed Concretization. In addition to employing careful algorithmic and representational choices, performant solver-aided code is also structured to provide as much information as possible about the feasible choices for symbolic values. Failing to make this information explicit is a common cause of bottlenecks that manifest as large evaluation graphs with many infeasible paths.

For example, consider the following toy procedure:

```
1  (define (maybe-ref lst idx) ; Return lst[idx]
2    (if (<= 0 idx 1)          ; if idx is 0 or 1,
3        (list-ref lst idx)    ; otherwise return -1.
4        -1))
```

Applying this procedure to a list of size N and a symbolic index results in an evaluation graph with $O(N)$ paths, only three of which are feasible. Refactoring the code to make explicit the concrete choices for `idx` leads to an asymptotically smaller evaluation graph and encoding:

```
1  (define (maybe-ref-alt lst idx)
2    (cond [(= idx 0) (list-ref lst 0)]
3          [(= idx 1) (list-ref lst 1)]
4          [else -1]))

6  (define-symbolic* idx integer?)
7  > (maybe-ref '(1 2 3 4 5 6) idx)     ; O(N) encoding
8  (ite (&& (<= 0 idx) (<= idx 1))
9       (ite* (⊢ (= 0 idx) 1) ... (⊢ (= 5 idx) 6))
10      -1)
11 > (maybe-ref-alt '(1 2 3 4 5 6) idx) ; O(1) encoding
12 (ite (= 0 idx) 1 (ite (= 1 idx) 2 -1))
```

In practice, this anti-pattern shows up in a more subtle form, where the feasible choices for a symbolic value are only known at run time. The fix then relies

on the host platform to provide a facility for expressing the set of feasible choices to the symbolic evaluator. We show an example of this more subtle issue and the corresponding fix in Section 5.5.1.

5.4 SYMBOLIC PROFILING

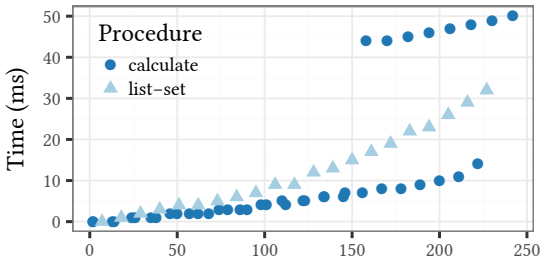
This section presents *symbolic profiling*, a new approach to identifying and diagnosing performance bottlenecks in programs under symbolic evaluation. As with any profiler, the key choice is what data to measure and where. We first review the space of alternative designs and then present our approach. We define the key parts of our performance model, the symbolic heap and evaluation graph; describe how a symbolic profiler analyzes them; and present two implementations of symbolic profiling.

5.4.1 *Designing a Symbolic Profiler*

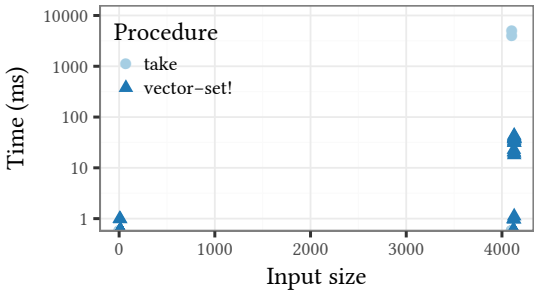
To help programmers identify performance bottlenecks in the symbolic evaluation of their code, a profiler must satisfy three key objectives. First, its output must be *explainable*: it must provide a few key concepts that programmers can use to understand the behavior of their code under symbolic evaluation, without understanding the implementation details of the underlying engine. Second, its output must be *actionable*, pointing programmers to the root cause of any discovered bottleneck—a location in the code that needs to be repaired to improve performance. Finally, a symbolic profiling technique should ideally be *general*, to handle the wide variety of symbolic evaluation strategies (from SE to BMC) and applications (e.g., bug finding, verification, and synthesis). Drawing on existing profiling and symbolic evaluation research, we evaluated several potential symbolic profiler designs against these criteria before settling on our approach.

Input-Sensitive Profiling. Our first design was based on input-sensitive profiling [49]. For each procedure in a program, input-sensitive profiling estimates its computational complexity as a function of its input size by fitting a function to its observed behavior. For example, such a profiler can determine that a linked-list traversal takes $O(n)$ time. Our intuition was that poor symbolic evaluation performance often comes from program locations that experience high complexity due to path explosion; a symbolic profiler could apply input-sensitive profiling and report the procedures with the worst computational complexity.

We implemented a prototype input-sensitive symbolic profiler to explore this hypothesis. However, we found that the correlation between input size and performance is often poor for code using symbolic evaluation. Minor perturbations in the input can cause the underlying engine to change its evaluation strategy, causing drastic changes in performance that make the estimated computational complexity inaccurate and noisy. For example, Fig. 5.5 shows the results from applying our prototype to the calculator program in Fig. 5.2 and the Ferrite case study in Section 5.5.1. For the calculator (a), the stratified results for `calculate` identify it as the most computationally complex function, even though `list-set`



(a) Calculator (Fig. 5.2)



(b) Ferrite (§5.5.1; note logarithmic y-axis)

Figure 5.5: Results from our input-sensitive profiling [49] prototype applied to two programs with known bottlenecks. In (a), the outlier results for `calculate` promote it to be the most computationally complex procedure. In (b), the profiler cannot fit a good function for `take`, and so identifies `vector-set!` instead.

is the true bottleneck. For Ferrite (b), there is no good function to fit for `take`, and so the profiler prefers functions such as `vector-set!` with more available data. In both cases, noise obscures the true bottlenecks.

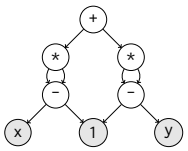
Path-Based Profiling. Our second design was inspired by the heuristics used in symbolic evaluation engines (e.g., [82]) to control path explosion. The resulting prototype symbolic profiler ranked functions based on the number of infeasible paths they explored and the total size of the path conditions generated during evaluation. Our intuition was that poorly performing procedures would generate many large, infeasible paths, and be likely candidates for repair.

However, this approach fell short in several ways. First, infeasible paths are not always the source of performance degradation. Applications such as program synthesis intentionally generate many large, feasible paths (e.g., to encode a sketch [122]), making this analysis ineffective. Second, when they are required, feasibility checks must be discharged by a constraint solver and so are extremely expensive; we observed profiler overheads of 100× on even simple benchmarks. Finally, a path-based profiler does not generalize to BMC-style evaluation, where performance bottlenecks manifest in the creation of large symbolic values during state merging, as illustrated in Section 5.2 for the `list-set` procedure.

With these experiences in mind, we sought to identify a performance model for symbolic profiling that would offer actionable advice in terms of a few abstract

```
1 (define-symbolic* x y integer?)
2 ; sq. dist from <x, y> to <1, 1>
3 (define dist
4   (+ (* (- x 1) (- x 1))
5      (* (- y 1) (- y 1))))
```

(a) A program that computes the squared distance between a symbolic and concrete point.



(b) The symbolic heap for the program.

Figure 5.6: The *symbolic heap* of a program tracks the structure of allocated symbolic values.

concepts, and accommodate the full spectrum of symbolic evaluation approaches. The remainder of this section describes our chosen design; our case studies in Section 5.5 and evaluation in Section 5.6 measure its success against these objectives.

5.4.2 *Instrumenting Symbolic Evaluation*

Our key insight is that every symbolic evaluator can be understood in terms of its interaction with two abstract data structures, the symbolic heap and the symbolic evaluation graph, which form our performance model for symbolic profiling. While explicit in our presentation, these data structures are usually implicit in an evaluator's implementation. We therefore also define a simple interface that any evaluator can implement to enable a symbolic profiler to reconstruct both data structures on the fly. The next section shows how to analyze these structures to produce actionable profile data.

Symbolic Heap. As an evaluator creates new symbolic values to reflect the program's semantics, it implicitly constructs a *symbolic heap*. For profiling purposes, the symbolic heap is analogous to the concrete heap: procedures that allocate many values on the heap are candidate bottlenecks.

Definition 5.1 (Symbolic heap)**.** *A* symbolic heap *is a directed acyclic graph* (V, E) *where each vertex* $v \in V$ *is a* term*. A term is either a* concrete constant*, or a* symbolic constant*, or an* expression*. Constants have no outgoing edges. Expressions are labeled with an operator, and have an outgoing edge to each of their subterms. Each symbolic constant and expression is annotated with a program* location $\mathrm{l}(v)$ *that created the term.*

For example, consider the program in Fig. 5.6a. The first line allocates two new symbolic constants x and y of type integer and binds them to the variables `x` and `y`. The second line constructs expressions out of these symbolic constants and the concrete constant `1`. The resulting symbolic heap, shown in Fig. 5.6b, comprises five expressions and three constants. The symbolic constants x and y have locations $\mathrm{l}(x) = \mathrm{l}(y) = 1$, while the expressions have locations $\mathrm{l}(\cdot) = 3$. Most symbolic evaluators use canonicalization (e.g., [56]) to improve sharing of symbolic terms, so only one instance of the terms $(- x\ 1)$ and $(- y\ 1)$ is usually constructed.

Symbolic Evaluation Graph. Where the symbolic heap reflects the flow of data during symbolic evaluation, the *symbolic evaluation graph* captures the control flow. This graph reflects the engine's evaluation strategy—where it explored multiple paths separately, and where those paths were merged together. By analyzing the symbolic evaluation graph, a symbolic profiler can identify candidate bottlenecks with significant branching or merging activity.

Definition 5.2 (Symbolic evaluation graph)**.** *A* symbolic evaluation graph *is a directed acyclic graph* (V, E) *in which each vertex* $s \in V$ *is a state of the program. Each edge* $(s_i, s_j) \in E$ *is a transition between two program states, and is annotated with a* location $\mathrm{l}(s_i, s_j)$ *reflecting the point in the program that caused the transition, and a* guard constraint $\mathrm{guard}(s_i, s_j)$ *reflecting the condition under which the transition was taken.*

As an example, consider again the program in Fig. 5.4a. Different evaluation strategies will produce different symbolic evaluation graphs for this program. In symbolic execution (Fig. 5.4b), each `if` statement will cause execution to diverge into two paths that never meet, and so the definition of `y` (line 5) will execute twice and the assertion (line 7) four times. On the other hand, bounded model checking (Fig. 5.4c) will immediately merge the two paths generated by each `if`, and so both the definition of `y` and the assertion will execute only once, on merged states.

Symbolic Profiling Interface. Most symbolic evaluators create the symbolic heap and evaluation graph only implicitly—the heap is implicit in a canonicalization cache, and the graph in the evaluator's control flow. To enable a symbolic profiler to track the evolution of these data structures, we define a generic *symbolic profiler interface* that engines should implement at important points in symbolic evaluation. A symbolic profiler can construct the symbolic heap and evaluation graph by instrumenting calls to this interface. In practice, these calls are often already implemented by symbolic evaluators, simplifying adoption. Section 5.4.4 describes implementations of the interface in two different symbolic evaluators.

Definition 5.3 (Symbolic profiler interface)**.** *The* symbolic profiler interface *comprises five instrumentation points that a symbolic evaluator should implement to expose profiling data:*

- $\mathsf{new}(x, l)$ *Allocate a fresh symbolic constant named* x *at program location* l*.*
- $\mathsf{new}(op, x_1, \ldots, x_n, l)$ *Allocate a new expression* $op(x_1, \ldots, x_n)$ *at program location* l*, where each* x_i *is a concrete constant or a previously allocated symbolic term.*
- $\mathsf{step}(s_0, \langle g_1, e_1 \rangle, \ldots, \langle g_n, e_n \rangle)$ *Starting from program state* s_0*, evaluate each program expression* e_i *(annotated with a corresponding program location* l_i*) under the guard* g_i*. Return a list of the resulting states* $s_1, \ldots, s_k$*, where* $k \geq n$*.*
- $\mathsf{merge}(s_1, \ldots, s_m, l)$ *Merge the states* $s_1, \ldots, s_m$ *at program location* l *using the evaluator's merging strategy, returning a new list of states* $s'_1, \ldots, s'_j$*, where* $1 \leq j \leq m$*.*

- solve(x, l) *Call a constraint solver at program location l to determine the satisfiability of the expression x.*

The two new calls in the profiler interface construct the symbolic heap. The symbolic evaluator invokes new each time it allocates a new symbolic term—either a fresh symbolic constant or an expression. The profiler adds the corresponding new node to the symbolic heap, with edges to the relevant (immediate) subterms if the new term is an expression.

The step and merge calls in the profiler interface reconstruct the symbolic evaluation graph. The symbolic evaluator calls step to evaluate a set of expressions (e.g., two branches of a conditional) under disjoint and exhaustive guards. It calls merge to merge a set of states, usually at a control-flow join point. For example, at line 4 in Fig. 5.4a, the evaluator invokes step$(s_1, \langle a, 1\rangle, \langle \neg a, \emptyset\rangle)$, which adds the edges $\langle s_1, s_2\rangle$ and $\langle s_1, s_3\rangle$, with the guards a and $\neg a$, to the evaluation graph. From this point, different evaluation strategies result in different calls to the profiler interface. A symbolic execution engine (Fig. 5.4b) never calls merge, instead evaluating each path separately by calling step twice for line 5 (once per path). A bounded model checker (Fig. 5.4c) immediately calls merge$(s_2, s_3, 4)$ to merge the two states at line 4, producing a single new state s_4. In either case, by instrumenting the step and merge calls, a symbolic profiler can reify the (otherwise implicit) evaluation graph.

Finally, the solve call in the interface allows a profiler to determine which parts of the symbolic heap flow to a constraint solver. The symbolic evaluator invokes solve(x, l) whenever it solves a constraint x, either to check the feasibility of a path condition or to discharge a solver-aided query. In the next section, we use this data to analyze the symbolic heap for terms unseen by the solver, which can indicate wasted allocations.

5.4.3 *Analyzing a Symbolic Profile*

Our symbolic profiler, SymPro, analyzes the symbolic heap (Definition 5.1) and evaluation graph (Definition 5.2) in three ways to present suggestions to users: computing *summary statistics* about each procedure in the program; determining *data flow* through the program to identify wasted allocations and work; and *ranking* procedures based on these two analyses to identify the most likely bottlenecks.

Summary Statistics. SymPro computes four summary statistics about each procedure call:

- *Time* is the exclusive wall-clock time spent in the call;
- *Term count* is the number of symbolic terms added to the symbolic heap;
- *Union size* is the sum of the out-degrees of all nodes added to the symbolic evaluation graph;
- *Merge cases* is the sum of the in-degrees of those nodes.

These statistics summarize the key aspects of symbolic evaluation: the time spent in each procedure; the size of the symbolic state allocated; how many times path

splitting (symbolic execution) was performed; and how many times merging (bounded model checking) occurred.

Data Flow. In addition to computing the summary statistics, SymPro uses the symbolic heap and the solve instrumentation to determine which terms in the heap are "used" by the program. To a first approximation, terms in a solver-aided program are not useful if they are never sent to the underlying constraint solver as part of a feasibility check or a solver-aided query. SymPro exploits this observation to produce an analysis of *unused terms* in the program. For each $\mathsf{solve}(x, l)$ call made by the symbolic evaluator, SymPro computes the set of all terms in the symbolic heap that are transitively reachable from x. Any term y that is in none of these transitive closures is unused: it is not part of any constraint sent to the solver.

Unused terms indicate either dead code (terms that were created but never used) or simplification by the symbolic evaluator. For example, consider the following program:

```
1 (define-symbolic* x boolean?) ; add x to the heap
2 (define A (or x (not x)))     ; add ¬x to the heap
3 > A                           ; but simplify x ∨ ¬x
4 #t
5 > (solve (assert A))          ; both x and ¬x unused
```

SymPro would report the terms x and $\neg x$ as unused, because the evaluator simplified them out of the query sent to the solver. Both causes of unused terms represent optimization opportunities: dead code should be removed, while excessive simplification suggests redundancy that a better algorithm or encoding could eliminate.

Ranking. Based on the summary statistics and data flow analysis, SymPro ranks each procedure in the program to suggest the most likely bottlenecks to the user. It first normalizes each statistic (time, term count, union size, merge count, and unused terms) to the range 0–1. Then it assigns each procedure a score by summing the normalized statistics. This score, a number between 0 and the number of statistics, is a simple ranking of which procedures do the most symbolic work. Our case studies in Section 5.5 and evaluation in Section 5.6 show this ranking is highly effective for navigating symbolic profiles. We also experimented with a machine-learned ranking scheme, training a binary classifier (a support vector machine) to identify bottlenecks using some of the benchmarks from Table 5.2 as training data. The resulting classifier had high recall but poor precision, identifying many false positive bottlenecks. For that reason, and because our manual ranking scheme is easier to explain, SymPro uses that scheme as the default.

5.4.4 *Implementation*

We have implemented the symbolic profiler interface (Definition 5.3) in two different symbolic evaluators—a fully featured implementation for Rosette [126,

127], and a proof of concept one for the Jalangi JavaScript analysis framework [117, 118].

Rosette. Our Rosette profiler instruments several key points in Rosette's evaluation engine, most of which are directly analogous to the calls in the profiler interface. To implement the new interface, we instrument Rosette's term creation cache, which performs hash-consing to canonicalize terms. To implement step, we record the creation of *symbolic unions*, which Rosette uses to track the multiple possible values of a variable during symbolic evaluation. Finally, to implement merge and solve, we instrument Rosette's corresponding procedures `merge` and `solver-check`. This instrumentation changes only 21 lines of the Rosette engine implementation. The code for the SymPro analyses comprises 1,000 lines of Racket and 1,400 lines of TypeScript. The Rosette profiler is open-source and integrated into the latest Rosette release [125].

Jalangi. Jalangi [117] uses a symbolic execution engine called MultiSE [118] to provide concolic test generation for JavaScript. We modified MultiSE to implement the symbolic profiler interface as follows. To implement new, we track calls to the constructors of symbolic term objects (strings, numbers, and booleans). MultiSE rewrites JavaScript programs with additional control flow to implement step, and so we track each time a new path is generated from a branch in the program. To instrument merge, we modify MultiSE's implementation of *value summaries*, which are lists of guard-value pairs reflecting the possible values of each variable. Section 5.6.4 presents our results with this proof-of-concept profiler.

5.4.5 *Discussion*

Two kinds of performance issues are outside the scope of symbolic profiling, which focuses on analyzing the behavior of symbolic evaluation. First, if the bottleneck is in constraint solving, a symbolic profiler can report that solving is taking the most time, but it cannot identify the cause of the issue. Second, while bottlenecks in concrete execution can be identified by symbolic profiling (which includes a measure of execution time), they will be ranked below symbolic evaluation bottlenecks because they cause no activity in the symbolic heap and evaluation graph.

Some bottlenecks can be repaired in multiple ways at different locations within a program; when symbolic profiling identifies such a bottleneck, it may not suggest the easiest location to repair. For example, consider this program, with a symbolic boolean input `b` passed to `outer`:

```
1 (define (outer b)
2   (when b
3     (inner)))
4 (define (inner)
5   ...)
```

Suppose the `inner` function has side effects (e.g., mutating global variables) that result in a bottleneck. Symbolic profiling will identify `outer` as the bottleneck, but

the issue could be repaired by modifying either `outer` (to not call `inner` under a symbolic path condition) or `inner` (by mutating less global state). As another example, irregular data representations (Section 5.3.2), such as the one on line 36 of Fig. 5.2, are most easily repaired where the data is constructed, even though symbolic profiling will identify the location the data is used (lines 13–14 of Fig. 5.2) as the bottleneck.

5.5 ACTIONABILITY CASE STUDIES

To demonstrate that SymPro produces *actionable* profiles, we performed a series of case studies on real-world Rosette programs. We collected a suite of benchmarks by performing a literature survey of all papers citing Rosette [126]. This suite, shown in Table 5.1, comprises all 15 tools that were open source (or that the authors made available to us) and that ran on the latest Rosette release.

We applied SymPro to each benchmark and used it to identify 8 performance bottlenecks summarized in Table 5.2. This section presents our results, with three in-depth case studies and brief overviews of four other findings. In each case study, we highlight a bottleneck found by SymPro, relate it to the anti-patterns of Section 5.3, and present repairs. Six of the eight bugs we found were in code bases with which we were not previously familiar. Section 5.6 evaluates SymPro against our other design criteria, *explainability* and *generality*.

5.5.1 *File System Crash-Consistency*

Ferrite [23] is a tool for reasoning about crash safety of programs running on modern file systems, which offer only weak consistency semantics. It consists of a verifier and a synthesizer. Given a *litmus test* program (i.e., a small, straight-line sequence of system calls), and a specification of crash safety for it, the verifier checks whether the program satisfies the safety specification under the relaxed semantics of a file system such as ext4, even in the face of crashes. If not, the synthesizer attempts to repair the program by inserting barriers (i.e., calls to `fsync`).

Ferrite represents files as a backing store (a list of bytes) together with the length of the file:

```
1 (struct file (contents length) #:transparent)
2 (define BLOCK_SIZE 4096)
3 (define F (file (make-list BLOCK_SIZE #x00) 0))
```

To model a write to the file `F`, which persists only if the system does not crash, Ferrite introduces a symbolic boolean value *crash?* to represent a non-deterministic crash:

```
1 (define N 2)
2 (define-symbolic* crash? boolean?)
3 (unless crash?          ; If not crashed
4   (match-define (file contents length) F)
5   (define new-contents  ; write 0x1 to first N bytes
6     (append (make-list N #x01) (drop contents N)))
7   (set! F (file new-contents (+ length N))))
```

Table 5.1: Rosette benchmarks used in our evaluation. LoC is lines of code. Performance results show the overhead of SymPro's analysis as the average of five runs; 95% confidence intervals for overhead are < 5 pp.

		Time		Peak Memory	
Benchmark	LoC	Time (sec)	Slowdown	Memory (MB)	Overhead
Bagpipe [135]	3317	16.1	51.1%	314	30.9%
Bonsai [40]	641	55.1	22.1%	341	128.7%
Cosette [44]	2709	12.8	7.4%	296	17.6%
Ferrite [23]	350	21.5	2.7%	690	5.0%
Fluidics [137]	145	17.7	5.7%	198	18.9%
GreenThumb [108]	934	2358.5	0.1%	2258	0.0%
IFCL [126]	574	96.4	0.7%	248	28.7%
MemSynth [26]	3362	24.0	45.7%	349	33.5%
Neutrons [106]	37317[†]	45.3	14.8%	1702	98.4%
Nonograms [34]	6693	15.1	3.1%	300	18.6%
Quivela [14]	5946	78.6	1.4%	496	20.0%
RTR [76]	2007	374.6	12.6%	822	35.5%
SynthCL [126]	3732	27.7	61.2%	445	133.2%
Wallingford [28]	3866	7.9	2.4%	618	86.3%
WebSynth [126]	2057	14.2	47.7%	467	122.8%

[†] Includes a 36,847-line Racket file automatically generated from the software being verified, which SymPro must instrument.

Table 5.2: Summary of performance bottlenecks found by applying SymPro to the benchmarks in Table 5.1, together with the speedups obtained by repairing them.

Program	Anti-Pattern	Description	Speedup
Bonsai	Irregular representation	Shape of tree data structure is enumerated multiple times (§5.5.4)	1.35×
Cosette	Missed concretization	Possible table sizes are enumerated in a nested loop (§5.5.2)	$> 6\times$[†]
	Algorithmic mismatch	Inefficient reduction builds a complex intermediate list (§5.5.2)	75×
Ferrite	Missed concretization	Length of an array is merged despite few feasible values (§5.5.1)	24×
Fluidics	Irregular representation	Grid data structure implemented with nested mutable vectors (§5.5.4)	2×
Neutrons	Irregular representation	Log of possible paths is maintained symbolically (§5.5.3)	290×
Quivela	Missed concretization	Object references are merged and obscure dynamic dispatch (§5.5.4)	29×
RTR	Algorithmic mismatch	Unnecessary fold over list of symbolic length (§5.5.4)	6×

[†] Without the repair, Cosette does not terminate within one hour.

To check the safety specification, Ferrite retrieves the final contents of the file:

```
(define cnts (take (file-contents F) (file-length F)))
(assert (or (equal? cnts '()) (equal? cnts '(1 1))))
```

This implementation is sufficient to verify crash safety at small block sizes (e.g., 32 bytes). But since many crash consistency bugs rely on boundary conditions around the size of disk blocks, Ferrite sets `BLOCK_SIZE` to a realistic value for a modern device (here, 4 kB). With this block size, even simple litmus tests cannot be verified (or repaired) in reasonable time.

Identifying the Bottleneck. SymPro identifies the call to `take` in the final step above as the source of poor performance. It ranks `take` high based on its large number of created symbolic terms and the fact that almost none of those terms reach the solver. In contrast, a time-based profiler ranks the subsequent `equal?` call as the hottest method.

Diagnosing the Bottleneck. The root cause of this issue is a *missed concretization*. Rosette merges the second input to `take`, representing the length of the file, into a symbolic term of the form $(\mathrm{ite}\ \textit{crash?}\ 0\ 2)$. When `take` receives a symbolic length argument, it performs symbolic execution, generating one path per potential length of the returned list. Since the input list `(file-contents F)` has length `BLOCK_SIZE` $= 4096$, the `take` call generates 4097 distinct paths, each with a path condition of the form $(\mathrm{ite}\ \textit{crash?}\ 0\ 2) = n$ for $0 \leq n \leq 4096$. All but two of these paths are infeasible.

Repairing the Bottleneck. To repair the program, we recover the feasible concrete values for the file's length in two steps. First, we remove the `#:transparent` annotation from the definition of the `file` data type, to prevent structural (field-wise) merging of files. Instead, Rosette will use symbolic unions (Section 5.3.1) to merge files. Second, we use Rosette's `for/all` annotation to evaluate the `take` call separately for each value in the symbolic union `F`:

```
(define contents
  (for/all ([f F])
    (take (file-contents f) (file-length f))))
```

The `for/all` annotation is Rosette's *symbolic reflection* facility [126], which allows programmers to control path splitting and merging. By default, Rosette evaluates the arguments to `take` first, merges the results, and then applies `take` once to the merged value. The `for/all` annotation tells Rosette to instead apply `take` to each possible value of `F` separately and then merge the results. This repair speeds up Ferrite by 24×, enabling it to replicate—in just a few minutes—a complex ext4 delayed allocation bug in Google Chrome [22].

5.5.2 *SQL Query Equivalence Verification*

Cosette [43, 44] is an automated prover for deciding the equivalence of two SQL queries. It uses Rosette to search for small counterexamples to equivalence, and Coq to construct proofs of equivalence if no counterexample is found.

Cosette's counterexample finder works by constructing a symbolic representation of a SQL table as a bag of tuples. Both the multiplicity of each tuple and its constituent elements are symbolic values. To execute a query against a table, Cosette constructs a new table in which the multiplicity of each tuple reflects the semantics of the query. For example, the result of executing the query `SELECT A FROM table WHERE C="a"` on a table is another table:

A	B	C	#		A	#
e_0	e_1	e_2	c_0	$\Longrightarrow$	e_0	**if** (= e_2 "a") c_0 0)
e_3	e_4	e_5	c_1		e_3	**if** (= e_5 "a") c_1 0)

To check if two queries are equivalent, Cosette executes each query on the same symbolic table, constructs a constraint asserting the two resulting tables are different, and solves this constraint using Rosette. Cosette makes extensive use of advanced Rosette features, including `eval` of dynamically generated code, making manual reasoning about performance particularly challenging.

A recent change to Cosette adjusted its encoding of SQL `WHERE` clauses to accommodate a richer subset of SQL's filter syntax. Previously, Cosette implemented filtering by removing the appropriate tuples from the bag; the change instead filters by setting those tuples' multiplicities to zero. After making this change, a Cosette benchmark that previously returned in under 15 seconds no longer returned within an hour. Our initial investigation showed the SMT solver was never called, suggesting the bottleneck was in symbolic evaluation, but offered no further details.

Identifying the Bottleneck. To identify the source of this bottleneck, we used SymPro's support for streaming profile data during execution. The streaming profiler applies the analyses in Section 5.4 incrementally as the symbolic heap and symbolic evaluation graph evolve, and periodically sends the resulting data to the profiler interface. For Cosette, the profiler implicated the following call to the `filter` function:

```
1 (map (lambda (t)
2        (sum (filter (lambda (r) (eq? t r)) table)))
3      table)
```

The profiler ranked these `filter` calls far above any other calls in the program due to their high number of new terms allocated on the symbolic heap and large numbers of merges in the symbolic evaluation graph.

Diagnosing the Bottleneck. This bottleneck is caused by a combination of two issues, a *missed concretization* and an *algorithmic mismatch*, which manifest as two distinct sources of path explosion.

The missed concretization is due to `table` being a symbolic union, reflecting the table's value along several control-flow paths generated by symbolic execution. The nested use of `table` thus creates quadratic path explosion—for each path in `table` explored when calling `map`, the evaluator explores every path in `table` when evaluating the inner `filter`.

The algorithmic mismatch is due to using `filter` to create an intermediate list just to `sum` its contents. The predicate used by `filter` depends on symbolic state, and so there are $O(2^N)$ paths for the return value of `filter`, as in the toy example from Fig. 5.1. The `sum` procedure must then run once for each such path.

Repairing the Bottleneck. An easy repair for the missed concretization is to apply symbolic reflection:

```
(for/all ([table table])
  (map (lambda (t)
         (sum (filter (lambda (r) (eq? t r)) table)))
       table))
```

Here, the `for/all` evaluates its body once for each path in `table`. During each such evaluation, `table` is bound to a single concrete value rather than a union, avoiding the first source of path explosion. With this repair, the problematic benchmark completes within 10 minutes—better than non-termination but still worse than the original version of Cosette.

To repair the algorithmic mismatch, we avoid building the intermediate list with `filter`. Instead, the procedure passed to `map` performs a fold over `table` to sum the values that satisfy the `filter` predicate. With this additional repair, the problematic benchmark completes in 8 seconds—faster than even the original Cosette implementation. We reported the regression to the Cosette developers, and they accepted our patch.

5.5.3 *Safety-Critical System Verification*

Neutrons [106] is a tool for verifying the safety of a radiotherapy system in clinical use. The system is controlled by a large program written in the EPICS dataflow language [55]. Neutrons provides a symbolic interpreter for EPICS programs, and a verifier (built with Rosette) to check that EPICS programs satisfy key safety properties. The Neutrons verifier is used for active development of the system's software, so its performance is important for developer use.

Identifying and Diagnosing the Bottleneck. We used SymPro to profile the Neutrons symbolic interpreter, and found a bottleneck with the interpreter's tracing feature. As the interpreter executes an EPICS program, it records each executed instruction in a trace—a list of executed instructions—which is used to visualize counterexamples:

```
(define (record-trace msg)
  (set! trace (append trace (list msg))))
```

However, since this call is made with a symbolic path condition, Rosette must merge the new and existing values of `trace` when performing the mutation. This leads to excessive path creation and merging, since Rosette will track each potential length of `trace` separately by symbolic execution, and the length of `trace` depends upon the execution path. In essence, `trace` has an *irregular representation*. SymPro identifies this tracing procedure as the key bottleneck.

Repairing the Bottleneck. To improve this program, we observe that tracking the shape of the trace is unnecessary for counterexample visualization. For each executed instruction, we need only record the path condition that was true when the instruction executed, together with the instruction:

```
1 (define (record-trace msg)
2   (raw-set! trace (append trace (list (cons (pc) msg)))))
```

Here, `(pc)` retrieves the current path condition, and `raw-set!` is Racket's unlifted implementation of **`set!`** that overwrites `trace` without any merging. The `trace` is now a list of every instruction executed by *any* possible interpretation of the EPICS program. When using this trace to visualize a counterexample, we simply hide any instruction whose corresponding path condition is not satisfied by the counterexample. This program transformation—which essentially adjusts the `trace` list to always have a concrete length—improves Neutrons' verification performance by 290× on a representative example. We reported this issue to the Neutrons developers, and they accepted our patch.

5.5.4 *Other Findings*

Our other findings in Table 5.2 include examples of all three anti-patterns presented in Section 5.3.2.

Type System Soundness Checking. Bonsai [40] is a synthesis-based tool for checking the soundness of type systems. It uses a novel tree representation for type checking, and has been used to replicate a soundness bug in the Scala type system. We applied SymPro to Bonsai and found two *irregular representation* issues. First, Bonsai represents trees as nested lists; since the trees have unknown size, these lists are merged into a symbolic union. When the tree is used multiple times during the same type checking call, the symbolic evaluator enumerates the members of this union once per use and merges the results. Instead, we used Rosette's `for/all` facility to perform this enumeration only once, as done in the Cosette case study. Second, each (recursive) type checking step can return either a subtree or a boolean (in case of failure), which Rosette will always merge into a symbolic union due to their different types. Instead, we used multiple return values to separate the returned boolean failure flag from the returned subtree. Together, these changes improved Bonsai's performance by 35% when checking the Scala type system.

Cryptographic Protocol Verification. Quivela [14] is a tool for verifying the security of cryptographic protocols. It takes as input an implementation and a specifi-

cation of a cryptographic protocol, along with a series of refinement steps between them, and checks that each refinement is valid. We applied SymPro to Quivela and identified a *missed concretization* issue. Quivela represents protocols in a simple object-oriented language in which all method calls are virtual; each object can store references to other objects, which Quivela represents as integer addresses. Because these references are integers, the symbolic evaluator's default strategy is to merge them. But the merged references obscure the targets of virtual method calls forcing the engine to evaluate many infeasible paths, as in the Ferrite case study. We modified Quivela to instead track references concretely, by wrapping references into an opaque structure type that cannot be merged. This change improved Quivela's verification performance by up to 29× on small benchmarks, and allowed it to quickly verify larger protocols that previously caused out-of-memory failures.

Microfluidics Control Synthesis. Fluidics [137] is a prototype tool for synthesizing programs that control a digital microfluidics array, used for executing biological wet-lab protocols. It takes as input the initial arrangement of samples on the array, and the desired final arrangement (potentially including mixtures of the samples), and synthesizes a series of movement and mixing instructions that produce the desired outcome. We applied SymPro to Fluidics and identified an *irregular representation* issue. Fluidics represents the state of the array as a two-dimensional vector of vectors, indexed by y and then x coordinates. However, this nested structure makes updates to the array expensive: because vectors are mutable data structures, the inner vector must be duplicated for each update to correctly track later mutations. Replacing the nested data structure with a flat one-dimensional vector improves Fluidics' performance by 2×, allowing it to synthesize more complex control programs and reason about larger microfluidics arrays.

Refinement Type Checker for Ruby. RTR [76] is a type checker for a new refinement type system for Ruby. It takes as input a Ruby program translated to a Rosette-based intermediate verification language, and checks that user-specified refinement types hold in the (translated) program. The RTR verification language reflects Ruby's object structure and control-flow constructs. We applied SymPro to RTR and identified an *algorithmic mismatch* issue in the way RTR initializes new Ruby objects. In Ruby, an array initialization supplies a length together with an anonymous function (a "block") defining the value at each index:

```
1 Array.new(5){ |i| i*2 }
2 #=> [0, 2, 4, 6, 8]
```

RTR represents arrays as a pair of a vector (holding the array's contents) and an integer (holding the array's actual length). To support bounded verification, array lengths can be symbolic. RTR's array initialization creates a separate vector/integer pair for each possible length of the array, taking quadratic time. SymPro identifies the array initialization procedure as the bottleneck. We repaired this issue by initializing a concrete vector of length equal to the upper bound on the

symbolic length; since RTR already tracks each list's length separately, the extraneous elements can simply be ignored. This repair improved RTR's performance on its slowest benchmark (Matrix) by 6×, from 6.1 minutes to 61 seconds, and reduces its peak memory usage by 3×. RTR's developers accepted our patch.

5.6 EXPLAINABILITY, GENERALITY, AND PERFORMANCE

To evaluate the performance, explainability, and generality of symbolic profiling, we sought to answer four research questions:

1. Is the overhead of symbolic profiling reasonable for development use?
2. Is the data collected by SymPro necessary for correctly identifying bottlenecks?
3. Are programmers more effective at identifying bottlenecks with SymPro?
4. Is SymPro effective at profiling different symbolic evaluation engines?

The first two questions address the key performance aspects of SymPro—runtime overhead and the necessity of the collected data for generating actionable feedback. The third question evaluates the explanatory power of SymPro's profiles. The fourth question assesses the generality of our approach. We use the Rosette profiler to investigate the first three questions, and the Jalangi profiler for the fourth. We find positive answers to all four questions.

5.6.1 *Is the overhead of symbolic profiling reasonable for development use?*

Table 5.1 shows the time and memory overheads for SymPro on a collection of real-world Rosette programs. All results were collected using an AMD Ryzen 7 1700 eight-core processor at 3.7 GHz and 16 GB of RAM, running Racket v6.12. For each benchmark we report the average overhead across five runs; 95% confidence intervals are below 5 percentage points for all overhead results.

Overall, SymPro slows applications by 0.1%–61.2% (geometric mean 16.9%), and increases peak memory use by 0.0%–133.2% (geometric mean 45.6%). These overheads are reasonable for development use, and are better than other tracing-based profiling tools. For example, the Racket version of profile-guided metaprogramming [29] averages 4–12× slowdown, and input-sensitive profiling [49] averages 30× slowdown for C programs. The highest overheads occur for benchmarks with many short-lived recursive calls. It would be possible to implement a sampling-based profiler if this overhead were to become unacceptable.

5.6.2 *Is the data collected by SymPro necessary for correctly identifying bottlenecks?*

To understand the importance of the data SymPro gathers, we performed a sensitivity analysis using all benchmarks in which we identified new bottlenecks (Table 5.2), as well as a collection of benchmarks with previously known bottlenecks.

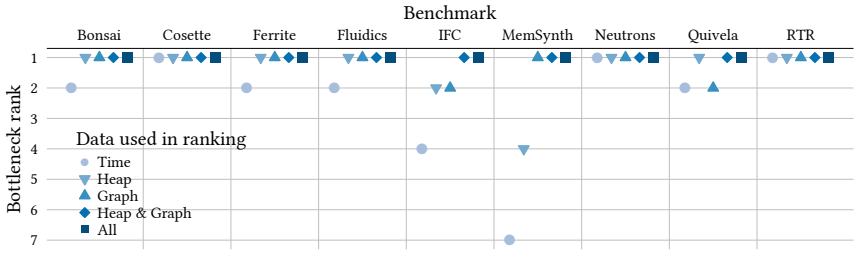


Figure 5.7: Sensitivity of profiler rankings to the data sources used in ranking. Each point is the rank of the bottleneck function in a profile of the program when using the specified data in the ranking function. Benchmarks are those in which we found new bottlenecks (Table 5.2) or had previously known bottlenecks.

For each benchmark, we manually investigated its profile to identify a single procedure we believe should be ranked as the primary cause of poor performance. We then varied the data available to SymPro, giving it access to only wall-clock time, only the symbolic heap, only the symbolic evaluation graph, or combinations of the three components.

Fig. 5.7 shows the results of the sensitivity experiment. For each benchmark, the y-axis measures the ranking of the known bottleneck when using only the specified source of profiling data. These results have three key highlights. First, timing data ● alone (i.e., the time spent in each procedure) identifies only three of nine bottlenecks. Second, no single data source is sufficient to identify the key bottleneck in every benchmark. While the symbolic heap ▼ and evaluation graph ▲ are each more effective than time alone, both are required ♦ to correctly rank all bottlenecks. Third, once both the symbolic heap and evaluation graph are available, including timing data ■ does not improve the quality of the rankings. However, SymPro still includes timing data in rankings, to help profile the parts of programs that do not perform symbolic evaluation.

5.6.3 *Are programmers more effective at identifying bottlenecks with SymPro?*

To help understand how effective SymPro is in real-world use, we conducted a small user study with Rosette programmers. Our study had eight graduate student participants, who each had previous Rosette experience ranging from "a few hours" to multiple published papers using Rosette. We first provided each participant a short tutorial on how to use both SymPro and existing Racket performance tools (the `time` form and the built-in Racket profiler [18]). We then asked each participant to study four benchmarks—three realistic solver-aided tools and a simple calculator program—and identify (but not repair) the primary performance bottleneck. For each benchmark, each participant was randomly assigned to either the baseline group (which had access to any tool except SymPro) or the SymPro group (which had access to SymPro as well). To help control for learning effects, each participant saw the benchmarks in a random order, and had at most 20 minutes to analyze each benchmark.

Quantitative Results. Fig. 5.8 shows the average time taken for users to identify the performance issue in each benchmark. SymPro improves the identification

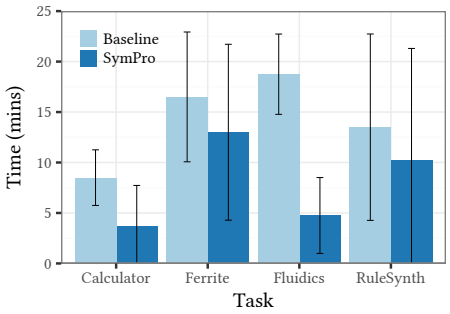


Figure 5.8: Average time taken for users to identify the performance issue in four benchmarks, with or without SymPro. Error bars are 95% confidence intervals for $n = 4$ users.

time for every benchmark, though due to the small sample size ($n = 4$ for each treatment), we do not claim statistical significance. There were 6 cases where a user in the baseline group failed to find the issue in a benchmark within the 20 minutes available; no users in the SymPro group ever reached this time limit.

Qualitative Observations. Given the limited size of our study, its main value was in the qualitative observations reported by the participants. Users with access to SymPro said it gave "insight into what Rosette is actually doing" which they lacked from other tools. One user said that SymPro was "extremely useful for investigating a performance issue," and that they could "see how I would optimize my own code using the [symbolic] profiler." Users generally reported they thought the symbolic profiler would be even more successful when run against their own code, because they "know what to ignore."

We found that users were most successful when using SymPro to conduct an initial investigation. SymPro's data analysis generally directed users to fruitful locations in the code to inspect more quickly than either manual exploration or analysis by existing performance tools. While we did not require users to identify potential repairs to the performance issues they found, they were more willing and able to do so voluntarily when using SymPro, suggesting a better understanding of the code.

5.6.4 *Is SymPro effective at profiling different symbolic evaluation engines?*

In addition to the Rosette profiler evaluated above, we also built a prototype symbolic profiler for the Jalangi dynamic analysis framework [117], as Section 5.4.4 describes. We applied the profiler to the three slowest publicly available benchmarks reported by Sen et al. [118]. For each benchmark, we ran both the symbolic profiler and a traditional time-based profiler to identify hotspots, and compared the results. The symbolic profiler added only negligible overhead ($< 1\%$).

Red-Black Tree. The red-black tree benchmark implements a self-balancing binary search tree with integer keys. The symbolic version of the benchmark inserts five unknown, symbolic keys into the binary search tree. The time-based profiler

identifies an internal key-comparison function as the only hotspot in the benchmark. But the symbolic profiler helps pinpoint why the key comparison is slow: it identifies the tree's `insert` procedure as being responsible for the creation of most symbolic state (due to branching), and reports that the key comparison creates very large terms on the symbolic heap. Guided by this profile, we modified the key comparison function to be branch-free, which improved the benchmark's performance by 2×. The profiler also suggests that path pruning with the SMT solver is ineffective on this benchmark: most paths are generated by the key-comparison function, but they are always feasible. Surprisingly, we found that the tree's rebalancing operations were not the sources of expensive symbolic operations.

Calculator Parser. The calculator parser benchmark implements a simple grammar for arithmetic expressions, and attempts to parse an expression from a symbolic input. The time-based profiler identifies the function `getsym`, which generates the next character of symbolic input, as the bottleneck. The symbolic profiler instead identifies `accept` and its callers, which interpret the output of `getsym` and form the core of the parser. In particular, the symbolic profiler identifies the grammar's "factor" production as being a bottleneck due to a large number of branches. Inspecting this function, we found most branches perform similar work, and so we refactored it to move that work outside of the branches. This small refactoring improved the benchmark's performance by 1.8×.

Binary Decision Diagram. The binary decision diagram (BDD) benchmark constructs a BDD with three unknown, symbolic operations (that can be either ∧ or ∨), each of which operates on two unknown, symbolic operands. The time-based profiler can only identify the top-level driver function of this benchmark as a potential hotspot. The symbolic profiler is more effective, identifying an internal hash table and the BDD `put` operation as the sources of symbolic complexity. We replaced the hash table with a linked list, improving performance by 10%. With this repair, the profiler now identifies `get` as the bottleneck instead of `put` (as we would expect, since `get` must now search the list). While a linked list is clearly less efficient for concrete code, it is more amenable to verification, and so this transformation may be preferable for verifying *clients* of the BDD library. In general, we expect SymPro to be useful for developing *models* of libraries and frameworks, which are simplified implementations intended for verification purposes and used by automated verification tools [36].

5.7 RELATED WORK

Optimizing Symbolic Evaluation. A high-performance symbolic evaluation engine must make good decisions about when to merge states from different paths. Query count estimation (QCE) is a heuristic for estimating the number of paths that will be created by merging at a given program point [82]. A QCE engine merges states only if the "hot" variables in each branch are the same, or are already symbolic. The "hot" variables are identified heuristically; a variable v is hot if many additional paths are likely to be generated by making v symbolic. In essence, QCE

is a heuristic for predicting the shape of the symbolic evaluation graph. SymPro, in contrast, tracks the shape of the graph and lets the programmer use this information to guide symbolic evaluation.

In addition to improving the performance of symbolic evaluation at the engine level (through better strategies and encodings), prior work has also proposed making improvements at the program level. Wagner, Kuznetsov, and Candea [132] advocate for a special compiler optimization mode tuned for emitting code amenable to symbolic execution, avoiding program transformations that exhibit poor behavior under symbolic evaluation. Cadar [35] presents a collection of program transformations (both semantics-preserving and -altering) designed to enable scalable symbolic execution. SymPro is an ideal companion to these approaches: when automated optimizations fail (as Cadar shows is often the case), a profiler can help identify potential bottlenecks for manual repair.

Profiler-Aided Development. Recent research has focused on how profiling information should be integrated into development workflows. For example, the *optimization coach* [16] feature of Racket communicates successful and failed compiler optimizations to programmers, while profile-guided meta-programming [29] integrates profiling data into the source-to-source transformations in Racket's macro system. One important property of a profiler is that its advice must be *actionable*: optimizing the functions it suggests as hot should improve execution time. The Coz causal profiler [51] achieves this by performing experiments at run time. To determine if a function f is hot, Coz simulates optimizing f by artificially slowing down every other function in the program. We took inspiration from all three of these techniques when designing SymPro.

Interactive Profiling. Ammons et al. [15]'s Bottlenecks tool is an interactive interface for profile data. Profilers implement a common interface defined by Bottlenecks, which then layers a command-line user interface on top of the generated data. Through that interface, Bottlenecks suggests interesting profile points using navigation heuristics that skip over uninteresting data; for example, procedures with little exclusive time are likely less interesting than their callees, so navigation "zooms" over these points. Ammons et al. used Bottlenecks to find 14 performance issues in IBM's WebSphere Application Server, and improve its throughput by 23%. SymPro's user interface (Fig. 5.3) and its common symbolic evaluator interface (Definition 5.3) both take influence from Bottlenecks.

5.8 CONCLUSION

Symbolic profiling is a new approach to identifying and diagnosing performance bottlenecks in programs under symbolic evaluation. Symbolic profiling makes explicit the key resources—the symbolic heap and evaluation graph—that programmers must manage to create performant solver-aided applications. These resources form a new performance model of symbolic evaluation that is actionable, explainable, and general. Our case studies show that symbolic profiling produces actionable profiles. Guided by these profiles, we identified, diagnosed, and re-

paired performance bottlenecks in published, state-of-the-art solver-aided tools, obtaining orders-of-magnitude speedups. Our experiments show that symbolic profiles have high explanatory power, helping programmers understand what the symbolic evaluator is doing, and that our profiling approach generalizes to different symbolic evaluation engines. As programmers increasingly apply solver-aided automation to new domains, symbolic profiling can help them more quickly reach the scale they need to solve real-world problems.

6

CONCLUSION

The pervasiveness of computer systems demands new techniques to ensure their reliability. This dissertation argues that the solution to this challenge lies in automated programming tools, and in particular in the ability to build specialized automated tools for each new problem domain. We have demonstrated the potential of such specialized tools by building one for memory consistency models (Chapter 3); the resulting tool found undocumented issues in two major computer architectures. To aid in building such tools, we have introduced metasketches (Chapter 4), a new abstraction for program synthesis tools, and shown that they can be used to solve synthesis problems that other state-of-the-art tools cannot. Finally, to help programmers scale automated tools, we developed symbolic profiling (Chapter 5), a technique for identifying and diagnosing performance bottlenecks in automated programming tools that has been used to speed up state-of-the-art projects by orders of magnitude. Together, these contributions show that new abstractions and tools can empower programmers to build specialized automated programming tools that ensure software reliability.

But many challenges still remain. Our work on MemSynth benefited from existing DSLs for specifying memory models. In general, much of the work of developing a synthesis tool is spent on DSL design. Helping programmers design new DSLs that are sufficiently expressive while remaining amenable to automation is an open problem [33]. Similarly, the new abstractions and techniques this dissertation develops still require human insight to use— For example, designing a good metasketch requires understanding how synthesis algorithms work, while interpreting the results of a symbolic profile requirse knowledge of possible repairs. In both cases, we have provided design patterns to help programmers use these tools, but ideally this process should be more automated. We should be able to automatically design a good metasketch for a particular synthesis problem; there are some early results suggesting this task can be automated [73, 101], but more work is needed to automatically identify *good* metasketches. Similarly, symbolic profiling should be able to automatically suggest repairs to an automated tool to improve its scalability. As this dissertation shows, well-designed automation and the right abstractions can help bring new advanced programming tools to more domains and a wider range of programmers.

BIBLIOGRAPHY

[1] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In: *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*. Seattle, WA, USA, June 1990, pages 2–14 (cited on page 43).

[2] Jade Alglave. A formal hierarchy of weak memory models. In: *Form. Methods Syst. Des.* 41.2 (2012), pages 178–210 (cited on pages 40, 44).

[3] Jade Alglave. Modeling of Architectures. In: *Advanced Lectures of the 15th International School on Formal Methods*. 2015 (cited on page 35).

[4] Jade Alglave, Mark Batty, Alistair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU concurrency: weak behaviors and programming assumptions. In: *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Istanbul, Turkey, Mar. 2015, pages 577–591 (cited on pages 16, 22).

[5] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In: *Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP)*. Savannah, GA, USA, Jan. 2009, pages 13–24 (cited on pages 15, 36, 43).

[6] Jade Alglave and Luc Maranget. *The Phat Experiment*. `http://diy.inria.fr/phat/`. 2010 (cited on pages 36, 38).

[7] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In: *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*. Edinburgh, United Kingdom, July 2010, pages 258–272 (cited on pages 15–18, 21, 22, 25–27, 36–38, 40, 43, 44).

[8] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. In: *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Saarbrücken, Germany, Mar. 2011, pages 41–44 (cited on page 36).

[9] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: modelling, simulation, testing, and data mining for weak memory. In: *ACM Trans. Program. Lang. Syst.* 36.2 (2014) (cited on pages 16, 21, 25, 35, 42–44).

[10] Rajeev Alur, Rastislav Bodik, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shamwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided

synthesis. In: *Dependable Software Systems Engineering*. Volume 40. 2015 (cited on pages 49, 54, 56, 66–68).

[11] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In: *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Portland, OR, USA, Oct. 2013, pages 1–8 (cited on pages 7, 49, 50, 68, 74).

[12] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. The second competition on syntax-guided synthesis. In: *Proceedings of the 4th Workshop on Synthesis (SYNT)*. San Francisco, CA, USA, July 2015, pages 3–26 (cited on page 67).

[13] Rajeev Alur and Milo M. K. Martin. Personal communication. July 2016 (cited on page 41).

[14] Amazon Web Services. *Quivela*. 2018. URL: `https://github.com/awslabs/quivela` (cited on pages 97, 101).

[15] Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. Finding and removing performance bottlenecks in large systems. In: *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*. Oslo, Norway, June 2004, pages 170–194 (cited on page 107).

[16] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. Optimization coaching: optimizers learn to communicate with programmers. In: *Proceedings of the 27th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Tuscon, AZ, USA, Oct. 2012, pages 163–178 (cited on page 107).

[17] Domagoj Babić and Alan J. Hu. Calysto: scalable and precise extended static checking. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. Leipzig, Germany, May 2008, pages 211–220 (cited on pages 3, 85).

[18] Eli Barzilay. *Profile: Statistical Profiler*. http://docs.racket-lang.org/profile/. 2017 (cited on pages 82, 104).

[19] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In: *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. St. Petersburg, FL, USA, Jan. 2016, pages 634–648 (cited on page 43).

[20] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Wjark Weber. Mathematizing C++ concurrency. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Austin, TX, USA, Jan. 2011, pages 55–66 (cited on pages 22, 43).

[21] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In: *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Amsterdam, The Netherlands, Mar. 1999, pages 193–207 (cited on pages 78, 85).

[22] Nicolas Boichat. *Issue 502898: ext4: Filesystem corruption on panic*. `https://code.google.com/p/chromium/issues/detail?id=502898`. June 2015 (cited on page 98).

[23] 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 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, USA, Apr. 2016, pages 83–98 (cited on pages 8, 96, 97).

[24] James Bornholt and Emina Torlak. Finding code that explodes under symbolic evaluation. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), 149:1–149:26 (cited on pages 3, 77).

[25] James Bornholt and Emina Torlak. *Ocelot*. 2017. URL: `https://jamesbornholt.github.io/ocelot` (cited on pages 2, 17).

[26] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Barcelona, Spain, June 2017, pages 467–481 (cited on pages 2, 15, 97).

[27] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In: *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. St. Petersburg, FL, USA, Jan. 2016, pages 775–788 (cited on pages 3, 35, 47).

[28] Alan Borning. Wallingford: toward a constraint reactive programming language. In: *Proceedings of the Constrained and Reactive Objects Workshop (CROW)*. Málaga, Spain, Mar. 2016 (cited on page 97).

[29] William J. Bowman, Swaha Miller, Vincent St-Amour, and R. Kent Dybvig. Profile-guided meta-programming. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, USA, June 2015, pages 229–239 (cited on pages 103, 107).

[30] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, Mar. 2014, pages 239–254 (cited on page 78).

[31] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In: *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*. Princeton, NJ, USA, July 2008, pages 107–120 (cited on page 38).

[32] Jabob Burnim, Koushik Sen, and Christos Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In: *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Saarbrücken, Germany, Mar. 2011, pages 11–25 (cited on page 38).

[33] Eric David Butler. Automatic Generation of Procedural Knowledge Using Program Synthesis. PhD thesis. University of Washington, 2018 (cited on pages 47, 109).

[34] Eric Butler, Emina Torlak, and Zoran Popović. Synthesizing interpretable strategies for solving puzzle games. In: *Proceedings of the 12th International Conference on the Foundations of Digital Games (FDG)*. 10. Hyannis, MA, USA, Aug. 2017 (cited on pages 8, 97).

[35] Cristian Cadar. Targeted program transformations for symbolic execution. In: *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy, Aug. 2015, pages 906–909 (cited on page 107).

[36] 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 Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA, Dec. 2008, pages 209–224 (cited on pages 85, 106).

[37] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. In: *Communications of the ACM* 56.2 (2013), pages 82–90 (cited on page 86).

[38] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In: *Proceedings of the 28th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Indianapolis, IN, USA, Oct. 2013, pages 33–52 (cited on page 54).

[39] Luca Cardelli, Milan Češka, Martin Fränzle, Marta Kwiatkowska, Luca Laurenti, Nicola Paoletti, and Max Whitby. Syntax-guided optimal synthesis for chemical reaction networks. In: *Proceedings of the 29th International Conference on Computer Aided Verification (CAV)*. Heidelberg, Germany, July 2017, pages 375–395 (cited on page 47).

[40] Kartik Chandra and Rastislav Bodik. Bonsai: synthesis-based reasoning for type systems. In: *Proc. ACM Program. Lang.* 2.POPL (Jan. 2018), 62:1–62:34 (cited on pages 88, 97, 101).

[41] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodík. Angelic debugging. In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. Honolulu, HI, USA, May 2011, pages 121–130 (cited on page 10).

[42] Swarat Chaudhuri, Martin Clochard, and Armando Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. San Diego, CA, USA, Jan. 2014, pages 207–220 (cited on pages 3, 75).

[43] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. *Cosette*. 2017. URL: `http://github.com/uwdb/Cosette` (cited on page 99).

[44] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: an automated prover for SQL. In: *Proceedings of the 8th Biennial Conference on Innovative Data Systems (CIDR)*. Chaminade, CA, USA, Jan. 2017 (cited on pages 97, 99).

[45] Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani, and Cristian Stenico. Satisfiability modulo the theory of costs: foundations and applications. In: *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Paphos, Cyprus, Mar. 2010, pages 99–113 (cited on page 76).

[46] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In: *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Barcelona, Spain, Mar. 2004, pages 168–176 (cited on pages 3, 85).

[47] Lori A. Clarke. A system to generate test data and symbolically execute programs. In: *IEEE Transactions on Software Engineering* 2.3 (1976), pages 215–222 (cited on pages 3, 78, 85).

[48] Compaq. *Alpha Architecture Reference Manual*. 4th. 2002 (cited on page 43).

[49] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. Input-sensitive profiling. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China, June 2012, pages 89–98 (cited on pages 89, 90, 103).

[50] James Crawford, Matthew L. Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In: *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR)*. Cambridge, MA, USA, Nov. 1996, pages 148–159 (cited on page 34).

[51] Charlie Curtsinger and Emery D. Berger. Coz: finding code that counts with causal profiling. In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, USA, Oct. 2015, pages 184–197 (cited on pages 3, 107).

[52] Andrei Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. Effective abstractions for verification under relaxed memory models. In: *Proceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Mumbai, India, Jan. 2015, pages 449–466 (cited on page 44).

[53] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Budapest, Hungary, Mar. 2008, pages 337–340 (cited on pages 8, 21, 31, 36).

[54] Brian Demsky and Patrick Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In: *Proceedings of the 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Pittsburgh, PA, USA, Oct. 2015, pages 20–36 (cited on page 44).

[55] EPICS. *Experimental Physics and Industrial Control System*. 2017. URL: `http://www.aps.anl.gov/epics/` (cited on page 100).

[56] A. P. Ershov. On programming of arithmetic operations. In: *Communications of the ACM* 1.8 (1958), pages 3–6 (cited on page 91).

[57] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In: *Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Vancouver, BC, Canada, Dec. 2012, pages 449–460 (cited on pages 54, 56, 66–68).

[58] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Philadelphia, PA, USA, June 2018, pages 420–435 (cited on page 2).

[59] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, USA, June 2015, pages 229–239 (cited on pages 47, 75).

[60] Matthew Flatt and PLT. *Reference: Racket*. Technical report PLT-TR-2010-1. PLT Design Inc., 2010 (cited on page 7).

[61] Galois, Inc. *Crucible*. 2018. URL: `https://github.com/GaloisInc/crucible` (cited on pages 4, 77, 81).

[62] Malay Ganai and Aarti Gupta. Tunneling and slicing: towards scalable BMC. In: *Proceedings of the 45th Design Automation Conference (DAC)*. Anaheim, CA, USA, June 2008, pages 137–142 (cited on page 86).

[63] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In: *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Chicago, IL, USA, June 2005, pages 213–223 (cited on page 85).

[64] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In: *Proceedings of the 15th Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA, Feb. 2008 (cited on page 85).

[65] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. Gprof: a call graph execution profiler. In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (CC)*. Boston, MA, USA, June 1982, pages 120–126 (cited on page 3).

[66] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Austin, TX, USA, Jan. 2011, pages 217–330 (cited on page 47).

[67] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. In: *Commun. ACM* 55.8 (Aug. 2012) (cited on page 75).

[68] 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)*. San Jose, CA, USA, June 2011, pages 62–73 (cited on pages 54, 67, 68, 74, 75).

[69] *Power ISA Version 2.06 Revision B*. IBM, 2010 (cited on pages 25, 36, 43).

[70] Jeevana Priya Inala, Rohit Singh, and Armando Solar-Lezama. Synthesis of domain specific CNF encoders for bit-vector solvers. In: *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Bordeaux, France, July 2016, pages 302–320 (cited on page 11).

[71] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Revision 53. Intel Corporation, 2015 (cited on pages 17, 22, 36, 38, 40).

[72] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. 2nd. MIT Press, 2009 (cited on pages 16–18, 20, 42, 44).

[73] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffery S. Foster. Adaptive concretization for parallel program synthesis. In: *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*. San Francisco, CA, USA, June 2015, pages 377–394 (cited on pages 54, 70, 109).

[74] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In: *ICSE*. 2010 (cited on page 74).

[75] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In: *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Berlin, Germany, June 2002, pages 304–314 (cited on pages 47, 54, 56).

[76] Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. Refinement types for ruby. In: *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Los Angeles, CA, USA, Jan. 2018, pages 269–290 (cited on pages 97, 102).

[77] James C. King. Symbolic execution and program testing. In: *Communications of the ACM* 19.7 (1976), pages 385–394 (cited on pages 3, 78, 85).

[78] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Philadelphia, PA, USA, Jan. 2012, pages 151–164 (cited on page 10).

[79] Ali Sinan Köksal, Yewen Pu, Saurabh Srivastava, Rastislav Bodik, Jasmin Fisher, and Nir Piterman. Synthesis of biological models from mutation experiments. In: *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Rome, Italy, Jan. 2013, pages 469–482 (cited on page 47).

[80] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Technical report. University of Toronto, 2009 (cited on page 74).

[81] Ivan Kuraj, Viktor Kuncak, and Daniel Jackson. Programming with enumerable sets of structures. In: *Proceedings of the 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Pittsburgh, PA, USA, Oct. 2015, pages 37–56 (cited on page 66).

[82] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China, June 2012, pages 89–98 (cited on pages 86, 90, 106).

[83] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In: *Commun. ACM* 21.7 (1978) (cited on pages 16, 21).

[84] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. Dakar, Senegal, Apr. 2010, pages 348–370 (cited on page 2).

[85] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with SMT solvers. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. San Diego, CA, USA, Jan. 2014, pages 607–618 (cited on pages 3, 76).

[86] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, USA, June 2015, pages 22–32 (cited on page 11).

[87] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. PipeCheck: specifying and verifying microarchitectural enforcement of memory consistency models. In: *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Cambridge, United Kingdom, Dec. 2014, pages 635–646 (cited on page 27).

[88] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated synthesis of comprehensive memory model litmus test suites. In: *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Xi'an, China, Apr. 2017, pages 119–133 (cited on page 44).

[89] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. Generating litmus tests for contrasting memory consistency models. In: *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*. Edinburgh, United Kingdom, July 2010, pages 273–287 (cited on pages 16, 21, 28, 34, 41).

[90] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. Litmus tests for comparing memory consistency models: how long do they need to be? In: *Proceedings of the 48th Design Automation Conference (DAC)*. San Diego, CA, USA, June 2011, pages 504–509 (cited on pages 16–18, 27, 41, 44).

[91] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In: *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*. Berkeley, CA, USA, July 2012, pages 495–512 (cited on pages 15, 36, 43).

[92] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Long Beach, CA, USA, Jan. 2005, pages 378–391 (cited on pages 43, 44).

[93] João P. Marques Silva and Karem A. Sakallah. GRASP—a new search algorithm for satisfiability. In: *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. San Jose, CA, USA, Nov. 1996, pages 220–227 (cited on page 2).

[94] Alexia Massalin. Superoptimizer: a look at the smallest program. In: *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Palo Alto, CA, USA, Oct. 1987, pages 122–126 (cited on page 54).

[95] Paul E. McKenney. *A Formal Model of Linux-Kernel Memory Ordering*. Linux Plumbers Conference. 2016 (cited on page 16).

[96] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Tauman Kalai. A machine learning framework for programming by example. In: *Proceedings of the 30th International Conference on Machine Learning (ICML)*. Atlanta, GA, USA, June 2013, pages 187–195 (cited on page 75).

[97] Leo A. Meyerovich. Parallel Layout Engines: Synthesis and Optimization of Tree Traversals. PhD thesis. University of California, Berkeley, 2013 (cited on page 47).

[98] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. Unifying execution of imperative and declarative code. In: *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. Honolulu, HI, USA, May 2011, pages 511–520 (cited on page 10).

[99] Aleksander Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. Alloy*: a general-purpose higher-order relational constraint solver. In: *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. Florence, Italy, May 2015, pages 609–619 (cited on pages 16–18, 20, 21, 28, 43, 44).

[100] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. SNNAP: approximate computing on programmable SoCs via neural acceleration. In: *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Burlingame, CA, USA, Feb. 2015, pages 603–614 (cited on page 54).

[101] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In: *Proceedings of the 6th International Conference on Learning Representations (ICLR)*. Vancouver, BC, Canada, Apr. 2018 (cited on page 109).

[102] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted Boltzmann machines. In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*. Haifa, Israel, June 2010, pages 807–814 (cited on page 74).

[103] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Munich, Germany, Aug. 2009, pages 391–407 (cited on pages 15, 43).

[104] Scott Owens, Susmit Sarkar, and Peter Sewell. *A Better x86 Memory Model: x86-TSO (extended version)*. Technical report UCAM-CL-TR-745. University of Cambridge, 2009 (cited on page 40).

[105] Seungjoon Park and David L. Dill. An executable specification, analyzer and verifier for RMO (Relaxed Memory Order). In: *Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. Santa Barbara, CA, USA, July 1995, pages 34–41 (cited on page 16).

[106] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. Investigating safety of a radiotherapy machine using system models with pluggable checkers. In: *Proceedings of the 28th International Conference on Computer Aided Verification (CAV)*. Volume 2. Toronto, ON, Canada, July 2016, pages 23–41 (cited on pages 97, 100).

[107] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: synthesis-aided compiler for low-power spatial architectures. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Edinburgh, United Kingdom, June 2014, pages 396–407 (cited on pages 47, 75).

[108] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, USA, Apr. 2016, pages 297–310 (cited on page 97).

[109] Oleksandr Polozov and Sumit Gulwani. FlashMeta: a framework for inductive program synthesis. In: *Proceedings of the 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Pittsburgh, PA, USA, Oct. 2015, pages 107–126 (cited on page 75).

[110] *The Racket Programming Language*. 2017. URL: `https://racket-lang.org` (cited on page 7).

[111] John D. Ramsdell. An operational semantics for Scheme. In: *SIGPLAN Lisp Pointers* V.2 (1992), pages 6–10 (cited on pages 49, 50).

[112] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In: *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*. San Francisco, CA, USA, July 2015, pages 198–216 (cited on pages 68, 69, 75).

[113] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Jose, CA, USA, June 2011, pages 175–186 (cited on pages 15, 43).

[114] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Savannah, GA, USA, Jan. 2009, pages 379–391 (cited on pages 2, 15, 38, 40, 43).

[115] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, TX, USA, Mar. 2013, pages 305–316 (cited on pages 47, 54, 56, 67, 69, 74).

[116] Roberto Sebastiani and Silvia Tomasi. Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ cost functions. In: *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR)*. Manchester, United Kingdom, June 2012, pages 484–498 (cited on page 76).

[117] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In: *Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Saint Petersburg, Russian Federation, Aug. 2013, pages 488–498 (cited on pages 4, 78, 81, 95, 105).

[118] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: multi-path symbolic execution using value summaries. In: *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Bergamo, Italy, Aug. 2015, pages 842–853 (cited on pages 3, 81, 86, 87, 95, 105).

[119] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. In: *Commun. ACM* 53.7 (July 2010), pages 89–97 (cited on pages 2, 15, 38, 40, 43).

[120] James E. Smith. Characterizing computer performance with a single number. In: *Communications of the ACM* 31.10 (Oct. 1988), pages 1202–1206 (cited on page 68).

[121] Armando Solar-Lezama. Program synthesis by sketching. PhD thesis. University of California, Berkeley, 2008 (cited on pages 7, 58).

[122] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Combinatorial sketching for finite programs. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. San Jose, CA, USA, Oct. 2006, pages 404–415 (cited on pages 2, 7, 16, 20, 31, 35, 48–50, 54, 58, 60, 68, 69, 74, 78, 90).

[123] Venkatesh Srinivasan and Thomas Reps. Synthesis of machine code from semantics. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Portland, OR, USA, June 2015, pages 596–607 (cited on page 75).

[124] Matthew Szudzik. An elegant pairing function. In: *NKS 2006 Wolfram Science Conference*. Washington, DC, USA, June 2006 (cited on page 66).

[125] Emina Torlak. *Rosette*. 2018. URL: `http://github.com/emina/rosette` (cited on pages 7, 81, 95).

[126] 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)*. Edinburgh, United Kingdom, June 2014, pages 530–541 (cited on pages 2–4, 7, 16, 18, 20, 36, 49, 50, 58, 78, 86, 87, 94, 96–98).

[127] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In: *Proceedings of the 2013 ACM Symposium on New Ideas in Programming and Reflections on Software (Onward!)* Indianapolis, IN, USA, Oct. 2013, pages 135–152 (cited on pages 2, 4, 7, 16, 18, 20, 36, 49, 54, 74, 94).

[128] Emina Torlak and Daniel Jackson. Kodkod: a relational model finder. In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Braga, Portugal, Mar. 2007, pages 632–647 (cited on pages 16–20, 42).

[129] Emina Torlak, Mandana Vaziri, and Julian Dolby. MemSAT: checking axiomatic specifications of memory models. In: *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, ON, Canada, June 2010, pages 341–350 (cited on pages 16, 18, 21, 27, 35, 43, 44).

[130] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. Transit: specifying protocols with concolic snippets. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Seattle, WA, USA, June 2013, pages 287–296 (cited on pages 54, 74, 75).

[131] Richard Uhler and Nirav Dave. Smten with satisfiability-based search. In: *Proceedings of the 29th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Portland, OR, USA, Oct. 2014, pages 157–176 (cited on pages 78, 82, 83, 85, 87).

[132] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. -Overify: Optimizing Programs for Fast Verification. In: *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS)*. Santa Ana Pueblo, NM, USA, May 2013 (cited on page 107).

[133] Henry S. Warren Jr. *Hacker's Delight*. Addison-Wesley, 2007 (cited on page 54).

[134] David L. Weaver and Tom Germond. *The SPARC architecture manual (version 9)*. SPARC International, 1994 (cited on page 43).

[135] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an smt solver. In: *Proceedings of the 31st ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Amsterdam, The Netherlands, Oct. 2016, pages 765–780 (cited on pages 8, 97).

[136] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In: *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Paris, France, Jan. 2017, pages 190–204 (cited on pages 16, 21, 28, 35, 36, 44).

[137] Max Willsey, Luis Ceze, and Karin Strauss. Puddle: An Operating System for Reliable, High-Level Programming of Digital Microfluidic Devices. In: *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Wild and Crazy Ideas Session*. Williamsburg, VA, USA, Mar. 2018 (cited on pages 97, 102).

[138] 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)*. Long Beach, CA, USA, Jan. 2005, pages 351–363 (cited on page 85).

[139] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: a framework for axiomatic and executable specifications of memory consistency models. In: *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*. Santa Fe, NM, USA, Apr. 2004 (cited on pages 18, 27, 44).

[140] Francesco Zappa Nardelli, Peter Sewell, Jaroslav Ševčík, Susmit Sarkar, Scott Owens, Luc Maranget, Mark Batty, and Jade Alglave. Relaxed memory models must be rigorous. In: *Proceedings of the Workshop on Exploiting Concurrency Efficiently and Correctly (EC^2)*. Grenoble, France, June 2009 (cited on page 15).