Synthesis of Modular and Optimal Programs

A dissertation

submitted by

Nate F. F. Bragg,

In partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Science

TUFTS UNIVERSITY

February 2026

ADVISOR: Jeffrey S. Foster

Abstract

Sketch-based program synthesis is a popular code generation technique that takes an incomplete program written in a high level language and produces a completed program that meets some correctness specification. A major challenge for sketching is that the size of the search space for that program's missing parts may be large enough that even if a solution exists, finding one in practice is intractable. Traditional sketching systems also lack support for when a user prefers one solution to another based on properties difficult to state with assertions, such as minimizing or maximizing some quantity. Because the results of synthesis depend on the amount and quality of information available to the system, users must carefully design sketches to be complete enough with a rich enough specification that they synthesize quickly and produce a desirable program.

This dissertation explores two novel approaches to leverage information within a sketch to improve synthesis. The first approach, Sketcham, augments an existing sketch-based system with an additional transformation to modularize a sketch by automatically generating mocks from its test suite. The second approach introduces a new sketch-based system using the functional core language Scimitar powered by the optimization-aided synthesis algorithm OACIS. This algorithm integrates a mixed integer linear program solver, enabling the user to add goal functions that encode properties about syntactic or semantic properties they want to optimize, such as program structure, resource usage, or other domain specific behavior.

These approaches establish viable new avenues for users to improve their ability to synthesize programs by augmenting the information about program structure and objectives available to these solver aided systems. Example sketches showcasing these algorithms demonstrate that both approaches are applicable to diverse domains including numeric functions, string manipulation, logistics, and resource management. The performance of these benchmarks shows that the approaches presented are effective and capable of matching or improving on existing techniques.

To my child

“All right, then ask it something else! Whatever you like! Go on! What are you waiting for? Afraid?!”

“Just a minute,” said Klapaucius, annoyed. He was trying to think of a request as difficult as possible, aware that any argument on the quality of the verse the machine might be able to produce would be hard if not impossible to settle either way. Suddenly he brightened and said:

“Have it compose a poem—a poem about a haircut! But lofty, noble, tragic, timeless, full of love, treachery, retribution, quiet heroism in the face of certain doom! Six lines, cleverly rhymed, and every word beginning with the letter *s*!!”

“And why not throw in a full exposition of the general theory of nonlinear automata while you’re at it?” growled Trurl. “You can’t give it such idiotic—”

But he didn’t finish. A melodious voice filled the hall with the following:

Seduced, shaggy Samson snored.
She scissored short. Sorely shorn,
Soon shackled slave, Samson sighed,
Silently scheming,
Sightlessly seeking
Some savage, spectacular suicide

“Well, what do you say to that?” asked Trurl, his arms folded proudly.

— Stanisław Lem

The Cyberiad: Fables for the Cybernetic Age

Translated from the Polish by Michael Kandel

Acknowledgments

There are so many people who have contributed to the success of my graduate studies, and to my growth as a person. I would not be here without their help, and I am humbled and blessed by their presence in my life.

To my advisor Jeffrey S. Foster, who was always ready to help, dispensed sage advice and guidance every step of the way, and always heard me out no matter how unconventional the idea,

To my master's degree advisor Kathleen Fisher, who helped launch my research career and convinced me that I could and should pursue a doctorate,

To my mentor Philip Zucker, who was always full of good ideas, knew just the right related work, and had a refreshing alternative perspective,

To my committee members Anselm Blumer and Mai Vu, who both taught me so much,

To Norman Ramsey, without whom I never would have even attended Tufts, and without whom this dissertation would certainly not exist,

To Moses Huang, Karl Cronburg, Lucía Nuñez, Matthew Ahrens, Jeanne-Marie Musca, Brian LaChance, Samuel Lasser, Jared Chandler, Sasha Fedchin, Fox Huston, Mark Aldrich, Antero Mejr, Milod Kazerounian, Sankha Guria, Ferdinand Vesely, and the other members and extended members of the Tufts PL family, who not only often offered me invaluable feedback on my work, but whom I consider all dear friends,

To my collaborator Armando Solar-Lezama, a giant on whose shoulders I sit,

To the Charles Stark Draper Laboratory and my colleagues Cody Roux, Zachary Stone, and Chris Casinghino, without whose generous financial support and mentorship I could not have completed this work,

To my Bose colleagues Yong Gao, Rameshwar Sundaresan, Herb Knapp, Mike Moretti, Michael Cook, Brian White, and Nathan Michaels, who believed in and encouraged me,

To Michael Ballantyne, Sos Oganessian, and Anik Rounds, for your camaraderie,

To Donna Cirelli, Megan Monaghan, Ellen Quirk, Sarah Richmond, Sandie Schulenberg, Sarah White, Angelica Carberry, Rebecca Curran, Johnny Redmond, Michael Bauer, Patrick Hynes,

Erik Patton, Song Hoang, and the rest of the Tufts CS department staff, without whom I would have been completely helpless,

To Sam Guyer, Greg Aloupis, Rajasekhar Inkulu, and Rob Jacob, who helped me on my academic path,

To Paulo Lizano and Luís Sandoval, who helped keep me from literally going crazy,

To TWRP, who helped me during very difficult times to remember that life is a party, and Dan Avidan for reminding me that the best bananas will be ripe in time,

To my adopted grandmother Kickan Chretien and her late husband Max, who took me in at my lowest point,

To my parents Jeffrey and Cheryl Bragg, and my sibling Kae, who have always been there for me, and always will be,

And to my wife Manu, who brings joy to my life, and fills my heart,

Thank you all.

Nate F. F. Bragg

TUFTS UNIVERSITY
February 2026

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Human beings are better at creating programs than reasoning about them. Software Engineering is difficult, and the problems we want to solve are often more complex than we believe and full of edge cases. Even well made programs written by experienced engineers fail when confronted with input from the real world, which is not always as well ordered as they had expected. Computers are however very good at correctness. We would like it if we could leverage our computers to help us create correct code.

This is the motivation for the study of *program synthesis*. The problem originates out of the field of program verification, and was introduced by Manna and Waldinger's seminal 1979 paper *Synthesis: Dreams → Programs* [MW79]. While verification strives to ensure that an existing program meets some correctness specification, synthesis endeavors to use that specification to create an entirely new program. There have been many approaches to tackling the problem, including syntax-guided, inductive, and neural-symbolic techniques, with diverse applications across all areas of computer science. [STB+06, SLAT+07a, SL08, Sol09, CSL10a, KMPS10, SGF10, GJTV11, TB13, SSX+14, FCD15, JQSLF15, RDK+15, ADG16, ISSL16, JQFD+16, LSO16, PKSL16, YKDC16, FMW+17, IPQ+17, Leu17, MFP+17, NWKF17, YWDD17, ERSLT18, WADM18, WDS18, WWD18, ASMS19, ENP+19, HZZK19, KWPH19a, MRX+19, SA19, SRHN19, SSL19, GJJ+20, KMM+20, WK20, HCDR20a, HQSW20a, EWN+21, IPP+21, KHDR21]

Program synthesis by sketching is a popular synthesis approach built around incomplete programs called *program sketches*. Users submit sketches along with their correctness specifications to a synthesis system that then fills in their gaps to produce completed programs conforming to those specifications.

As with other synthesis approaches, a major challenge for sketching is that while a given sketch may be synthesizable, the size of the search space for that solution may be so large that finding a result in practice may be intractable due to the fact that synthesis results are only as good as the information a user puts into the program sketches. The less information provided to the synthesis system, the longer it will likely take to get a result and the lower the likelihood of a high quality result. Results can be improved by adding more high quality information to the sketch and there are many ways for users to provide such extra information. A common one is to write a more fleshed out program sketch. Another approach is to modify the specification. Yet

another is to manipulate the program's meta information by setting synthesizer parameters.

In this work we explore two novel approaches that leverage such extra information in different ways to improve sketch synthesis from within by modifying the systems themselves. Both approaches are designed to increase the system's knowledge of the problem without imposing additional constraints. The first novel approach builds a new program transformation onto an existing synthesis system that modularizes a program sketch by breaking it up using information latent in the sketch's own specification and separately synthesizing its pieces. While promising, this strategy's limits require us to set aside modularization and to regard the problem of improving synthesis from a new perspective. The second novel approach overcomes the limits of the first by introducing an entirely new sketching synthesis system based on a functional core language which integrates a numerical optimization solver. This system includes fundamental changes to the synthesis algorithm that, rather than transforming the program sketch, enable the user to add goal functions that can direct the progress of synthesis.

> The central claim of this dissertation is that augmenting the information about program structure and objectives available to these solver aided systems leads to better results, both when synthesizing programs and when searching for optimal program values.

1.1 Contributions

Chapter 2 introduces Sketcham, a program transformation that modularizes the sketch by taking advantage of information implicit in the original program specification. It modifies the sketch to include a model of its own specification without the user needing to supply additional information by traversing the original sketch looking for specifications of individual functions, then breaking them apart to synthesize them individually. It schedules these in the synthesizer in order of dependencies, so that each function is synthesized before those that depend on it. It then attempts the complete sketch to verify the results. Doing so often results in speedups, but the modular approach's benefits are limited to programs that are highly separable and sufficiently complicated that it is intractable to solve them directly.

Chapters 3 and 4 move beyond the limits of Sketcham. We add a fundamentally new capability missing from existing systems, which is a larger departure from traditional synthesis systems, as it requires three major changes. First, Chapter 3 introduces the optimization-aided language Scimitar, describes its semantics, and gives a number of encodings used by the compiler backend with an optimization solver to create an entire new synthesis system. Next, Chapter 4 uses Scimitar as the platform for the new sketch synthesis system Synthitar. This lets us replace the traditional system solver with an optimization solver, adding the ability to specify objective functions, which are goals that direct the solver's decision making. This system also augments the core sketch synthesis algorithm to enable constructing complement optimization programs and modifies the subroutines that verify the generated program meets the user's specification to handle our encoding.

We now discuss the key ideas of this dissertation in more detail.

1.2 Modular Synthesis

Program synthesis by sketching, as embodied by the Sketch synthesis tool [STB+06], is a popular technique that has been applied to a wide variety of problems [CSM12, ERSLT18, IPQ+17, ISSL16, JQFD+16, JQSLF15, MRX+19, SKR19, SLAT+07a]. A Sketch input (henceforth a *sketch*) is a program written in a C-like language augmented with *holes*, unknown constants, and *generators*, unknown expressions. The solution for a sketch is specified using test cases called *harnesses*, also written in the Sketch language, that make assertions about the results of to-be-synthesized code. Here is an example of a simple sketch written in the Sketch language [SL20]:

```
harness void double(int x) { assert ?? * x == x + x; }
```

This harness function takes an input `x`, and consists of a specification with a single constraint, that for all values of `x`, `x + x` is equal to `x` times `??`, which designates a hole. The correct assignment for this hole is 2, which Sketch is tasked with finding. Sketch searches for a solution using *counterexample-guided inductive synthesis (CEGIS)*, which alternately synthesizes a candidate solution and then uses a verifier to check the assertions; any counterexamples from verification feed into the next round of synthesis [Sol09].

One key challenge of using Sketch is that it does not specifically support modular synthesis. More precisely, even if an input sketch is divided into a number of functions that call each other, Sketch solves them all together. This approach potentially limits scalability, as formulas created by Sketch can grow quite quickly as function calls are inlined. A Sketch user could potentially work around this issue by manually replacing calls to to-be-synthesized functions with calls to Sketch *models* [SSX+14], which are *mocks*, i.e., functions that, in place of full implementations, approximate the desired behavior with a specification in the form of assertions about individual cases. However, writing additional specifications is both time consuming and redundant with developing the original sketch.

Explicitly, a program's assertions encode the desired behavior of the functions they refer to. For a sufficiently decomposable problem, these same assertions also implicitly define the behavior that callers rely on. Properly collected, these assertions can form mocks that the synthesizer could use to break apart the program into modular components without the developer needing to explicitly include new information.

In Chapter 2, we introduce SKETCHAM (short for *Sketch and Mocks*), a novel technique that converts a regular sketch problem into a modular sketch problem by *automatically* generating mocks from harnesses. SKETCHAM is a program transformation that collects assertions referring to functions into mocks. It operates at function granularity because functions are the most basic modular building blocks of programs. After grouping these assertions, the algorithm then constructs new subproblems by replacing calls to these functions with their mocks. Any valid solution to the overall problem will also satisfy these subproblems. They help the synthesizer make better initial guesses as to what the finished product is going to be, but do not give the complete overall solution. A valid overall solution is guaranteed by running the original problem with the subproblem holes pre-filled.

We implemented Sketcham as an additional pass to Sketch in three algorithms, one to collect assertions, a second that generates mocks, and the last that constructs new harnesses and adds them to the synthesis problem. We evaluated Sketcham on ten benchmarks including simple arithmetic, string manipulation, and list processing functions. We found that, for six of ten benchmarks, Sketcham performs up to $5\times$ faster than Sketch, for one benchmark Sketcham is slower by a factor of up to $0.9\times$, and for the remaining three benchmarks performance is indistinguishable. As expected, the main speed improvement was in the CEGIS synthesis phase rather than the CEGIS verification phase.

1.3 Optimization-aided Programming

Our discussion of optimization-aided synthesis is broken into two parts. We begin by turning our attention to the core underlying optimization-aided programming language. This language gives users the ability to optimize some goal function subject to some constraints. A mixed integer linear programming (*MILP*) solver determines the values needed by the code to achieve an optimal result from that goal function.

MILP is a classic constraint optimization approach for problems with linear and integer variables and constraints. It has applications ranging from simple to complex, including network and logistics problems, and machine learning.

Despite the success of MILP, in practice it can still be very challenging to write MILP programs. The difficulty lies in MILP's limited expressiveness, which means that MILP programmers must use complex encodings to map the semantics of their problems into the MILP language. For example, encodings enable MILP to be applied to domains such as boolean logic [Bra12], nonlinear functions like multiplication [Dom18] or piecewise linear functions [HV22], conditional constraints [GAM22, HV19, RJG$^+$12], and others [Wik24, Par88]. However, while such encodings are effective, they require significant expert knowledge to use, and they make programs difficult to maintain as they complicate and obscure the underlying problem.

This limitation makes it non-trivial to apply optimization principles to program synthesis, as synthesis systems are programmed in high-level languages. Synthesizing non-trivial optimization problems within the implementation of a particular sketch is challenging or even intractable without direct access to an optimization solver.

In Chapter 3 we introduce Scimitar, an *optimization-aided language* that enables seamless integration of MILP constraint optimization problems into functional programs. Several researchers have explored integrating constraint solvers into programming languages, e.g., logic languages like SWI-Prolog [WSTL12], verification-aware languages like Dafny [Lei10], prover languages like Why [Fil03], and the Rosette solver-aided language [TB13]. However, these languages focus on decision procedures such as *satisfiability modulo theories (SMT)*, a generalization of *Boolean satisfiability (SAT)*. In contrast, the design and implementation of Scimitar show how to integrate an *optimization solver* into functional programming. The key novelty of Scimitar is that functional expressions containing function calls, conditionals, loops, and more, can be compiled into optimization problems via a special *minimize* construct.

We implement Scimitar in the Racket programming language, and use the popular off-the-shelf Gurobi MILP solver [Gur23]. To demonstrate Scimitar's features and explore its capabilities, we show several examples. These included both traditional logistics problems and real-world programming use cases such as an arena allocator. We measure median performance, giving evidence that a dedicated MILP solver compares favorably with the more general purpose SMT solver on optimization applications. Our evaluation splits out the solve time to better show the performance of the algorithms on their own, excluding the performance of the compiler. The benchmark compile times were <1ms–67ms for Rosette, ignoring one that timed out, and 30ms–2200ms for Scimitar, 7–400× slower than Rosette. The solve times for Rosette were <1ms–125000ms, ignoring one that timed out, while Scimitar's were <1ms–1000ms, 4–200× faster than Rosette. Combined, Scimitar was from $45\times$ faster than Rosette to $80\times$ slower.

1.4 Optimization-aided Synthesis

With our optimization-aided language at hand, we are now prepared to move onto the second part of our discussion of optimization-aided synthesis. The system we introduce here is designed to optimize an objective function, leading to a synthesized program that maximizes or minimizes some user-supplied goal.

As discussed above, program synthesis using CEGIS iteratively generates programs using two competing SMT solvers: the synthesizer, which finds candidate solutions that meet a user-supplied specification, and the verifier, which attempts to find counterexample inputs to these solutions that violate the specification. To do so, the language compiles the high-level program into a SMT formula modeling the original. The solver then searches for any assignment to the holes that satisfies the specification completely without any input counterexamples. However, there are times when not just any hole assignment will do—users often want to find some assignment that also satisfies properties difficult to state with assertions alone. A common desirable property is to minimize some quantity, such as code size, execution time, energy, network or disk usage, etc. One way to create heuristics that optimize these properties is via MILP.

Because any satisfying assignment is an acceptable solution, it is challenging for SMT-based CEGIS synthesis to direct solving using other heuristics, including hyperproperties of holes (e.g., a minimal number, or some order of preferences) without major changes to the user's own program. Any such changes would likely burden both solver and user significantly, as they would only implement a simulation of the heuristic. An example would be to name all of the program's holes (e.g., using intermediate variables), then add constraints that impose the desired property on the holes. Users should not need to resort to such ad hoc and unreliable techniques, and we can overcome this by augmenting the synthesis system so they can express these properties directly.

In Chapter 4 we introduce the *Optimization Aided Counterexample-guided Inductive Synthesis (OACIS)* algorithm, embodied in the brand new synthesis system Synthitar, that extends Scimitar with a generalized CEGIS algorithm using a MILP solver in place of the usual SMT solver. Adding a synthesis query enables users to augment sketches with objective functions, goals that specify how to direct synthesis. Objective functions with access to a sketch's holes can encode

important properties that the sketch itself cannot express, but that are nevertheless necessary or desirable in any solution. By adding objectives, a user can influence the size, value, or shape of program values and structure, either in isolation or in relation to other parts of the program.

The use of a numerical solver makes Synthitar particularly well suited to numerical problems, specifically those with holes that have floating point types. Because its objective functions are written directly in the language, users can reference any of their program variables or functions, making it more expressive as compared to more restrictive systems that can only minimize syntactic properties such as program size.

The OACIS algorithm relies on three key novelties. First, unlike traditional SMT encodings, a MILP-based approach requires special symbolic *local* variables both to encode internal program structure during compilation, and to allow for more flexibility for users to model their domain. These variables cause the traditional CEGIS loop to fail, as the verifier will find unreachable program states rather than input counterexamples. To prevent this failure mode, OACIS replaces the verifier with an additional specialized inner CEGIS loop. This gives rise to a generalized algorithm we call *General CEGIS*, which we then specialize for both the outer synthesis and inner verification loops. Second, while negating an SMT formula is straightforward, finding the complement of an optimization problem is tricky as MILP has no built-in disjunction or negation, so detecting an individual constraint's violation is nontrivial. Our new *Maximum Variable Delta (Max-Δ) Verification* approach detects violations through the use of a special Δ variable to find worst-case counterexamples. The Max-Δ algorithm goes hand in hand with our third novel contribution, a new MILP inversion algorithm that, given an optimization-aided sketch, constructs its complement. The solution to this problem is a point that is infeasible in the original problem, exactly what we need for a counterexample.

As an extension of Scimitar, Synthitar is also implemented in the Racket language. To evaluate Synthitar, we give a suite of benchmarks that exercises various aspects of the language, whose complexity ranges from simple mathematical expressions to an algorithm to calculate square roots. We measure performance and convergence of OACIS, finding it to be linked primarily to the size of the input domain and the structure of the complement problem.

1.5 Road Map and Publication History

The remainder of this dissertation is organized as follows:

- Chapter 2 presents Sketcham, which modularizes the Sketch synthesis system by transforming harnesses into mocks, based on the *Computer Aided Verification (CAV)* 2021 paper "Program Sketching by Automatically Generating Mocks from Tests" [BFRSL21].
- Chapter 3 introduces Scimitar, an optimization-aided functional language, first presented in the *Onward!* 2024 paper "Scimitar: Functional Programs as Optimization Problems" [BFZ24a].
- Chapter 4 describes Synthitar, a complete synthesis system that extends Scimitar with the OACIS algorithm.
- Chapter 5 concludes the dissertation and outlines potential avenues for future work.

Chapter 2

Program Sketching by Automatically Generating Mocks from Tests

2.1 Introduction

Program synthesis by sketching is a technique that lets users write program sketches in a high-level language augmented with *holes*, unknown constants, and *generators*, unknown expressions, that are synthesized by encoding their *harnesses* into Boolean satisfiability (SAT) formulas submitted to a *CEGIS* solver. This approach is exemplified by the Sketch synthesis tool [STB+06]. Sketch's C-like language provides a rich set of language features, but it does not specifically support modular synthesis, the ability to break a sketch apart to synthesize it in smaller pieces.

This chapter introduces SKETCHAM (short for *Sketch and Mocks*), a technique that converts a regular sketch problem into a modular sketch problem by automatically generating mocks from harnesses. It collects assertions referring to functions, groups them, then programmatically constructs new subproblems by replacing calls to these functions with their mocks, radically simplified versions that partially replicate the functionality of the original. More specifically, suppose SKETCHAM is given a sketch in which function f calls g and g is tested by harness h. SKETCHAM first converts h into a mock g_m that has the same function signature as g but whose body encodes the assertions from h. Then, SKETCHAM augments the original sketch, prepending new code in which f calls g_m instead of g, thereby allowing f to be synthesized separately from g. Thus, by converting tests (harnesses) to mocks (specs), SKETCHAM enables modular synthesis without extra work from the user. Section 2.2 gives an overview of SKETCHAM.

SKETCHAM generates the new, modular sketch problem using a sequence of three algorithms. First, SKETCHAM traverses the original sketch to build a mapping A from function names to a set of assertions in which each function is called. Note that we place some limitations of the assertions—e.g., they can contain at most one function call—to guarantee we can always translate them from harness assertions to mock assertions. Next, SKETCHAM traverses A, generating a mock f_m for each function $f \in dom(A)$, where f_m encodes the assertions in $A(f)$. Finally, SKETCHAM generates new mock harnesses that are the same as the original harnesses, except they call mocks instead of the underlying functions. Section 2.3 presents SKETCHAM's core algorithms.

We implemented SKETCHAM as an additional compiler pass to Sketch, which we evaluated on ten benchmarks. We found a high variance in running time, both under Sketch and under SKETCHAM. This variance is due to the use of randomness in the algorithm—the solver's random seed determines its initial conditions, which in turn determines how the solver proceeds to search for the solution, and a good initial guess at a solution will make things proceed much more swiftly than an unfortunate guess. To account for this difference, we used the Clopper-Pearson method [CP34], repeatedly running each configuration (synthesis tool–benchmark combination) from 11 to 1487 times to reach 95% confidence that the true median running time lies within 20% of the experimental median, excluding failures and runs exceeding a 60 minute timeout. We found that, for six of ten benchmarks, SKETCHAM runs up to 5× faster than Sketch; for one benchmark SKETCHAM is up to a factor of 0.98× slower; for the remaining three benchmarks, performance is indistinguishable. We examined one benchmark, deduplication of elements in an array, in detail. We found that the performance improvement is largely due to a mock that does a thorough job representing the function it mocks, and that the performance improvement occurs during the CEGIS synthesis phase rather than the CEGIS verification phase. Section 2.4 presents our evaluation.

In summary, SKETCHAM demonstrates that modular synthesis can be achieved by automatically generating mocks from tests (specs from harnesses) without additional user effort.

2.2 Overview

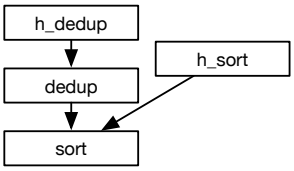
To illustrate SKETCHAM, consider Figure 2.1a, which shows a simplified sketch whose solution deduplicates an array of integers. This sketch makes use of Sketch holes `??`, which are unknown constants, and generators such as `expr(vars, ops)`, which is an unknown expression composed of variables `vars` combined with operands `ops`, including `PLUS` for addition and `MINUS` for subtraction. The correct solutions for the holes and generators are shown in end-of-line comments.

At the top of Figure 2.1a, function `dedup` takes a length `n` and array `vs`, and it returns the deduplicated array and, by reference, the deduplicated array's length `sz` (in Sketch, functions can only have at most one return value, hence the return-by-reference `sz`). The `dedup` function begins by calling another function, `sort`, to sort the array (line 3). Then it initializes `sz` to a hole and loops through the array (lines 4-5). In each iteration, it computes an expression `j` of `sz` and `i` (line 6) used in a conditional guard (lines 7-8) along with `sz` and the sorted and deduplicated arrays. If the condition holds, the element at position `i` is copied into `res` and `sz` is updated; otherwise the element is ignored. Finally, `dedup` returns the result array `res`.

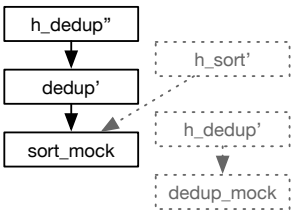
The `sort` function (line 14) takes the length and array and returns a sorted array. This particular sketch is for merge sort. Here the programmer knows that merge sort involves sorting two sub-arrays but is not sure about the details. After some initialization, it makes two recursive calls to sort sub-arrays (lines 16 and 17). Then it loops over the sorted sub-arrays, merging the elements into array `vs`, which is returned. The loop guard (line 18) uses a different generator, `exprBool(vars, ops)`, that generates arithmetic comparisons (`<`, `<=`, etc) among expressions generated by calling `expr(vars, ops)`.

```
int[n] dedup(int n, int[n] vs,
             ref int sz) {
 int[n] svs = sort(n, vs); int[n] res;
 sz = ??; // 0
 for(int i = ??; i < n; ++i) { // 0
  int j = expr({sz, i}, {PLUS, MINUS}); // sz - 1
  if(sz == ?? || // 0 >
     {| svs[i] (> | >= | < | <= | != | ==) res[j] |} ) {
    res[sz] = svs[i];
    sz = expr({sz, i}, {PLUS, MINUS}); // sz + 1
 }}
 return res;
}
int[n] sort(int n, int[n] vs) {
 int mid = n/2, rem = n - n/2, i = 0, j = 0;
 int[mid] vas = sort(vs[0::mid]);
 int[rem] vbs = sort(vs[mid::rem]);
 while(exprBool({i, j, n}, {PLUS})) // i+j<n
  /* add vas[i++] or vbs[j++] to vs */
 return vs;
}
```

(a) dedup and sort (simplified).



(b) Original harnesses.



(c) Mock harnesses.

```
harness void h_sort(int n,
                    int[n] vs) {
 int[n] svs = sort(vs);
 for(int i=0; i<n-1; ++i)
   assert svs[i] <= svs[i+1];
 /* also elts(vs)=elts(svs) */

}
```

➪

```
model int[n] sort_mock(int n,
                       int[n] vs) {
 int[n] svs = sort_uf(vs);
 for(int i=0; i<n-1; ++i)
   assume svs[i] <= svs[i+1];
 /* also elts(vs)=elts(svs) */
 return svs;
}
```

(d) Translating sort’s test harness into a mock.

Figure 2.1: SKETCHAM applied to deduplication via sorting

Harnesses and Mocks. To test the expected behavior of dedup and sort, the sketch also includes two harnesses, h_dedup and h_sort. Figure 2.1b shows the call graph of the sketch with the harnesses, and the left side of Figure 2.1d shows a portion of h_sort (for the complete harness, and for h_dedup, see Figure A.7). This harness calls sort and then makes assertions about the results, e.g., that the output array is sorted. Harnesses are distinguished from regular functions by the keyword **harness**, and their arguments are treated as universally quantified—the harness must hold no matter the concrete value that one of its input variables takes on. Thus, h_sort tests that for all n and arrays vs of length n, the sort function is correct.

To solve this synthesis problem, Sketch converts dedup, sort, and a harness into a single SAT formula and then uses CEGIS to find a solution. This approach works, but the formula passed to the solver is large, because it contains both functions’ worth of code, and complex, because reasoning about the code in dedup requires simultaneously reasoning about the code in sort. Thus, combining both functions into a single SAT formula potentially limits the scalability of Sketch.

```
int doub(int m) {
  return m * 2;
}
harness void h(int n) {
  int out = doub(n * 10);
  assert out == (n + n) * ??;
}
```

(a) Double.

```
model int doub_mock(int m) {
  int out = doub_uf(m);
  assume (0 == m%10) ⟹
     out == (m/10 + m/10) * ??;
  return out;

}
```

(b) Mock double.

Figure 2.2: The double function and its mock

The key idea of Sketcham is to observe that this sketch is actually modular—it has been divided into two functions, each with their own tests. Sketcham takes advantage of this modularity by creating a new synthesis problem that includes mock versions of functions in the sketch, which can then be used to enable separate reasoning about each function.

The right side of Figure 2.1d shows `sort_mock`, the mock version of `sort`. The mock has the same signature as `sort`, but instead of containing the actual sorting code, it contains assertions from `h_sort` about `sort`'s expected behavior. In place of calling `sort`, the mock calls a fresh uninterpreted function `sort_uf` on line 3. Such uninterpreted functions are symbols whose only property is equality, so that for $a = b$, then $f(a) = f(b)$. Then it makes assumptions (rather than assertions) about the result array `svs` (line 5). While assertions place requirements on the postconditions which the synthesizer must satisfy to succeed, these assumptions make guarantees about the precondition that must hold for the sketch to be valid. The mock then finally returns `svs` (line 7). The mock itself is a Sketch model (indicated by the **`model`** keyword), and where the mock is called, Sketch will replace the call with the assumptions in the model's body [SSX+14].

Next, Sketcham creates new code that uses the mock, as shown in Figure 2.1c. (Here the dashed, greyed boxes are for functions and harnesses that are generated but do not improve solving time; see Section 2.4.2.) In particular, `dedup'` is the same as `dedup`, except it calls `sort_mock` instead of `sort`, and `h_dedup''` is the same as `h_dedup` but it calls `dedup'` instead of `dedup`.

The final sketch includes `h_dedup''`, `h_dedup'` (a trivial harness that calls a mocked `dedup`), and `h_dedup`—in that order—as well as the harnesses for `sort`. Sketcham searches for a solution for each harness in order, i.e., it tries to solve `h_dedup''` first. Notice that, critically, when Sketcham solves `h_dedup''`, it need not consider the code of `sort`, but rather only its specification as encoded in the mock. In practice, this means that Sketcham can solve `h_dedup''` up to 18.1× faster than Sketch solves `h_dedup`, a significant speedup.

Moreover, `sort_mock` encodes the specification of `sort`, so once Sketcham solves `h_dedup''`, it has found a solution for `h_dedup` as well. To preserve correctness, Sketcham keeps the original harnesses such as `h_dedup`, because mocks with partial specifications can lead to partially incorrect solutions to the harnesses using them. However, even in these cases, the counterexamples they generate can still help more quickly narrow the synthesis search space for the original harness, and lead to an ultimately valid solution.

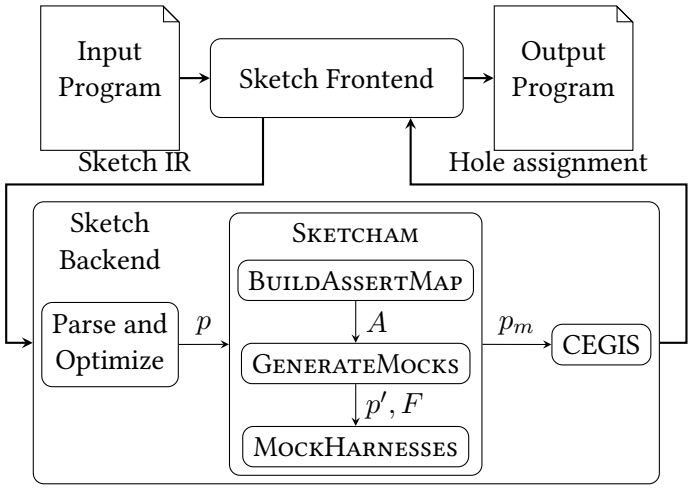


Figure 2.3: SKETCHAM architecture

Quantifier Elimination. In Figure 2.1d, the translation from harness to mock was straightforward: the call to the mocked function becomes a call to an uninterpreted function, and **`assert`**s become **`assume`**s. Sometimes, however, the translation is more complex. Consider the sketch in Figure 2.2a, which includes a function doub that doubles its input and a harness h that calls doub`(n*10)` and asserts the result is `(n+n)*??` for some hole.

Notice this assertion only describes arguments of the form `n*10` for some n, i.e., implicitly there *must exist* some m such that `m = n*10` for the assertion to hold. This *existential quantifier* introduces n, which our translation must then get rid of. SKETCHAM performs *quantifier elimination* [Bjø10, BM07] on such nested existentials, following the approach of Kuncak et al. [KMPS10]. Figure 2.2b shows the resulting mock. Here, in the assumption, n is replaced by witness candidate `m/10`. Because m is an integer, we also add a precondition that m is evenly divisible by 10.

We note that SKETCHAM includes quantifier elimination for completeness, and in our evaluation we consider the sketch in Figure 2.2a. However, we did not find quantifier elimination necessary for our other benchmarks.

2.3 The SKETCHAM Algorithm

Next we more formally describe SKETCHAM, which is implemented as a compiler pass within Sketch as shown in Figure 2.3. The presentation that follows reflects this Sketch implementation without loss of generality of the core algorithm for converting tests to mocks. The Sketch *frontend* consumes the input sketch and transforms it into the Sketch intermediate representation language (Sketch IR), which is passed to the Sketch *backend*. Sketch IR encodes first-order logic augmented with theories of arithmetic, arrays, functions, and more, as discussed below. When the backend loads the IR, it performs transformations that are needed by the solver [SL08], including loop unrolling and function inlining, which emplace the body of a loop or function at each occurrence up to some user defined number of repetitions. The output of all these transformations is a program p. Standard Sketch then uses CEGIS to solve the synthesis problem, outputting a hole assignment that the frontend uses to produce the solved sketch. SKETCHAM modifies this process by inserting

$$
\begin{array}{lll}
p & ::= & (h \mid d)^* \\
h & ::= & \text{harness } d \\
d & ::= & \text{def } f\ (\ x\ ,\ \ldots,\ x\ )\ \{\ s^*\ \} \\
s & ::= & x := e \mid \text{return } e \mid \text{assert } \phi \mid \text{assume } \phi \\
e, \phi, \psi & ::= & f\ (\ e\ ,\ \ldots,\ e\ ) \mid \mathit{uop}\ e \mid e\ \mathit{bop}\ e \mid n \mid x \mid ??\ x \\
\mathit{uop} & ::= & \neg \mid - \\
\mathit{bop} & ::= & \wedge \mid \vee \mid \oplus \mid \implies \mid = \mid + \mid - \mid * \mid / \mid \%
\end{array}
$$

$x, y \in$ variable names $\quad G \in$ graphs $\quad A : f \to \Phi$

$f, g \in$ function names $\quad \Phi \in$ set of ϕ $\quad F : f \to f$

Figure 2.4: SKETCHAM's fragment of Sketch IR

a *mock rewriting* phase, described below, that transforms p into the augmented program p_m for CEGIS.

We formalize SKETCHAM on the fragment of Sketch IR shown in Figure 2.4. This subset of the grammar omits types, and we assume the sketch is type-correct. A program sketch p is a sequence of harness and function definitions. A harness definition h tags a function definition as a test harness. A function definition d is given named parameters[1] and a body, which is a sequence of statements. Statements s are assignments, returns, assertions, and assumptions. The most critical expressions e in our algorithm are function calls $f\ (\ e\ ,\ \ldots,\ e\ )$ with their arguments. The detailed grammar for the remaining expressions is unimportant in the remainder of this section, but for completeness we show expressions for unary and binary logical and arithmetic operations *uop e* and *e bop e*; constants *n*; variables *x*; and named holes ?? *x*. Below, we sometimes use the metavariables ϕ and ψ in place of e to indicate an expression used for Boolean-valued formulas.

Given the input Sketch IR program p as shown in Figure 2.3, SKETCHAM creates the output sketch by first calling BUILDASSERTMAP (Algorithm 1) to build mapping A from function names to assertions from tests of those functions. Next, GENERATEMOCKS (Algorithm 2) uses A to construct mocks for functions in the domain of A, yielding program p', which includes the original sketch p plus those mocks. This step also returns a mapping F from the original function names to the corresponding mock names. Finally, MOCKHARNESSES (Algorithm 3) creates the output sketch p_m, which augments p' with copies of the original sketch's harnesses, except the copies call the mocks instead of the original functions.

Critically, during this last step, holes are *not* renamed when the harnesses are copied. Moreover, the newly generated harnesses are prepended to the sketch. Thus, when CEGIS tries solving each harness in p_m in order, it will first find solutions that are consistent with the mocks. Then when it reaches the original harnesses (which must remain in case there is information in them not captured by the mocks—see discussion of GENERATEMOCKS below), CEGIS can use the information it already derived from the mocks to find the ultimate solution to the original problem.

The remainder of this section describes each step of the algorithm in detail. Below, we

[1]For simplicity, we assume parameter names are unique across the whole program.

Algorithm 1 Mock rewriting: building the assertion map

Input: p - the sketch
Output: A - finite map of function names to sets of assert formulas

1: **function** BUILDASSERTMAP(p)
2: $A \leftarrow \emptyset$
3: $\Phi \leftarrow \{\phi \mid \texttt{assert}\ \phi \in p\}$ ▷ all solver-reachable asserts in p
4: $\Phi_0 \leftarrow \{\phi \in \Phi \mid 0 = |f(\ldots) \in \phi|\}$ ▷ asserts with 0 function calls
5: $\Phi_1 \leftarrow \{\phi \in \Phi \mid 1 = |f(\ldots) \in \phi|\}$ ▷ asserts with 1 function call
6: **for all** $f \in \Phi_1$ **do**
7: $\Phi_f \leftarrow \Phi_0 \cup \{\phi \in \Phi_1 \mid f \in \phi\}$ ▷ asserts with 0 calls, or 1 call to f
8: $\Psi \leftarrow \Phi \setminus \Phi_f$
9: **while** $\Psi \neq \emptyset$ **do**
10: $X \leftarrow \mathrm{FV}(\Psi)$ ▷ inputs and holes free in Ψ
11: $\Psi \leftarrow \{\phi \in \Phi_f \mid X \cap \mathrm{FV}(\phi) \neq \emptyset\}$
12: $\Phi_f \leftarrow \Phi_f \setminus \Psi$
13: **end while**
14: $A[f] \leftarrow \Phi_f$
15: **end for**
16: **end function**

capitalize the names of sets of a given metavariable (e.g., Φ is a set of formulas ϕ, etc.), and we use vector notation to indicate arrays (e.g., $\vec{s}$ is an array of statements s).

Building the assertion mapping. Each mock expresses the specification of an original function as it is encoded by that function's tests. To start, SKETCHAM collects assertions from those tests into an assertion mapping. Algorithm 1 builds the assertion mapping A from the input sketch p. The algorithm begins by initializing A to empty and Φ to the set of all assertions from all tests in p. It then selects two subsets of Φ. The set Φ_0 contains all assertions that do not include calls to any functions, and the set Φ_1 contains all assertions that include exactly one function call. We exclude assertions with multiple function calls so that mocks are standalone, to conform to the technical requirements Sketch imposes on models. As a consequence, we exclude some terms that present no such concerns (e.g., conjunctions of otherwise unrelated terms), as translating them to assumptions may be much more complex or even impossible.

For each function f called in an assertion in Φ_1, on line 7 we next compute the set Φ_f from Φ_0 (the assertions that hold throughout each test, including at calls to f) and the subset of Φ_1 that refers to f. For example, consider the assertion in `h_sort` in Figure 2.1d. This code refers to the result of calling `sort(n, vs)`, so $\Phi_1 = \{\phi_i(\texttt{sort(n,vs)})\}$, where the ϕ_is capture the assertions in `h_sort`. Additionally, if we picked, say, a loop unrolling bound of 4, then Sketch would implicitly `assert n<4`, resulting in $\Phi_0 = \{\texttt{n<4}\}$. In general, Φ_0 might contain additional assertions that are irrelevant to the calls in Φ_1. For example, loop unrolling for harness `h_dedup` (not shown) might add another bound `m<4` to Φ_0 for `sort`. However, such irrelevant assertions will not change the resulting mock.

In some cases, we cannot add assertions in Φ_f to A because other assertions on the same variables interfere. For example, suppose the sketch includes `assert f(x)` and `assert g(x)`. Then

Algorithm 2 Mock rewriting: generate mocks

Input: p - the sketch
Input: A - output of Algorithm 1
Output: p' - the sketch augmented with mock definitions
Output: F - finite mapping from an original function name to its mock

1: **function** GENERATEMOCKS(p, A)
2: $F \leftarrow \emptyset, p' \leftarrow p$
3: **for all** $f \mapsto \Phi \in A$ **do**
4: `def` $f(\vec{x})\{\ldots\}$ $\leftarrow$ the definition of f in p
5: $f_u \leftarrow$ FRESHNAME(f)
6: $\vec{s} \leftarrow [\,]$
7: $\Phi_0 \leftarrow \{\phi \in \Phi \mid 0 = |f(\ldots) \in \phi|\}$
8: $\Phi_1 \leftarrow \{\phi \in \Phi \mid 1 = |f(\ldots) \in \phi|\}$
9: **for all** $\phi \in \Phi_1$ **do** ▷ convert asserts into assumes
10: $f(\vec{e}) \leftarrow$ the lone function call in ϕ
11: $\phi_u \leftarrow \phi[f(\vec{e}) := f_u(\vec{x})]$ ▷ substitute uninterpreted function
12: $\Psi \leftarrow \{x_i = e_i \mid 0 \le i < |\vec{x}|\}$ ▷ equate parameters to arguments
13: $\phi' \leftarrow (\bigwedge \Phi_0) \wedge (\bigwedge \Psi) \implies \phi_u$ ▷ the condition where ϕ holds
14: $\phi'' \leftarrow [\![\text{FV}(\phi); \Psi \vdash \phi']\!]$
15: $\vec{s}$.append(`assume` ϕ'')
16: **end for**
17: $f_m \leftarrow$ FRESHNAME(f)
18: $F[f] \leftarrow f_m$
19: $d_m \leftarrow$ `def` $f_m(\vec{x})\{$ ▷ create the mock definition
$\vec{s}$
`return` $f_u(\vec{x})$
$\}$
20: p'.insert(d_m)
21: **end for**
22: **end function**

Φ_f might not completely characterize `f`—the assertion in Φ_f is valid only if `assert g(x)` also holds, which puts an unknown (until the full sketch is solved) constraint on `x`. Thus, in this case, our algorithm discards the assertions in Φ_f. More specifically, on line 9, the loop removes any $\phi \in \Phi_f$ whose free variables overlap with free variables outside of Φ_f. The process iterates in case free variable dependencies cascade. For example, the existence of `assert g(x)` would eliminate `assert f(x-y)`, which would in turn eliminate `assert f(y)`. The result is the transitive closure of the allowable assertions about each function.

Generate mocks. Next, Algorithm 2 iterates through each function in the domain of A, generating a corresponding mock to add to the augmented sketch p'. As it does so, it also builds a map F from function names to the names of the generated mocks.

For each $f \mapsto \Phi \in A$, GENERATEMOCKS begins by finding the definition of f and creating a corresponding freshly named uninterpreted function f_u. It then initializes $\vec{s}$, the assumptions to be inserted into the new mock body, to empty. Then, from each asserted formula $\phi \in \Phi$, the algorithm creates a formula ϕ_u by substituting the single function call $f(\vec{e})$ in ϕ with a call $f_u(\vec{x})$, where $\vec{x}$ are the formal parameters of f (line 11). Notice this call to f_u is the same no matter the original call

Algorithm 3 Mock rewriting: mock harnesses

Input: p' - the sketch from Algorithm 2
Input: F_1 - the name map from Algorithm 2
Output: p_m - the sketch augmented with mock harnesses

```
1:  function MockHarnesses(p', F)
2:      G ← CallGraph(p'), p_m ← p'
3:      for i ← 1, maximum mock call graph depth do
4:          F_{i+1} ← ∅
5:          for all def g(y⃗){s⃗} ∈ Callers(G, dom F_i) do        ▷ similarly, harness def
6:              g' ← FreshName(g)
7:              d' ← def g'(y⃗){                                  ▷ respectively, harness def
                        {s[f := f' | f ↦ f' ∈ F_i] | s ∈ s⃗}
                    }
8:              p_m.insert(d', before g)
9:              F_{i+1}[g] ← g'
10:         end for
11:     end for
12: end function
```

to f, which ensures the generated mock conforms to the technical requirements Sketch imposes on models. To encode the actual information at the call site, we next add a precondition. The algorithm constructs ϕ' (line 13), which is an implication denoting that ϕ_u holds if the ancillary asserts Φ_0, and the equalities $x_i = e_i$ from the call to f hold. One nuance we elide here is that Sketch augments all function calls with an additional explicit *path condition* parameter. The path condition records the branch taken for each conditional up to the point of the call, which makes it easier for Sketch to translate the IR into a SAT formula. For soundness, we include this path condition as a premise of ϕ' and assign f_u the path condition $\top$. Note that our implementation trims Φ_0 before adding it to ϕ' to the subset containing only the variables in $\vec{e}$.

Next, the algorithm performs quantifier elimination on ϕ', yielding ϕ'' (line 14). More precisely, $[\![\text{FV}(\phi); \Psi \vdash \phi']\!]$ eliminates variables in $\text{FV}(\phi)$ from ϕ', searching for witnesses in Ψ. Then, ϕ'' is added to $\vec{s}$ as an `assume`, and the loop continues until all mappings for f have been handled.

Finally, on lines 17-19 the algorithm computes a fresh Sketch name f_m for f, adds a mapping to F, and creates function definition d_m for f_m. The function f_m takes the same arguments as f, assumes all formulas in $\vec{s}$, and returns f_u on f_m's arguments. Thus, when f_m is called, the assertions about f from its original test suite in p are assumed on f_m's arguments, as we saw in Section 2.2. The definition d_m is added to p', and mock generation continues until all mappings in A have been traversed.

New mock harnesses. The last step of Sketcham adds calls to the mocks generated by GenerateMocks. One naïve approach would be to simply replace each call to f with a call to f_m for all $f \mapsto f_m \in F$. However, this will not work for two reasons. First, we need a full solution for the holes in all functions, including those that are mocked. Replacing calls to f with calls to f_m would remove many constraints on the holes in f, underconstraining their solutions. Second, as we saw

earlier the template for f might contain additional information excluded by BuildAssertMap, so replacing f by f_m might underconstrain f's callers.

Our solution is to create an output sketch that includes both the original sketch—including all calls to f in their original form—and duplicate sketch code that calls f_m in place of f. The duplicated code refers to the same holes as the original sketch. Hence, information derived from the duplicated code can potentially greatly speed up solving of the original code.

Algorithm 3 shows MockHarnesses, which creates this duplicate code. The algorithm begins by constructing a call graph G from the sketch p' from the previous step. Note that none of the mocks in p' are called yet, so the call graph is the same as for the original sketch. Next, the algorithm duplicates the sketch one level of the call stack at a time, starting at the mocks and working up toward the harnesses. To limit duplication, e.g., for mocks called by recursive functions whose duplication would loop infinitely, the algorithm bounds the duplication depth. For each level i, it iterates through all functions $g \in \textsc{Callers}(G, \operatorname{dom} F_i)$, meaning functions g that call a function in the domain of F_i. It duplicates each such g, replacing calls to functions $f \in \operatorname{dom} F_i$ with calls to $F_i[f]$, and then adds the duplicated function to the sketch. Since g has now been renamed, $g \mapsto g'$ is added to a new mapping F_{i+1}, and calls to it are duplicated in the next iteration, repeating until reaching the root of the call graph or the maximum duplication depth. Note the process is the same for both regular function definitions and for functions that are harnesses.

For example, suppose harness h calls function g, which in turn calls function f, and assume GenerateMocks created f_m and g_m. Then in the first iteration, MockHarnesses creates a duplicate h' that calls g_m and a duplicate g' that calls f_m. In the next and final iteration, it creates a duplicate h'' that calls g'.

When we insert the duplicate functions, we insert them *before* the original functions. Doing so ensures that when we insert the duplicate harnesses that call the mocks, Sketch will solve those harnesses before solving the original ones.

2.4 Evaluation

We evaluated Sketcham on ten benchmarks, running each from 11 to 1487 times until reaching statistically significant results. We found that, for six of ten benchmarks, Sketcham performs up to 5× faster than Sketch, for one benchmark Sketcham is slower by a factor of up to 0.9×, and for the remaining three benchmarks performance is indistinguishable. We examined the benchmark *dedup* (Figure 2.1) in depth and found that, as suspected, overall performance improvement is due to improved synthesis time when using `sort_mock`.

Implementation. Sketcham comprises approximately 1075 lines of C++ code within the Sketch backend. The user enables Sketcham with `-mock` and specifies the max mock duplication depth via `--bnd-mock-depth`, which defaults to 3.

Because they clone and then rearrange the input Sketch IR program, the run time of Algorithms 1-3 is approximately linear in the number of functions and the number of asserts in

the sketch. Our implementation covers the features given as part of the Sketch IR fragment in Figure 2.4, with the modification that we explicitly depict assignment, which Sketch IR does not require because it structurally hashes expressions to yield a compact in-memory representation [SL08].

Benchmarks. We used the following benchmarks:

- *double*, the integer doubling program given in Figure 2.2.
- *absval*, the absolute value function, shown in Figure A.1.
- *fib*, the linear-time Fibonacci function. The specification requires its output to be equivalent to the exponential time algorithm. See Figure A.2.
- *datetime*, a simplified implementation of the C `strptime` function. This function accepts a format that it uses to parse a date/time string.
- *boyerMoore*, which implements the Boyer Moore string search algorithm [BM77].
- *regex*, a regular expression matching engine and compiler.
- *spellcheck*, a program that suggests a corrected version of its input using the Levenshtein edit distance from entries in a dictionary. See Figures A.3 and A.4.
- *minpair*, uses edit distance to find the closest pair out of an array of values. See Figures A.3 and A.5.
- $dedup_i$, deduplication with insertion sort, shown in Figure A.6, and $dedup_m$, deduplication with merge sort, shown in Figure A.7. We saw a simplified version of $dedup_m$ earlier in Figure 2.1.

Sketch has a multitude of configuration options that can have a large effect on performance. The middle portion of Table 2.1 gives values for the four options that differ across the benchmarks: *int type*, whether Sketch uses symbolic integers (in either a bit-vector encoding or a sparse encoding [SL08]) or native integers [SL20]; *int bits*, the number of bits per integer; *loop unroll*, the maximum loop unrolling depth; and *func inline*, the maximum depth of function call inlining.

We selected values for these options that reflected each benchmark's design and demonstrated pronounced run time differences from Sketch to SKETCHAM, as follows. *double* and *absval* use Sketch's defaults. *fib* tests recursively computing the Fibonacci sequence up to the tenth entry, so function call inlining is set accordingly. *regex* is required to reject bad matches, which requires higher unrolling and inlining. *datetime*, *boyerMoore*, *spellcheck*, and *minpair* need higher loop unrolling to iterate over long strings. These last three and both *dedup*s also do much better using native integers. The *dedup*s also run unreasonably slowly with more bits or higher unroll, so we reduced the amount of unrolling. In all our benchmarks, any configuration options not discussed here were left as their defaults, including the mock duplication depth, with the default of 3.

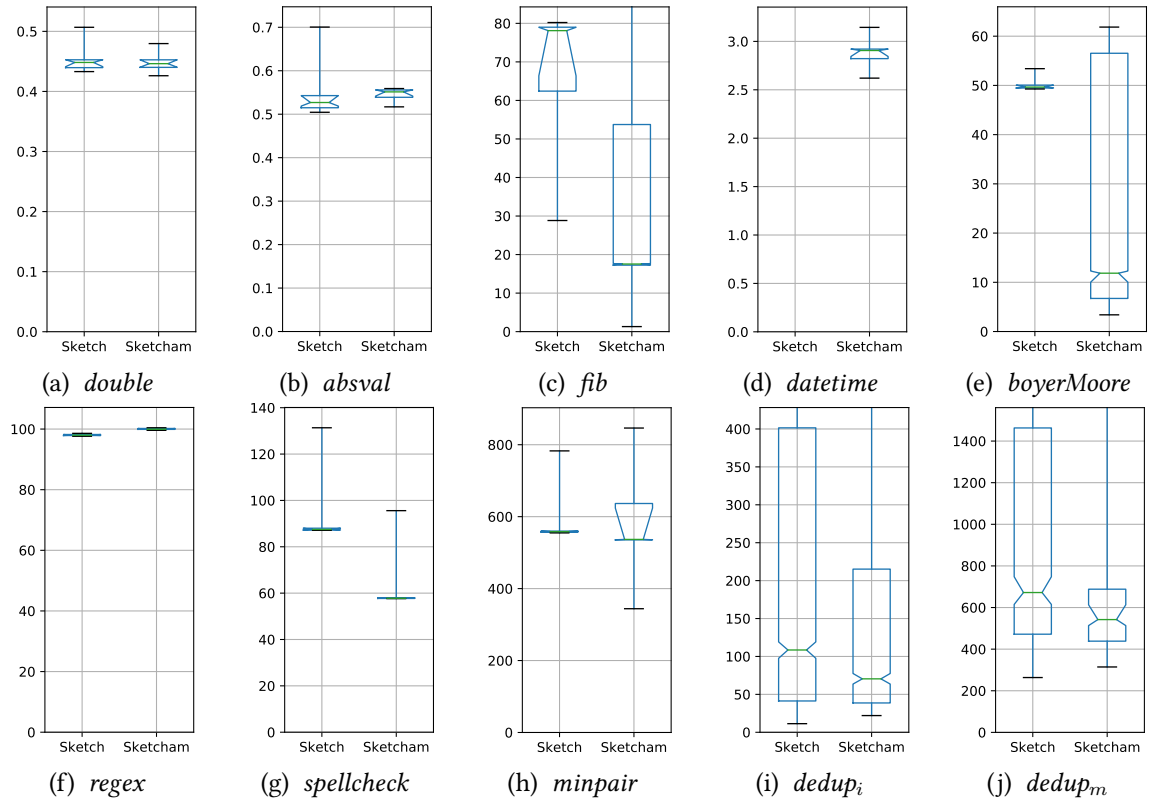
	# lines	# holes	int type	int bits	loop unroll	func inline	Sketch runs total	Sketch runs failed	SKETCHAM runs total	SKETCHAM runs failed
double	8	1	symbolic	5	8	5	17	0	17	0
absval	69	9	symbolic	5	8	5	17	0	17	0
fib	46	4	symbolic	6	8	10	20	0	65	0
datetime	177	3	symbolic	11	20	5	11	11	17	0
boyerMoore	136	16	native	7	13	5	17	0	153	19
regex	357	5604	symbolic	5	30	7	17	0	17	0
spellcheck	94	5	native	5	9	5	17	0	17	0
minpair	113	3	native	5	10	5	17	0	22	2
$dedup_i$	73	1134	native	2	4	5	1487	88	762	23
$dedup_m$	80	9008	native	2	4	5	648	281	88	16

Table 2.1: Benchmark config options and characteristics.

Methodology. All measurements were taken on a 3.2 GHz AMD Ryzen 5 1600 system with 32GB of RAM. We found that while most benchmarks consistently perform within half an order of magnitude under both Sketch and SKETCHAM, in a few cases synthesis time varies by as much as two and a half orders of magnitude. To account for this variance during our evaluation, we repeatedly ran each benchmark until achieving statistical significance, between 11 and 1487 times, as listed in the rightmost portion of Table 2.1. Each run was executed with the system otherwise almost totally idle to minimize interference. While most runs completed successfully, we exclude those that exceed a 60 minute timeout or fail to synthesize due to exhausting system memory or a crash within Sketch. To give an idea of the problem size, the leftmost portion of Table 2.1 lists the numbers of lines and holes per benchmark.

As other work has observed [GBE07], performance evaluation methodologies that lack rigor can lead to misleading and incorrect conclusions. To avoid this problem, we collect enough data to calculate a percentile's confidence interval (CI) at a given confidence level (CL). We employ the classic Clopper-Pearson [CP34] (or "exact") method using the probabilities of the Binomial distribution to iteratively calculate confidence intervals for a given dataset. While other methods are often used, many of these assume an underlying Gaussian distribution. The underlying distributions for our measurements are not known and do not appear to be Gaussian, a case the exact method handles correctly.

Run time variance is not correlated across configurations, so the number of runs needed for significance can differ from Sketch to SKETCHAM, as reflected in the "total" columns of Table 2.1. We ran each configuration repeatedly until measurements met two statistical significance conditions. First, that they reach a 95% CL that the population median lies within at most a 20% CI around the sample median. For example, for a sample median of 100s, the population median might lie between 90s and 110s, or between 98s and 118s, depending on the underlying distribution. Second, the CI must range between the first and third quartiles to increase the confidence that the median measurements adequately reflect the underlying distribution. In seven out of ten benchmarks these two conditions were sufficient to yield CIs that did not overlap across Sketch and SKETCHAM, which allows for statistically significant performance claims about these benchmarks.



(a) *double* (b) *absval* (c) *fib* (d) *datetime* (e) *boyerMoore*

(f) *regex* (g) *spellcheck* (h) *minpair* (i) $dedup_i$ (j) $dedup_m$

Figure 2.5: Total time (s). Times are drawn as notched box plots, which give the distribution's median inside a notch indicating its confidence interval. As usual, the box extends to the first and third quartiles, and whiskers extend to the full distribution. To better focus on the data, we truncate some whiskers. Note differing y-axis scales both here and below.

2.4.1 Performance

Figure 2.5 shows the running times of Sketch and SKETCHAM on our benchmarks. The distribution of times is shown as notched box plots. The boxes extend from the first to the third quartile, with the median shown as a mid-line. The CI is indicated by the notch. The whiskers extended to the minimum and maximum values (some whiskers are truncated to allow for a closer view of the median).

Following standard practice, we conclude that two configurations have a statistically significant difference in performance if their CIs do not overlap, as there is then high probability that the median times of the distributions are different. We see that for six of the ten benchmarks, SKETCHAM is faster than Sketch, while one is marginally slower and three display no significant performance change. We investigated each benchmark's performance in detail, discussed next. The performance differences we report are ratios of the run time of Sketch to SKETCHAM for a given benchmark. Due to uncertainty we report speedup ranges for the median, comparing the opposite extents of each CI. This comparison ranges from, at minimum, the ratio of the faster end of Sketch's CI to the slower end of SKETCHAM's CI, up to, at maximum, the ratio of the slower end of Sketch's CI to the faster end of SKETCHAM's CI.

The times shown are total run time, which can be broken down into synthesis, verification,

and overhead time. For Sketcham, overhead can further be broken down into mock construction and normal Sketch overhead. The total runtime overhead of mock construction is less than 0.4% for all benchmarks except regex (3%) and both dedups (∼20%). In most cases, this time was dominated by the GenerateMocks and BuildAssertMap phases.

The *double* benchmark's performance is approximately the same in both cases. In fact, the CIs overlap almost completely, suggesting the performance may be dominated by constant factors in Sketch.

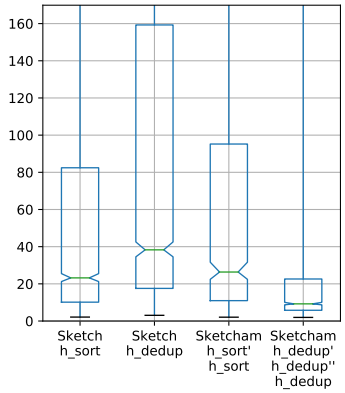
The *absval* benchmark is also approximately the same. It is another simple program that Sketch solves very quickly, and as such the mocks only add to the verification time.

The *fib* benchmark asserts that, on integers 0 to 9, the to-be-synthesized linear-time Fibonacci implementation returns the same result as an exponential-time implementation. In Sketch, the calls to the exponential-time algorithm cause a slowdown. But since Sketcham replaces calls to the exponential-time algorithm with calls to a (constant-time) mock, Sketcham achieves a speedup of 3.8–4.5×. While it is difficult to make out in the plot, the median and CI lie immediately above the first quartile for Sketcham.

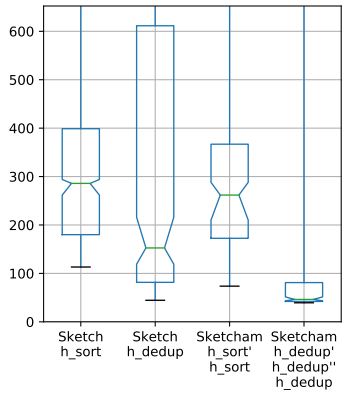
The *datetime* benchmark fails to synthesize in Sketch due to memory exhaustion, but it consistently synthesizes in just a few seconds using Sketcham. Investigating further, we found the bottleneck is a function that parses strings into integers in a loop that converts digits and adds them to a running total. For example, the digit sequence `abc` is converted to the integer `100*a+10*b+c`. This conversion loop is unrolled to the maximum bound by Sketch, and the input strings are of varying sizes, which is encoded as a separate formula for each possible length. The SAT conversion algorithm translates symbolic arithmetic formulas according to combinations of possible values of their subformulas, which results in very large SAT formulas in this case. Later in the conversion, these are merged back together in another quadratic operation. Due to the number of formulas and overall formula size, this translation eventually exhausts memory. While Sketcham technically faces the same issue, it does so after decomposing the sketch into smaller formulas, and thus these limits are never approached.

The *boyerMoore* benchmark runs 4–5× faster under Sketcham than Sketch. The reason is similar to the previous case. *boyerMoore* includes a generator that constructs arithmetic expressions that add and subtract a small set of values including a hole. Sketch constructs these expressions recursively so they grow quickly, with the total number of terms determined exponentially by the degree of function inlining, and the resulting expressions have high symmetry, both factors that slow down solving, further compounded by the location of this expression deep within the sketch. Because Sketcham breaks the problem's dependencies, this expression can be synthesized separately from the rest of the program, which proceeds much more quickly.

The *regex* benchmark's overall performance using Sketcham is statistically significantly slower by a factor of 0.98×, which is a minimal difference in practice. The main mocked function here performs compilation of a regular expression into instructions for a virtual machine. Because compilation is recursive, it is difficult to give a specification that Sketcham can use. It is instead given by example with an exhaustive set of subproblems, which greatly increases the number of harnesses to solve. While most harnesses keep similar performance and the slowest harness is 8%



(a) *dedup*$_i$



(b) *dedup*$_m$

Figure 2.6: Breakdown of *dedup* runtimes grouped by harness (s)

faster in SKETCHAM, this is not enough margin to improve overall solve time.

The *spellcheck* benchmark using SKETCHAM sees a speedup of 1.5×, while *minpair* performs roughly the same (0.89–1.04×). Both rely on the same Levenshtein edit distance algorithm. The harness for this algorithm, which is the most time-consuming in either sketch, runs last in both settings, which reveals the source of the performance difference between the two benchmarks. *minpair* is dominated by synthesis time and *spellcheck* by verification time, which means that harnesses for the minimum pair function are more difficult to synthesize than for the spellcheck function, and so the former accumulates more state within the solver that is compounded when solving the Levenshtein harness. This slows it down enough to decrease the overall performance. On the other hand, the improvement of *spellcheck* is distributed across all individual harnesses, and across both synthesis and verification time, more than making up for the time it takes to construct and solve the mock harnesses.

Finally, the *dedup*s show a notable performance improvement with SKETCHAM. In both *dedup*$_i$ and *dedup*$_m$, the problem is large and complex enough that plain Sketch struggles with it. SKETCHAM eliminates the interactions of holes across the deduplication and sorting functions, which speeds up synthesis by a factor of 1.3–1.9× for *dedup*$_i$ and 1.003–1.5× for *dedup*$_m$.

2.4.2 Case Study: Deduplication

Next, we examine the performance of *dedup*$_i$ and *dedup*$_m$ in detail, as they illustrate the strengths and weaknesses of SKETCHAM. We break our discussion into comparisons of solving time across harnesses and comparisons of CEGIS synthesis time to CEGIS verification time.

Time to Solve Each Harness. Both *dedup*$_i$ and *dedup*$_m$ are structured the same way, and SKETCHAM creates the harnesses and mocks shown in Figure 2.1c for both. Figure 2.6 breaks down the total times for *dedup*$_i$ and *dedup*$_m$, grouped by the harnesses for `sort` and for `dedup`. We exclude overheads such as time spent in mock construction, parsing the input, and reassembling the output.

We make several observations. First, comparing the first and third columns within each

subfigure, we see the time for solving `h_sort` is the same for Sketch and SKETCHAM. This makes sense because `h_sort'` adds no information—it calls mocked `sort` and then immediately asserts the same specification as in the mock. Note that, while the trivial `h_sort'` harness could be elided here, creating an analogous harness would be useful if the harness accidentally contained a contradiction. In such a case, SKETCHAM would almost instantly decide the harness is unsatisfiable, whereas Sketch could spend an arbitrary amount of time reasoning about the computation in the actual called function before detecting the contradiction.

Second, comparing the second and fourth columns within each subfigure, we see that the CI of `h_dedup` using SKETCHAM lies well below the CI using Sketch. The speed improves by a factor of 3.2–4.7× for $dedup_i$ and 2.2–4.9× for $dedup_m$. Examining this result in detail, we find that SKETCHAM works exactly as intended: `h_dedup''` calls the mocked `sort`, enabling it to synthesize quickly and assign holes correctly, which are then simply verified when checking `h_dedup` (and `h_dedup'` is trivial, similarly to `h_sort'`).

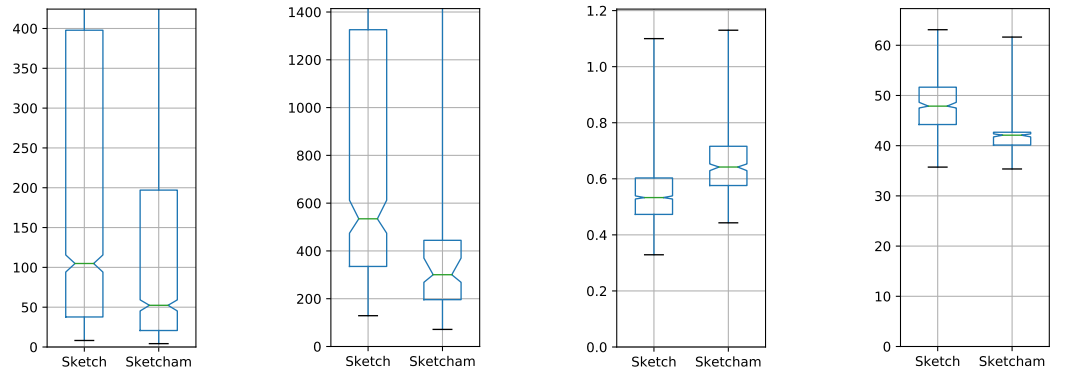
Third, also comparing the second and fourth columns, we see the variance in performance for Sketch is much greater than for SKETCHAM. Investigating further, we found this increase occurs for two reasons. First, the specification in `h_sort` is weak enough[2] that sometimes an incorrect hole assignment for `sort` satisfies the verifier and is only discovered while synthesizing `h_dedup`, forcing the solver to backtrack at great cost and simultaneously consider the holes in both functions. Second, even when the solver finds a correct assignment for `sort`, it includes the entire formula again while solving `h_dedup`, resulting in a much larger problem and corresponding variability. In contrast, with SKETCHAM, `h_dedup''` is decoupled from `sort`, eliminating these issues.

Fourth, we observe that both Sketch and SKETCHAM can solve `h_sort` about 10× faster for $dedup_i$ than for $dedup_m$. Overall, merge sort is more challenging for Sketch than insertion sort (note that since Sketch finitizes the problem by, e.g., unrolling loops, asymptotic complexity does not play a role). More surprisingly, synthesizing `h_dedup` is also faster for $dedup_i$ compared to $dedup_m$. We believe this occurs because synthesis of `h_dedup` must sometimes recover from a bad hole assignment from `h_sort`, which will be quicker for $dedup_i$, and because the easier synthesis of $dedup_i$ means the solver accumulates less state, such as conflict clauses, that would otherwise slow down solving subsequent harnesses.

Finally, we begin to get a clearer picture of the divergence between $dedup_i$ and $dedup_m$. In $dedup_i$, `h_dedup` synthesis is the performance driver, and the improvement using SKETCHAM has a significant impact on total performance improvement. In $dedup_m$ it is overshadowed by `h_sort`, which dominates to the point that improvement elsewhere is not as significant a contributor. Combined with the overhead of mock construction, this leads to a less pronounced improvement in total performance.

Synthesis and Verification Time. Figure 2.7 shows the times for the CEGIS synthesis phase and verification phase for each benchmark under Sketch and SKETCHAM. Not shown are the overheads of mock construction, parsing, etc., which for $dedup_i$ we found took 3–4s in Sketch versus

[2] In addition to the specification we have supplied, a complete specification of `sort` relies on the existence of a permutation function over the array's indices.



(a) $dedup_i$ synthesis (b) $dedup_m$ synthesis (c) $dedup_i$ verification (d) $dedup_m$ verification

Figure 2.7: Breakdown of *dedup* runtimes grouped by phase (s)

17–19s in Sketcham, and for $dedup_m$ took 90–96s in Sketch versus 201–207s using Sketcham. We believe much of the difference between these could be eliminated with additional engineering effort.

Looking at verification times in Figures 2.7c and 2.7d, we see that while the verification times for Sketch and Sketcham are different, they are still relatively close: Sketcham is 0.81–0.86× slower for $dedup_i$ and 1.12–1.16× faster for $dedup_m$. In contrast, comparing synthesis times in Figures 2.7a and 2.7b, we see a more significant speedup for Sketcham over Sketch: 1.59–2.55× for $dedup_i$ and 1.28–2.28× for $dedup_m$. Moreover, if we compare synthesis and verification time, we see that the overall solving time for both benchmarks is dominated by synthesis time. Indeed, we observed even greater synthesis speedups on other benchmarks including *fib* (4.2–5.1×) and *boyerMoore* 5.2–6.9×, but the most extreme of which was *spellcheck*, which saw synthesis speed up by 308.4–345.7× using Sketcham. Thus, we find that Sketcham's performance improvements come from reducing synthesis time by introducing mocks that decrease the number of holes that need to be considered at once.

2.4.3 Discussion

In general, we found that Sketch's performance is unpredictable in practice due to the use of randomness in the algorithm. For example, in terms of overall solving time, our experimental runs included several outliers (not shown in Figure 2.5) near the 60 minute timeout. In these cases, Sketch essentially makes a very poor initial guess for the holes, and verification produces counterexamples that do not add much information. Both Sketch and Sketcham exhibit this issue.

Moreover, often what seem like minor changes in the program sketch or configuration options can result in totally different solver behavior, and hence performance. One example of this was *boyerMoore*, which turned out to be non-linearly sensitive to the loop unrolling parameter. This benchmark was also extremely fickle about the problem formulation—holes in what seemed to be innocuous locations would lead to timeouts in both Sketch and Sketcham. Another example is *dedup*, which initially had a specification that omitted a requirement that the output array did not have a negative length. Without this constraint, the performance benefit of Sketcham was overwhelmed by the variability of the solver exploring ultimately impossible scenarios.

Overall, our results suggest that while Sketcham can't always outperform plain Sketch, it performs best on problems split into functions whose tests cover the behavior the sketch actually relies on while being easier to compute than the functions' actual implementations. While Sketcham affected the performance of both CEGIS phases, the best improvements were observed when the solving time of dependencies was dominated by the synthesis phase. For programs with these properties, Sketcham can exhibit a performance improvement of as much as 5× overall, with synthesis time improvements alone of up to 345.7×. Moreover, in some cases, such as *datetime*, Sketcham can solve problems that are out of reach of plain Sketch. For programs where these properties do not hold, Sketcham performance is typically similar to plain Sketch.

2.5 Related Work

There are several threads of related work.

Program Synthesis with models. As discussed earlier, our work builds on work by Singh et al. [SSX+14], who propose manually created models for Sketch. While Sketcham relies on the core algorithm of that work, Sketcham frees the Sketch user from needing to write models, because we create mocks automatically from normal sketches. Mariano et al. [MRX+19] use algebraic specifications to model libraries. In contrast, our approach derives specifications from the input program's assertions, without requiring the programmer to add annotations.

Deriving mocks and specs from tests. Saff et al. [SAPE05] use the capture and replay of actual test executions to automatically generate mock dependencies with the goal of speeding up test execution. Fazzini et al. [FGO20] further generalize this capture-and-replay technique to consistently model the environment of a mobile app under test, allowing for testing apps that use an inconsistent resource like a database or network device. Both of these target normal testing rather than synthesis. Nguyen et al. [NWKF17] leverage symbolic execution over input-output test pairs to perform program repair. However, they use these tests to model individual expressions instead of modeling entire functions. The insight underlying these approaches is similar to ours, however Sketcham is capable of both input-output pairs and general properties, and does not rely on either concrete or symbolic execution of tests.

Component-based synthesis. Gulwani et al. [GJTV11] model programs using logical input-output relations to synthesize loop-free bit-vector programs. Shi et al. [SSL19] combine many solutions that each only partially meet a specification into one that meets the entire specification. Both approaches limit the synthesis search space by building their solutions from the bottom up, from a selection of base components. Smith and Albarghouthi [SA19] prune the search space using bottom up algebraic rewriting of the program into an equivalent normal form. In contrast to these, Sketcham derives its benefits from breaking apart input sketches from the top down, at function level granularity.

Modular synthesis using symbolic or actual execution. Samak et al. [SKR19] derive specifications of class methods using symbolic execution and use them to synthesize a replacement shim class one method at a time. Van Geffen et al. [VGND+20] use symbolic execution to model abstract virtual machines to modularly synthesize a compiler one instruction at a time. In contrast, because

our approach derives mocks directly from the input's assertions, we need not consider the code itself when modeling it. Hua et al. [HZZK19] modularize the synthesis of library calls through execution of actual partial programs. In contrast, we attempt to avoid called functions entirely by relying on their inferred specifications.

Other approaches. Bodík et al. [BCG+10a] finalize incomplete programs using angelic nondeterminism. In contrast, SKETCHAM does not introduce arbitrary angelic values, but instead constrains any angelic-like behavior using a function's inferred specification. Huang et al. [HQSW20a] use a divide-and-conquer strategy to iteratively split synthesis problems according to heuristics. In contrast, SKETCHAM splits problems structurally in a single pass. Polikarpova et al. [PKSL16] speed up synthesis through modular verification using refinement types. In contrast, our approach achieves a similar kind of modularity without being type-directed.

2.6 Summary

Modular synthesis is the technique of breaking apart large synthesis problems to synthesize their pieces individually. SKETCHAM is an approach to modular synthesis in the Sketch language that uses specification meta information learned from a sketch's existing harnesses themselves to construct mock implementations of functions. The mocks can be used in place of calls to the originals, decoupling the use of a function from its implementation. By constructing new subproblems in this manner, each function can be synthesized in relative isolation. The algorithm presents these subproblems to the synthesizer in order of dependency, followed finally by the original sketch to guarantee correctness. During our evaluation, we observed that SKETCHAM can speed up synthesis by as much as a factor of $5\times$, and was at worst a marginal $0.98\times$ slower.

Ultimately, while the SKETCHAM approach is a promising approach that can benefit synthesis, it is limited to highly complicated yet highly decomposable sketches. Program transformations in general are also often costly, and when applied outside of their domain are just as likely to slow synthesis down more than solving the problem as it was originally stated. For a more broadly applicable approach, we have to leave behind modularization and explore an entirely new sketching synthesis system that, rather than finding satisfying solutions, searches for optimal ones. In the next chapter we change gears to look at the optimization-aided core language of that system.

Chapter 3

SCIMITAR: Functional Programs as Optimization Problems

3.1 Introduction

This Chapter explores the *optimization-aided language* SCIMITAR, that enables seamless integration of mixed integer linear programming (*MILP*) constraint optimization problems into functional programs, a necessary building block for optimal synthesis. Unlike the imperative Sketch language discussed in the previous chapter, the functional paradigm is stateless, a design feature which helps to simplify modeling optimization problems.

MILP has limited expressiveness, which prevents users from easily encoding complex semantic concepts. SCIMITAR overcomes this using a high level language with a compiler that transforms a user's program into MILP using expert encodings.

Our approach is inspired by the *Satisfiability Modulo Theories (SMT)* solver-aided language Rosette. Like Rosette, SCIMITAR combines functional and symbolic reasoning in the same program, but unlike Rosette, SCIMITAR is optimization-aided—specifically by a MILP solver—rather than SMT-aided. SCIMITAR provides the (`minimize o e`) construct, which minimizes the objective `o` with respect to the constraints in the body expression `e`. This expression contains function calls, conditionals, loops, and more. It is compiled by SCIMITAR into optimization problems, which relieves SCIMITAR programmers of the burden of encoding these constructs by hand, enabling a level of abstraction not available in normal optimization problem representations. Section 3.2 gives two examples demonstrating our language design while explaining SCIMITAR's behavior in detail.

Another difference is that SCIMITAR separates host semantics from the solver semantics used in the body of `minimize`. This distinction is important because it allows users to distinguish between normal execution and optimization solving. Host semantics behaves the same as any normal functional language, while solver semantics translate source code into MILP problems in *standard form*, which it then submits to a solver. SCIMITAR also includes a type system that tracks variable bounds, which are required by optimization solvers. Section 3.3 gives a grammar and formal semantics for SCIMITAR that precisely captures how functional and optimization code interact.

Scimitar uses a wide range of encodings to lower functional programming features to optimization code, while Rosette relies directly on its underlying SMT solver, Z3, for its smtlib2 encoding and built-in translations. Scimitar has encodings for features such as multiplication, booleans, conditionals, and dynamic vector indexing. Since solver problems cannot model infinite loops or recursion, Scimitar must also perform function call inlining and loop unrolling during compilation, which emplace the body of a loop or function at each occurrence up to some user defined number of repetitions. Section 3.4 describes these encodings.

Finally, the applications of these languages are quite distinct. Users who are focused on numerical applications that require optimal values should generally prefer to use Scimitar, while those interested in purely discrete logical domains or have no need for optimal values will benefit more from using Rosette. Concrete examples of applications given by Rosette's developers include formal verification of systems such as Just-In-Time (JIT) compilers, synthesis of GPU kernels, program repair, and model checking. On the other hand, Scimitar targets applications such as logistics problems, resource allocation, and design optimization. Section 3.5 presents Scimitar benchmarks in different domains to demonstrate the capabilities in our system's design. We measure performance, giving evidence that a dedicated MILP solver compares favorably with the more general purpose SMT solver on optimization applications.

3.2 Examples

Scimitar is a unique language design that lets a user seamlessly combine optimization with classical computation. It combines these paradigms through an integrated MILP solver, which is controlled via constraint assertions and an optimization construct. Scimitar provides functional features including recursive, anonymous, and (limited) higher-order functions; conditionals; let bindings; and values including numbers, booleans, and tuples.

This section introduces Scimitar via two examples, a recursive summation function and an arena allocator. Section 3.5 walks through several more examples.

3.2.1 Sum from Zero to N

Scimitar programs are written as a specialized language within Racket. In addition to standard functional language features, Scimitar includes a form `(minimize o e)`, where expression `o` is the objective function to minimize subject to constraints in the expression `e` over some set of symbolic variables scoped to that term. These are variables whose values are to be decided by the solver, declared using the `(symbolic)` term. For example, Figure 3.1 shows a simple Scimitar program that finds the minimum value of the symbolic variable `n` subject to the constraint $\sum_{i=0}^{n} i \geq 100.0$. This constraint uses the recursive function `sum-to-n`, defined within the minimization expression with two variables `n` and `acc`. On line 6, if `n` is zero, `sum-to-n` returns `acc`. Otherwise it recurses, decrementing `n` and adding `n` to the accumulator. When executed, the final result is $n = 14$.

Normal program semantics executes with a known n and calculates a final value sequentially. On the other hand, solver semantics must model the whole program at once, divining the initial n from the constraint on the result value.

```
1 (minimize n
2  (define n (symbolic))
3  (letrec
4   ((sum-to-n
5     (lambda (n acc)
6      (if (= n 0.0) acc
7       (sum-to-n (- n 1.0) (+ n acc))))))
8   (assert (>= (sum-to-n n 0.0) 100.0))))
```

Figure 3.1: Example of a recursive function

The recursion in `sum-to-n` effectively creates a constraint against the conditional branches at each recursion depth. Recursion is handled via inlining, which relies on a sentinel constraint to indicate dynamically to the solver that the inlining depth has been reached. The constraint on line 8 is finally satisfied when $n + (n-1) \ldots + 0.0 \geq 100.0$, and the value of n is chosen.

In solver semantics, the entire **`if`** expression on lines 6-7 is active, including the guard and both branches. Branches taken can be picked dynamically, and can even decide the guard. This example forces the solver to choose between branches according to the accumulator value rather than n.

However, during normal execution both branches cannot be active simultaneously, so the solver must have a way to enable one and disable the other depending on the result of the condition expression. Enabling branches dynamically can not be done directly, but it can be accomplished indirectly using the encoding $[\![\text{if } c \text{ then } t \text{ else } f]\!] = c \cdot t + (1-c) \cdot f$. The condition expression `(= n 0.0)` has a binary type, and we use that as the indicator variable c. If c is true, the result of the true branch t is multiplied by one, and the false f by zero, and vice versa. I.e., if $c = 0$, then $0 \cdot t + (1-0) \cdot f = f$, which selects the false branch. We expand this multiplication using McCormick envelopes, discussed below in Section 3.2.2.

Scimitar's *functional language compiler* translates this example to a restricted constraint language via this encoding as well as many others. From there, the *optimization language compiler* outputs a matrix format understood by Scimitar's virtual machine, which repeatedly invokes a MILP solver and decodes the optimum into a *continuation*, a pair of a function and its input values, to then load and execute.

This example demonstrates how in the high level language users can reason about each constraint sub-problem independently, and Scimitar does the heavy lifting of composing these into an overall problem. By considering a term in isolation, users can ignore its structural relationship with adjacent terms. While such decomposability does not make reasoning about the solver's decision process for a particular program any easier, it does provide some guarantees about the program's construction. This is in contrast to normal optimization problems, where users must put in effort to tie parts of their problem together, which later on could represent a maintainability hurdle. Our goal is to allow users to implement certain classes of problems, including traditional optimization problems such as those presented in Section 3.5.

```
1 (define arena (Init-Arena))
2 (define (Allocate size)
3   (let ((bucket-ix (Bucket-For-Size size)))
4     ...
5     (Update-Arena-Dist arena)
6     (Grab-Block bucket-ix)))
```

Figure 3.2: Allocate example: `Allocate`

3.2.2 Arena Allocator

The previous section gives a basic example of a top-level **`minimize`** expression, but in general Scimitar allows such expressions within the body of a larger program, which enables iteratively solving the problem. Figures 3.2 and 3.3 explore Scimitar's *host-solver boundary* with a more complex example.

This Section demonstrates a memory allocator like malloc. Some malloc implementations use an arena of buckets based on allocation size, with allocations drawn from the least greater sized bucket. When users request a block of memory, the allocator retrieves one from the appropriate bucket. Because the dynamics of memory allocation cannot be predicted, it is impossible for an algorithm to perfectly balance the arena statically. Over time, some buckets are used more than others, and the logical solution is to dynamically rebalance the buckets to prevent that.

Figure 3.2 defines `Allocate`, a stateful allocator function that follows this design. It is responsible for the mutable state, including the current arena as well as the history of arenas, used for hysteresis. Line 1 initializes the global constant `arena`, the number of buckets for each size. The exact blocks the arena uses are managed outside of the optimization, and are not shown.

Line 3 selects the bucket containing the requested block, which is then updated to ensure one is available. This bucket is used in various other operations that we omit for brevity. The call to `Update-Arena-Dist` on line 5 rebalances the buckets according to the user's algorithm. We discuss our implementation of `Update-Arena-Dist` below. Finally, line 6 returns a new block to the user.

A straightforward way to rebalance buckets is to use heuristics such as high water mark or average value. Most heuristics may not be optimal overall, and some might not be easily specified.

Our implementation adjusts the number of slots within each bucket via an optimization problem. For explanatory purposes, we give an outline for one possible version of this in Figure 3.3; our evaluation includes a benchmark with a different approach to the problem in Figure D.2. The solution to this problem redistributes slots from an underused bucket to an overused one within the arena. It passes values back and forth cyclically until finally settling on a result that satisfies both host and solver. The goal of the implementation is to try to approximate an optimal future bucket layout based off of knowledge of past allocation behavior. Note that we make a number of simplifying assumptions here—a real implementation must handle all the complexity of a real system.

This example uses three new features—**`for`**, `vec-ref`, and functions over vectors. Iteration `(for ([i (range n)]) e)` can perform some computation `e` for index `i` from `0` through `n`. Vector

```
7  (define (Update-Arena-Dist old-arena)
8   (let ((new-arena
9    (optimum-ref optimal-arena
10    (minimize mem-usage
11     (define optimal-arena (symbolic))
12     (define mem-usage (symbolic))
13     (define derate (symbolic))
14     (define adjust-bucket (symbolic))
15     (assert (>= optimal-arena
16                 (... history derate)))
17     (for ([b (range bucket-count)])
18      (if (vec-ref adjust-bucket b)
19       (assert (... derate outliers))
20       (assert (... derate))))
21     ;; ... plus additional constraints to
22     ;; establish the relationship between
23     ;; mem-usage and optimal-arena ...
24     ))))
25   (if (Converges? new-arena old-arena)
26    (Set-Arena new-arena)
27    (Update-Arena-Dist new-arena))))
```

Figure 3.3: Allocate example: `Update-Arena-Dist`

indexing (`vec-ref v k`) retrieves the `k`th index of the vector `v`. Both the index `k` and the upper bound `n` can be concrete values or symbolic variables. We discuss these in Section 3.4.

Line 7 defines `Update-Arena-Dist`, a recursive function that calculates a new arena by minimizing the total memory according to various constraints until it converges. On line 10 we cross the host-solver boundary via the **`minimize`** expression, going from normal functional semantics to Scimitar's solver semantics. Note how the syntax does not change across the transition, allowing programmers to focus on the semantics of their problem. See Section 3.3.1 for more details. This **`minimize`** expression introduces the symbolic `optimal-arena`, `mem-usage`, `derate`, and `adjust-bucket` variables. The result of the expression minimizes the total memory used by the new arena, while retrieving the arena itself. The constraints shown make up the core heuristic, which relies on the two symbolic variables `derate` and `adjust-bucket` to incorporate weighted allocation trends into the current arena via hysteresis. We omit a few constraints establishing the relationship between the total memory and the new arena. Conceptually, these constraints flow the minimization of the memory usage down to the individual buckets, weighted by their sizes.

The implementation of that heuristic uses the three features mentioned above. We determine `derate` dynamically by looping over the boolean vector `adjust-bucket` (up to the parameter `bucket-count`), starting on line 17. We access each bucket one at a time to determine the manner in which `derate` is constrained. The number of true buckets is upper bounded by a parameter set by the system. The conditional considers both derating possibilities simultaneously. If true for a particular bucket (line 19), its derating factor is constrained by a call to a function, including some `outliers` parameter known to the system. If false (line 20), `derate` is constrained by some different function that does not depend on the `outliers`.

$$
\begin{array}{lll}
e ::= & x \mid v \mid \text{symbolic} \mid \text{letrec } f \Leftarrow e \text{ in } e & \mathcal{C} ::= \; e \preceq e \\
\mid & \text{if } e \text{ then } e \text{ else } e \mid e\,;\ldots;\,e \mid \lambda\, x\,.\,e \mid e\;e & v ::= \; () \mid n \mid \alpha \mid \langle\, e\,,\ldots,\,e\,\rangle \\
\mid & \text{for } x \Leftarrow e \text{ do } e \mid \text{sum } x \Leftarrow e \text{ of } e & o ::= \; x \mid n \mid \alpha \mid o \cdot o \mid o + o \\
\mid & e + e \mid e \cdot e \mid \text{ref } e\;e \mid (\, e\,,\ldots,\,e\,) & x, y \in \text{vars} \quad n \in \mathbb{Z} \quad \alpha, \beta \in \mathbb{R} \\
\mid & \text{assert } \mathcal{C} \mid \text{optimum-ref } x\;e \mid \text{minimize } o\;e & f, g \in \text{funcs}
\end{array}
$$

Figure 3.4: The Scimitar source language

The interaction between the two variables is subtle, and it may be the case that attempting to optimize the new arena according to the hysteresis criteria may fail some non-linear allocator convergence criteria. The algorithm transitions on line 25 back to the host semantics, then tests whether the new and old arenas converge. If the new arena has converged, then on line 26 the algorithm sets the arena and returns. Otherwise (line 27) it recurses with the new value. A real implementation would schedule rebalancing on some cadence instead of recursing, and after convergence to the steady state distribution would presumably switch to some cheaper but more approximate algorithm such as high water mark.

Note that unlike Figure 3.1, this recursion uses host semantics. Rather than creating a single optimization problem, host semantics generate a series of optimization problems executed one at a time. This execution strategy is made easy by the design of Scimitar's virtual machine, which directly integrates continuations, easily allowing for such chaining (see Section 3.3.3). We discuss recursion using solver semantics in Section 3.4.2.

3.3 Functional Language

Figure 3.4 shows Scimitar's source language. The language includes named variables x; values v; the symbolic variable constructor; a letrec $f \Leftarrow e1$ in $e2$ expression that defines a function f, whose body $e1$ is always a lambda; conditionals if e then e else e; lambdas $\lambda x\,.\,e$ and function application $e\;e$; sequences of expressions $e\,;\,\ldots\,;\,e$, executing each expression for the constraints it might introduce and returning the result of the final expression; loops for $x \Leftarrow e$ do e that iterate through a tuple, assigning each tuple element in turn to x and evaluating the loop body; a native summation operation sum $x \Leftarrow e$ of e that is more efficient that summing via a loop; addition $e + e$; multiplication $e \cdot e$[1]; vector indexing ref $e\;e$; and tuples $(\, e\,,\,\ldots\,,\,e\,)$.

The language also includes the assertion form assert $\mathcal{C}$, which generates the constraint $\mathcal{C}$. Without loss of generality, in the grammar we write constraints as $e \preceq e$ where, as is standard in the optimization literature, $\preceq$ stands for $\geq$, $=$, or $\leq$. To specify the optimization goal, the language has *solve expressions* of the form minimize $o\;e$, where the objective o indicates the expression to be minimized under the constraints generated in the body e. Objectives o are a syntactic subset of expressions. The result of a minimize expression is a *solution record*, a data structure with concrete assignments for all variables mentioned in the objective. The form optimum-ref $x\;e$ returns the solution for x from the solution record e. Note that while the grammar here is limited to minimize, our implementation also supports a maximize solve expression.

[1]Note that Scimitar does not support multiplying two continuous variables—see Section 3.4 for details.

$$o = \sum_{i=1}^{n} \alpha_i \cdot y_i + \alpha \cdot y + \beta$$

$$\min \alpha_1 \cdot y_1 + \ldots + \alpha_n \cdot y_n + \beta; \mathcal{C} \vdash e \rightsquigarrow y$$

$$[\![o]\!] = c^T x \qquad [\![\mathcal{C}]\!] = Ax \preceq b$$

$$\frac{\text{solve}(\text{argmin}\, c^T x \,\text{s.t.}\, Ax \preceq b) \to x^\star \qquad [\![v]\!] = x^\star}{\textsf{minimize}\ o\ e \Downarrow v}\ \textsc{Solve}$$

$$\frac{e \Downarrow v}{\textsf{optimum-ref}\ x\ e \Downarrow v[x]}\ \textsc{OptRef}$$

Figure 3.5: The solve rules of the functional host semantics
The judgment form here is the usual big-step $e \Downarrow v$

Values in Scimitar are unit, numbers, and vectors. One deviation from the usual functional language semantics is that Scimitar does not support general lists, with list syntax instead denoting fixed-width tuples. The omission of variable-length lists is due primarily to Scimitar's inability to express general algebraic data types, a limitation born from the solver's requirement of a finite problem size (see Section 3.3.4 for more details).

As stated previously, programs in this language are sent to the functional language compiler, which outputs code in $\mathcal{O}$, the optimization problem language. $\mathcal{O}$ is almost a strict subset of the grammar presented in Figure 3.4. The only exception is the addition of a form to declare named constraint problems we term *primatives*. For a complete formalization, see Appendix C.

3.3.1 Semantics

Scimitar uses two sets of semantics: *functional host semantics* and *functional solver semantics*. Scimitar crosses this *host-solver boundary* via solve expressions.

Top-level code outside of solve expressions has the standard scheme-like semantics. Figure 3.5 gives a specification of the functional host semantics' minimize and optimum-ref rules. The Solve rule minimizes some objective o subject to the constraints induced by the expression e to produce the solution record value v. The user supplies the objective o in a form that Scimitar can compile down to the equation $o = \sum_{i=1}^{n} \alpha_i \cdot y_i + \alpha \cdot y + \beta$.

To evaluate e, we switch to the functional solver semantics. Instead of executing e as in the host semantics, the solver semantics derive an equivalent optimization problem, if it exists, via angelic nondeterminism [BCG+10b]. The solution to this problem is an assignment for the objective that, when substituted into the body of e, yields a program that correctly obeys the host semantics. The judgment form used in the solver semantics is trace-based. The relation $e \rightsquigarrow y$ states that the solver can only reason about the operation e in a context where the solver is aware of some result variable y. The solvability of this relation is contingent on the existence of some equifeasible MILP problem that minimizes an objective subject to constraints $\mathcal{C}$ over the objective and result variables. For such an equifeasible MILP problem to exist, the variables and constants

in its objective and result must be a superset of those requested by the user-supplied objective o. While angelic in theory, in practice $\mathcal{C}$ and y are intermediate products of compilation. For a complete discussion of the functional solver semantics, please see Appendix B.2.

The SCIMITAR objective and constraints are converted to the solver's vector and matrix format $\operatorname{argmin} c^T x \,\text{s. t. } Ax \preceq b$ known as *standard form*. Standard form minimizes the function $c^T x$, where x is a vector comprised of all program variables $\langle y_1, \ldots, y_n, y \rangle$ for the solver to decide, and c is the given coefficient vector $\langle \alpha_1, \ldots, \alpha_n, \alpha \rangle$ (we ignore β since it doesn't impact the argument values). In the constraints $Ax \preceq b$, the term A is the matrix of coefficients, and b is the vector of bounds. Once assembled, this standard form problem is submitted directly to the solver.

The result is the vector of the optimal values of x, known as the *optimal point* $x^\star$. We convert $x^\star$ into the opaque SCIMITAR solution record v, whose members are accessed by OPTREF.

3.3.2 Types

SCIMITAR includes a type system built around vector shapes and value sets. The language includes a variety of types, but they are ultimately converted into $\mathcal{I}^\mu$ (vectors of shape μ over some interval $\mathcal{I}$). Of particular interest are the set of reals $\mathbb{R}$ and the set $\{0, 1\}$. We discuss types in detail in Appendix B.1.

3.3.3 Virtual Machine

SCIMITAR's virtual machine uses a CPS execution model internally. The compiler breaks code blocks into parameterized continuations, which it stores in a table with an associated key. The virtual machine loop looks up a key, loads the continuation, applies parameters, and finally executes it. When complete, each continuation returns the key for the next one to execute.

Because SCIMITAR supports first class functions, optimization problems can take advantage of this execution model by using continuations to direct control flow dynamically across the host–solver boundary (for more on this topic, see Sections 3.2.2 and 3.3.1).

3.3.4 Solver Awareness

To use SCIMITAR, users must sometimes be aware of the solver's behavior and configuration. Solvers themselves have limits, like allowable solver values and finite problem size.

For example, different solvers have different numeric limits internally, which restricts the algorithm's precision and the values it can use. To control this behavior, SCIMITAR users may have to configure the largest and smallest allowed values before compilation to ensure the solver can handle all values needed by the program. For example, the *contradict* benchmark (Section 3.5) with the default bounds is infeasible using Gurobi. By setting the bounds within 6 orders of magnitude from largest to smallest value, it solves correctly. In general, multiplication and conditional constraints may introduce such precision issues. To avoid them, programmers must take care that values used in those expressions have tighter bounds on their type (see Section 3.3.2).

While programs written by users may have unbounded behavior, problems in the solver must be finite, which comes into play with constant propagation in loop unrolling, as we discuss

$$
\begin{array}{rcl}
[\![x \cdot y]\!] & \geq & x^l \cdot y + x \cdot y^l - x^l \cdot y^l \\
[\![x \cdot y]\!] & \geq & x^u \cdot y + x \cdot y^u - x^u \cdot y^u \\
[\![x \cdot y]\!] & \leq & x^l \cdot y + x \cdot y^u - x^l \cdot y^u \\
[\![x \cdot y]\!] & \leq & x^u \cdot y + x \cdot y^l - x^u \cdot y^l
\end{array}
$$

Figure 3.6: Encoding of McCormick envelopes

in more detail in Section 3.4.2. If a loop's bounds can be reduced to constant numeric values via constant propagation, Scimitar unrolls the loop precisely that many times, propagating the loop variable's value for each iteration. However, if the loop's bounds cannot be statically determined, e.g., if they use unknown optimization variables, then Scimitar can not know how much to unroll. The language must still finitize the program though, so it employs a common [SL20] *approximate* solution to the problem by automatically unrolling the loop up to a user supplied loop unroll bound parameter. The user has to be sure that Scimitar is configured with an unroll limit that is sufficient to solve the problem. They must also take care that the limit is tight, because if it is too high, the solve time can increase dramatically. The user must be aware of these limitations when designing their programs.

3.4 Encoding Scimitar to Constraints

Scimitar uses a range of techniques and strategies to encode high-level program features. This Section demonstrates some of the most important ones. We also discuss some of the areas where Scimitar must take special care to avoid potential pitfalls.

3.4.1 Exact Encodings

Figures 3.6, 3.7, and 3.8 give the encodings used in Scimitar for several key language constructs (described in Section 3.3): variable multiplication, conditionals, and dynamic indexing. Figure 3.9 presents the definitions of two libraries: Boolean algebra and variable comparison.

Variable Multiplication. Unfortunately, linear programs cannot multiply two variables, nor can MILP programs without special constraint formulas that create a relaxation of the expression. For this discussion, we break apart variable–variable multiplication into three cases: general continuous–continuous, binary, and integer.

The first is not possible in MILP, and requires a more powerful solver.

Scimitar does implement multiplication by a binary variable using the standard encoding known as McCormick envelopes [Dom18], which can encode multiplication by any set with two values. For the complete encoding see Figure 3.6. This encoding creates a convex relaxation that simulates multiplying such a variable with any other integer or continuous variable, which approximates the original nonlinear function. This method relies on knowing the upper and lower bounds (denoted by superscript u and l, respectively) of each variable. This simulated multiplication takes the product of these known constant coefficients with each variable individually, thereby

$$
\begin{array}{rcl}
[\![x \cdot y \in \{0,1\}]\!] & \geq & x^l \cdot y \\
[\![x \cdot y \in \{0,1\}]\!] & \geq & x^u \cdot y + x - x^u \\
[\![x \cdot y \in \{0,1\}]\!] & \leq & x^l \cdot y + x - x^l \\
[\![x \cdot y \in \{0,1\}]\!] & \leq & x^u \cdot y \\
[\![x \cdot y \in [0, 2^n - 1]]\!] & = & \sum_{i=0}^{n-1} [\![2^i x \cdot y_i]\!] \\
& \textbf{s.t.} & y = \sum_{i=0}^{n-1} 2^i y_i \\
& & \forall\, 0 \leq i < n \,.\, y_i \in \{0,1\}
\end{array}
$$

Figure 3.7: Encodings of integer multiplication

$$
\begin{array}{rcl}
[\![\text{if } c \text{ then } b_t \text{ else } b_f]\!] & = & c \cdot b_t + (1-c) \cdot b_f \\
[\![\text{if } c \text{ then } b_t \text{ else } b_f]\!] & = & \forall\, r_{t_i} \in \text{free}(b_t). \\
& & \quad v_{t_i} = \text{if } c \text{ then } r_{t_i} \text{ else } d_{t_i} \\
& & \forall\, r_{f_j} \in \text{free}(b_f). \\
& & \quad v_{f_j} = \text{if } c \text{ then } d_{f_j} \text{ else } r_{f_j} \\
& & b_t[r_{t_0} \mapsto v_{t_0}] \ldots \\
& & b_f[r_{f_0} \mapsto v_{f_0}] \ldots
\end{array}
$$

$$
\begin{array}{rl}
[\![\text{ref } (v_0 \ldots v_n)\ y]\!] = \sum_{i=0}^{n} b_i \cdot v_i & \text{s.t.} \quad y = \sum_{i=0}^{n} i \cdot b_i \\
1 = \sum_{i=0}^{n} b_i & \forall\, 0 \leq i < n \,.\, b_i \in \{0,1\}
\end{array}
$$

Figure 3.8: Encodings of various useful language features

avoiding multiplying the two variables directly. SCIMITAR carries these bounds in the variable's type, which makes implementing this translation simple (see Section 3.3.2). Experts may find introducing McCormick envelopes simple for small programs, but as the program grows it becomes increasingly difficult to track and maintain them.

Multiplication by integer variables extrapolates on binary multiplication. For the full encoding, see Figure 3.7. Without loss of generality, we assume that y is the integer or binary variable. In a given multiplication, the integer variable is decomposed by constraints that convert it into a bitvector. These bits are then used in a binary multiplication constraint, and then all of them are recomposed into the multiplication's output. The binary multiplication is a specialized, simplified version of the complete McCormick envelope.

Despite being a relaxation, the MILP solver algorithm guarantees that because one variable is assured to be binary, the solution is valid for the original multiplication up to the internal numerical limits of the solver.

If-Then-Else. Conditionals are implemented in two cases, shown at the top of Figure 3.8.

The *simple case* is used to encode conditional formulas that do not include any constraints. Here, SCIMITAR encodes the guard as an indicator variable c that is multiplied by the branches b_t and b_f, thus effectively "disconnecting" the other branch. By plugging true and false into c, we can see that the corresponding branch value is returned.

The *constraint case* is necessary when encoding conditionals where one or both branches contain constraints, and requires a more complex encoding. As with the simple case, to translate

$$
\begin{aligned}
\text{true} &= 1 \\
\text{false} &= 0 \\
\text{not}(b) &= 1 - b \\
\text{and}(b_1\ b_2) &= b_1 \cdot b_2 \\
\text{or}(b_1\ b_2) &= b_1 + b_2 - b_1 \cdot b_2 \\
\text{xor}(b_1\ b_2) &= b_1 + b_2 - 2 \cdot b_1 \cdot b_2
\end{aligned}
$$

$$
\begin{aligned}
&\text{cmp}(n\ m\ c_< \ c_= \ c_>) = \\
&\quad \{\ 1 = c_< + c_= + c_>, 0 = c_< \cdot (m - n - \epsilon), \\
&\qquad 0 = c_= \cdot (n - m), \quad 0 = c_> \cdot (n - m - \epsilon)\}
\end{aligned}
$$

Figure 3.9: Encodings of useful library functions

the constraints, we must disconnect each branch depending on the result of the guard. This is more complicated here because asserts must only activate when their corresponding branch is selected. We disconnect branches by replacing their variables with dummies. First, we collect the *real* free variables r_x from each branch, where x ranges over the free variables r_{t_i} and r_{f_i}. For each variable, we create a *dummy* variable d_x. We then select between these one by one using the encoding from the simple case, assigning each to the *used* variable v_x. Finally, the branches are encoded but substituting the real variables for the used ones.

This approach allows the solver to satisfy each branch's constraints with variable assignments that do not impact the rest of the problem. The dummies are unconstrained slack variables that "float" without impacting the computation. Thus the active branch uses the real variables, while constraints in the inactive branch have no effect.[2]

Dynamic Indexing. Dynamic vector and matrix indexing ref $(v_0 \ldots v_n)\ y$ is the process of selecting a specific vector element v_k using a variable y as the index. Dynamic indexing is required, for example, in a recursive function with an argument representing the index into a vector, as discussed in Section 3.4.2. Because the index is a variable, the compiler does not know in advance how to access the vector, and must leave it to solve time. Dynamic indexing is not a simple operation, and needs constraints proportional to the size of the vector to break it apart. In contrast, selecting an element using a constant is trivial, only needing a constant number of constraints to extract a known offset. The implementation of dynamic indexing is shown at the bottom of Figure 3.8.

First, the encoding needs one-hot indicator variables b_i for each index v_i in the vector. Only the indicator b_k is nonzero, so the sum of all $b_i \cdot v_i$ is equal to the corresponding vector entry v_k. The variable y is translated into the one-hot encoding by scaling each indicator by its corresponding index. For example, to select the third index, the third indicator variable b_3 is multiplied by three. This will then select the third vector element v_3 as the result. The encoding is constructed this way so that both constraints on v_i and y affect the value of the ref, and so that constraints on the ref affect the values of the v_i and y.

[2]With one exception: if the programmer includes unsatisfiable constraints such as $0 = 1$ in a branch that is not taken, the problem will be infeasible overall.

Boolean Algebra. The encoding of Boolean algebra is straightforward, as shown in the top set of equations in Figure 3.9. The SCIMITAR library implementation uses one for true and zero for false, and each Boolean operation can be modeled using simple sums and multiplications of their arguments. By plugging true and false into each formula, we can see that the corresponding truth table is satisfied.

Variable Comparison. Testing variables for equality or inequality is a necessary operation for any numeric computation system. SCIMITAR's library encodes it using one-hot indicators $c_<$, $c_=$, and $c_>$, each of which is true when one input is less than, equal to, or greater than the other, respectively. The encoding of this test relies on some parameter ϵ, as shown in the cmp equations at the bottom of Figure 3.9. Because they are mutually exclusive, if $c_=$ is true, it must be the case that $n - m$ is zero. Otherwise, this difference must be at least ϵ away in one direction or the other (as the smallest value in our system, we do not worry about differences smaller than ϵ).

3.4.2 Approximate Encodings

Figure 3.8 omits some important encoding techniques for language constructs. One thing to note is that unlike the exact encodings, in certain cases these techniques can lead to infeasible problems, and some do not achieve the optimal result. The user must pay special attention to their use in practice, as programmers often have sideband information that is not encoded directly into the program and that is impossible to determine programmatically. Avoiding these bad cases may require tweaking some runtime parameters, such as inlining and unrolling bounds, as we discuss next.

Inlining. An optimization-aided language must inline all function calls, both normal and recursive. As with traditional languages, function inlining directly replaces a function call with the function body. Unlike traditional languages, optimization problems are unable to reuse code. Each call to a function must be translated into a separate copy of that function's constraints to allow the solver to freely set its variables independently of the other calls to the same function.

Optimization problems are intrinsically bounded. Finitizing a recursive function is challenging because it can only be inlined up to some depth limit parameter supplied by the user at compile time, and is unable to be invoked beyond this depth. When the limit is reached, SCIMITAR compiles in a sentinel constraint with an associated path condition. Given a sufficient depth limit, this path condition will not be met at solve time because the solver will be able to reach the program's base case. The sentinel is triggered when the solver has no choice but to meet its path condition, which conditionally makes the program infeasible (as discussed above in Section 3.3.1). As a consequence of this, recursion without a base case will always yield infeasible results. Note that this design also makes it impossible for a user to distinguish whether their problem has no solution or if a solution might be found by increasing the inlining depth limit—a fundamental problem faced by all solver-aided systems with finite problem size.

Loop Unrolling. Scimitar finitizes loops by unrolling them. We discuss two cases of loops: known bounds and unknown bounds. Scimitar can completely unroll finite loops because the number of iterations is known at compile time. If it is impossible to calculate the range of iteration, such as when the upper bound of the type uses a solver variable, compilation falls back on a translation to a recursive function using an index parameter. This translation produces a sequence of index-guarded loop bodies up to a user-supplied unroll limit parameter. As with inlining, the drawback of this approach is that the problem will be infeasible when the solver is unable to find a solution within the number of unrolled iterations.

3.4.3 Other Considerations

Finally, to avoid explosion of the size of the compiled program, it is critical to optimize the whole-program representation. This includes using compact code and data representations, type information, compiler passes such as constant propagation, and constraint normalization.

Data and Problem Representation. It is critical to have an efficient representation for optimization problems and their data. High level languages are highly compositional, with most terms containing several subterms. During the translation to solver-level constraints, this can quickly lead to large optimization problems, as translated subproblems will constantly have to be linked up using conjoining constraints, introducing a constant number of extra constraints per subterm. Because concatenating constraints is a frequent operation, the speed is critical. Thousands of intermediate constraints may be constructed, so this must take as little time as possible.

Unavoidably, the compiler may have to inspect or modify the contents of a given solver constraint or value, e.g., when selecting an index, merging two constraints, splitting one constraint into multiple, or scaling a value. An efficient representation can have an asymptotic speedup for these operations. In Scimitar, we elected to represent optimization problems as normalized sparse matrices augmented with type metadata. Due to the compositionality discussed above, most constraint rows reference very few variables, and those variables are often only used in adjacent rows. Because the output problem generated by our compiler is very sparse and clustered around the matrix's diagonal, this is a very efficient representation.

Types. Scimitar's types are invaluable for several reasons (see Section 3.3.2). Firstly, we must track the type of data for the sake of the solver, whose internal representation requires variables to be bounded. As usual, it is important to verify whether operations are even permitted on certain variables. Without types, it is impossible to implement the encodings we have mentioned, e.g., McCormick envelopes and dynamic vector indexing, as they need to know the bounds and shapes of their arguments. The approximate encodings mentioned above would also be impossible without tracking variable types, as we need them to check the path condition of the sentinel. Finally, data flow and control flow across the host-solver boundary would not be possible without types, since the compiler is not designed to introspect and infer the types of runtime data. Even if the functional language values were themselves untyped, we must know the relationship between

the functional language representation and the optimization language representation in order for values to cross the boundary. We discuss types in detail in Appendix B.1.

Compiler Passes. The Scimitar compiler is comprised of several successive passes. Transformations such as constant propagation are critical to minimize the number of constraints. Because assembling the final standard form problem is the most expensive step of compilation, the fewer constraints that reach that phase, the faster the program will compile. By eliminating or combining redundant and coupled constraints, we can offset changes later in the pipeline where additional constraints must be generated, such as in the case of McCormick envelope expansion. There is a trade off, however, because the time spent performing preprocessing may actually exceed the time spent in the solver. We found that in practice, most passes executed in a matter of microseconds, or at worst, a few milliseconds. This is in contrast to assembling the standard form problem, which was the compiler performance driver, and represents the biggest opportunity for performance improvements; see Section 3.5.

Constraint Normalization. One particularly important requirement for the compiler is to reformulate the constraints into a normalized representation by successively rewriting terms into lower level representations and guaranteeing that some terms only appear in certain positions, like requiring the left subterm of multiplication to be a number. It is important to eliminate terms as early as possible, thereby requiring fewer redundant decisions. This is one reason we separate the Scimitar grammar from the optimization language grammar. There are even cases where one constraint must be split into two to create this streamlined representation. In addition to performance, a side benefit of normalization is that it helps in simplifying subsequent compiler passes.

3.4.4 Pitfalls

While the encodings discussed above present opportunities to support a diverse selection of high level language constructs, there are many challenging details that had to be carefully overcome or avoided to implement Scimitar. We briefly discuss two of these issues.

Types. As powerful and necessary as types are in Scimitar, they are not trivial. The compiler infers the types of variables and expressions, which is nuanced. The core feature of the type system is the shape and bounds of vectors as mentioned in Section 3.3.2, and type information is required to correctly lower many operations constraints.

Vectors of different bounds and shapes must often interact. Such interactions could be problematic in a system that required strict type equality, as many expressions would not type. Because vector types in Scimitar form a partially ordered set over their bounds and shapes we can accommodate such interaction by performing subtype inference.

In some corner cases, such as general variable multiplication or indexing a vector by the index of another vector, Scimitar cannot infer the type correctly, because the type system lacks the richness to cover these cases. Scimitar currently requires type annotation for those cases, and will produce a type error if they are missing.

Higher Order Functions. Compiling higher order functions in Scimitar requires careful handling of function arguments. There can be conflicts between supplying a function argument and actually applying that parameter in the body of the higher order function. These conflicts arise with functions that can not be fully inlined. In cases such as returning a function from another function, Scimitar is not able to decide how that resulting function can be invoked. As a result, Scimitar's implementation of first class functions is incomplete.

3.5 Evaluation

To demonstrate Scimitar's features and explore its capabilities, we developed several benchmarks:

- *logistics* is a traditional logistics example of optimizing for profit. Two versions of the problem are shown in Figures D.3 and D.4.
- *pipes* is a simple network flow problem demonstrating basic language features. This and *logistics* show Scimitar's applicability to classic optimization problems. See Figure 3.10.
- *malloc* is a full implementation of the program sketch described in Figures 3.2 and 3.3. We give the core optimization problem for the evaluated version in Figure D.2.
- *recitation* minimizes the number of sections required to adequately serve the students from a class. See Figure D.5 for the core of the benchmark.
- *contradict* tests a conditional that contains a contradiction.
- *bounce* recurses on a simple heuristic that bounces back and forth until converging. Our implementation is shown in Figure D.1.
- *sum-to-n* is the problem presented in Figure 3.1.
- *imp* is a complete implementation of Winskell's Imp language, which demonstrates Scimitar's full modeling power. We give an example of Imp program in Figure 3.11.

We discuss the design of each benchmark below and give its size in terms of source lines of code (*sloc*), including the extra Racket support code required to decide program properties at compile time such as input data formatting. For each benchmark, we also report Scimitar's performance in terms of compile and solve times. For several benchmarks, we wrote a corresponding problem directly in the optimization language $\mathcal{O}$. In these cases, we compare the performance of the Scimitar-generated problem encoding to the directly written version. Most programs compile faster in $\mathcal{O}$, but the difference in performance is unpredictable. For several problems, we also compare performance against Rosette versions of the benchmarks. We use Rosette's `optimize` query, which depends on Z3's MaxRes algorithm. The performance differences are split between time spent encoding programs and time needed by the underlying solver. Overall, Scimitar's compile times are two orders of magnitude slower than Rosette, but its solve times are an order of magnitude faster.

Our benchmark code is available along with the Scimitar implementation [BFZ24b].

3.5.1 Benchmarks

Logistics One of the classic domains addressed by optimization solvers is logistics problems. We implemented an example logistics problem that optimizes and returns the maximum profit expected from some commercial warehouse and trucking enterprise.

Originally, we implemented the benchmark in the optimization language—at the time, Scimitar source language was not fully developed. We then continued to co-develop the source language and the example. This back-and-forth drove the design of Scimitar, and caused us to add support for dynamic loop unrolling, dynamic recursion inlining, and dynamic vector indexing. Scimitar also influenced the implementation of the benchmark. Because solve-time conditionals are central to Scimitar, as we ported to the functional language we replaced operations that were implemented using MILP encoding techniques with structured code using for and if. Doing so led to simpler and more intelligible code.

Overall, we found that Scimitar source language code is much easier to update and modify. Although the raw number of constraints generated by our compiler for a given problem is a constant multiple greater than the number in the manual version, the Scimitar version is more maintainable because as the input program grows linearly in size, the number of constraints in the compiler output (and the manual version) grows quadratically. Because of the design of this benchmark, most constraints in both the Scimitar and $\mathcal{O}$ versions are highly redundant, and some are similar but have nuanced differences. This can make it difficult to maintain the $\mathcal{O}$ version if the data changes. Furthermore, the Scimitar version can be much more easily changed, e.g., to also optimize the number of trucks, while in a hand-written version doing so would require drastic changes.

As a demonstration of the ease with which we can change the higher-level benchmark, we measured two versions of the problem, one smaller and one larger. The *logistics-s* program has 1 product, 2 cities, 1 road, and 1 truck, while *logistics-h* has 4 products, 4 cities, 4 roads, and 8 trucks. As stated previously, this increase in program size results in a disproportionately larger number of rows and variables in the output, and a greatly increased solve time, as shown below in Table 3.1.

Our implementation of *logistics* is 145 Scimitar sloc, with 77 sloc of Racket support code.

Pipes Figure 3.10 shows an excerpt from the Scimitar code for *pipes*, which calculates the maximum flow through a pipe network, a classic optimization problem. For the flow to be valid, the inflow and outflow of every junction must balance (lines 6-11) given each pipe's capacity (lines 12-17). Additional constraints matching source and sink flows have been omitted. Our implementation resembles the usual formal problem statement, but is written such that the programmer need not directly encode each constraint and couple it to the data. A modest amount of extra support code is needed to preprocess the data from a graphlike representation into the format that the algorithm expects. After the Scimitar program is compiled to the optimization language, the constraints generated are similar to a hand-written version of the problem.

This basic problem demonstrates Scimitar's capabilities on a convex problem with no integer variables. The Scimitar code for this traditional optimization problem is simple to implement, easy to understand, and straightforward to verify.

```
(optimum-ref sink-out
 (maximize sink-out
  (define sink-out (symbolic))
  (define pipes (symbolic))
  ; omitted source and sink constraints
  (for ([j (range (length num-is))])
   (assert ; junction inflows = outflows
    (= (sum ([i (range (vec-ref num-is j))])
        (vec-ref pipes (vec-ref j-is `(,i ,j))))
       (sum ([i (range (vec-ref num-os j))])
        (vec-ref pipes (vec-ref j-os `(,i ,j)))))))
  (for ([i (range (length pipes))])
   (begin ; pipe flows in allowed range
    (assert (<= (- (vec-ref pipes i))
                (vec-ref pipe-flows i)))
    (assert (<= (vec-ref pipes i)
                (vec-ref pipe-flows i)))))))
```

Figure 3.10: The *pipes* problem

Pipes also serves as a good benchmark to compare with Rosette since the two implementations are virtually identical.

The *pipes* benchmark is 21 Scimitar sloc, with 113 supporting Racket sloc.

Malloc The *malloc* benchmark shows a more real-world utility, something a developer might want to write as a part of their program. This benchmark is a version of the memory allocation example presented in Section 3.2.2. The original example serves as framework that an author of a memory management system could implement using their own criteria. The snippet shown in Figure 3.3 is incomplete, and serves to explain the features of Scimitar. The benchmark implementation we measured uses a different heuristic that is more complex and fleshed out than the one in Figure 3.3. *Malloc* also demonstrates interaction across the host–solver boundary, where variable optimums in one iteration become parameters in the next iteration. The *malloc* benchmark is 29 Scimitar sloc, with 71 supporting Racket sloc.

Contradict, Bounce, Sum-to-n These benchmarks are toy examples that we used to demonstrate the basics of Scimitar. The *contradict* benchmark attempts to minimize the expression `(if x (assert (= x 0)) (assert (<= x 1)))`. The true branch introduces a contradiction, and Scimitar correctly determines x to be false. The *bounce* example recurses on the output of a toy heuristic, passing the previous result into the next round until the values converge. We give the code for *sum-to-n* in Figure 3.1.

Recitation This benchmark minimizes `rec-count`, the number of sections required to adequately serve the students in a class (thereby reducing the number of TAs, rooms, etc.). The problem guarantees that all registered students are assigned a section, and that recitations are only scheduled if they meet a minimum registration count. Recitations are modeled using symbolic variables for the

```
(:= y 0)
(:= x 0)
(while (<= y 5)
  (:= x (+ x y))
  (:= y (+ y 1)))
```

Figure 3.11: Example of an Imp program

possible section slots that could be scheduled and their attendance. To solve this, the program uses `rec-count` as the upper loop bound (as discussed in more detail in Section 3.4.2). This construction simplifies the representation, because the relationships between the students and sections can be enumerated without worrying about the number of sections needed or precisely which ones are selected. The *recitation* benchmark is 43 sloc for both Scimitar and supporting code, respectively.

Imp Winskell's Imp language is a minimalistic imperative language supporting conditionals, loops, and basic operations on numbers and booleans. For example, Figure 3.11 shows a simple iterative summation program in Imp, akin to Figure 3.1. The code initializes two variables `x` and `y`, then loops incrementing `y` and adds that to `x` until `y` is greater than 5.

We developed a compiler that translates an Imp source program to Scimitar source. We chose this example to illustrate the versatility of Scimitar to handle more complex iterated domains. Note that Imp is unlike other benchmarks, which are written directly as Scimitar source programs. The compiler works by splitting Imp source into *basic blocks*, sequences of instructions without jumps. Each basic block is compiled to a continuation that contains a **`minimize`** expression. As the compiled program executes, control flow from one basic block to another corresponds to one continuation calling another in the virtual machine.

Mutable state is represented as an environment that gets passed from one continuation to the next. For example, for Figure 3.11, the environment is a pair containing `x` and `y`.

Note that the compiler deliberately does not unroll loops in Imp. In theory, loops like the one shown in Figure 3.11 could be unrolled. However, we wanted the compiler to support full Imp semantics, including non-termination of loops such as (`while true skip`). Generally, functionality which can not be shown to terminate or is unbounded in size can not be finitized by Scimitar into an optimization problem; doing so requires stepping back into the host language.

The Imp compiler is 475 sloc. Unlike other benchmarks, the implementation of the compiler intermixes Scimitar and Racket code together, with no simple breakdown between the two.

3.5.2 Solver

Our evaluation used the popular off-the-shelf Gurobi MILP solver [Gur23]. We chose Gurobi because of its versatile and rich API and excellent performance. The sparsely encoded matrix and vector representation our compiler uses is easily translated to Gurobi's expected format, and from there it is loaded using the solver's FFI. Scimitar performs this translation immediately before the solving step in the virtual machine. We attribute the time spent in this translation and loading step to compile time.

Table 3.1: Measurements in seconds for Scimitar, optimization language, and Rosette programs

			Scimitar		$\mathcal{O}$		Rosette	
	vars	**rows**	**compile**	**solve**	**compile**	**solve**	**compile**	**solve**
pipes	161	179	34.9 ms	0.60 ms	0.60 ms	18.8 ms	0.45 ms	0.0 ms
logistics-s	268	293	53.5 ms	2.0 ms	1.4 ms	21.3 ms	2.2 ms	30.8 ms
logistics-h	10884	14026	478.7 ms	1032.9 ms	674.3 ms	112.6 ms	66.6 ms	>20 m[2]
contradict	158	229	33.3 ms	1.0 ms	0.52 ms	2.3 ms	0.08 ms	25.2 ms
sum-to-n	11817	17990	1417.2 ms	54.7 ms	—[1]	—[1]	>20 m[2]	—
bounce	1858	2538	130.5 ms	12.9 ms	—[1]	—[1]	1.1 ms	51.1 ms
recitation	34846	52514	2190.1 ms	615.2 ms	—[1]	—[1]	49.7 ms	125490.0 ms
malloc	227	239	128.6 ms	4.2 ms	—[1]	—[1]	8.7 ms	189.4 ms
imp-s	8171	10158	201.4 ms	16.5 ms	—[1]	—[1]	—[1]	—[1]
imp-h	11068	13074	689.6 ms	16.6 ms	—[1]	—[1]	—[1]	—[1]

[1] These benchmarks do not have a corresponding non-Scimitar version.
[2] This benchmark exceeds a 20 minute time out.

3.5.3 Results

Table 3.1 gives the median run time performance in milliseconds of each benchmark program and the variable and row count for the compiled code. All measurements were taken on a 3.2 GHz AMD Ryzen 5 1600 system with 32 GB of RAM using Racket's `current-inexact-milliseconds` function.

Note that the comparisons of Scimitar to Rosette are, to a certain extent, comparing the efficiency of Gurobi and Z3. Z3 implements primal simplex and MaxRes [Bjo22], while Gurobi uses branch-and-bound [Gur23] (with relaxation to simplex), which we expect to be faster. However, the benefit of using a dedicated optimization solver is also part of the benefit of using Scimitar.

The comparison of Scimitar to Rosette also measures the efficiency of our respective encodings. In both cases, this time is split between compile and solve time, though in different ways. Scimitar's encoding and high-level optimization are mostly performed at compile time, with lower-level optimization left to Gurobi. In contrast, in Rosette, the encoding happens at compile time, and the optimization happens at solve time.

Table 3.1 compares our benchmarks written in Scimitar, in the optimization language $\mathcal{O}$, and in Rosette. Generally, the performance bottleneck in Scimitar is the compiler, which compares unfavorably with the others. Conversely, the solve time of Scimitar is dramatically better than in Rosette. Surprisingly, the solve time of the optimization language version exceeds Scimitar in some cases, which implies that the compiled version is more efficient than one that would be written directly. This is because solver performance is unpredictable, and small changes sometimes greatly affect running time.

Examining the results in more detail, we make several additional observations. The performance of *logistics-h* as compared to *logistics-s* degrades non-linearly. The increase in size by a factor of over $40\times$ is because there is simply much more data to account for. The benchmark's complexity scales with the scaled sum of products of these parameters. As a result, the problem has 14026 constraints and 10884 variables, many of which are binary. Unfortunately, while it is

often possible to relax binary variables to continuous ones, in this case it is not possible because all of them are indicator variables. In this benchmark, performance is wildly different across the three versions. Both Scimitar and the optimization language version return correct results, but the latter version is over nine times faster. The Rosette version is virtually the same as the Scimitar version, but it exceeds a 20 minute time out, so we are unable to evaluate it. We do not know if this timeout is an issue with Rosette or with Z3.

While Scimitar executes *sum-to-n* quickly, Rosette hits a 20 minute timeout while compiling. As such, there is no solve time measurement for this benchmark in Rosette.

The performance of *recitation* is over two orders of magnitude faster in Scimitar than in Rosette. We believe this is because the benchmark optimizes over a loop bound, which we encode efficiently.

While executing, *malloc* makes multiple solver calls, and the times presented reflect the sum total across all runs. Because of the nature of this design, we think this particular benchmark's performance would benefit disproportionately from compiler performance improvements.

The two *imp* benchmarks also make multiple solver calls, but unlike *malloc*, which only has a single problem that is solved repeatedly, *imp* changes between different problems dynamically. The sizes given are the sum across all generated problems. *Imp's* compiler performance reflects the high number of individual problems that a given *imp* program uses. Both *imp-s* and *imp-h* are similar. In fact, *imp-s* is basically a subset of *imp-h*. The size difference is linked to overall program size. Both benchmarks are large overall compared to our other benchmarks due to the difficulty of encoding an imperative programming language as a constraint problem.

It was not possible to replicate all Scimitar benchmarks using the optimization language. The use of some key Scimitar features would require reproducing large chunks of the output of the compiler. Additionally, Scimitar programs may solve problems successively, which the optimization language is incapable of. For the *imp* compiler, while it would be possible to write a similar optimization language program for a given Imp program that does not require these features, it would represent an enormous engineering effort, on the order of writing the original compiler. Likewise, we did not attempt to replicate *imp* using Rosette as such an implementation would require a significant engineering effort.

The performance of these benchmarks is clearly correlated with the number of variables and constraints.[3] One nice feature of our encoding is that, while these numbers seem to suffer from a blowup with increased number of variables v and rows r, growing at the rate of $O(v \cdot r)$, the encodings that Scimitar produces are normally very sparse, leading to an increase that is closer to $O(v + r)$.

In summary, while Scimitar's compiler is not the fastest overall, the improved solver speed gives evidence that a dedicated MILP solver compares favorably with the more general purpose SMT solver on optimization applications.

[3]The exception being *malloc*, whose performance is skewed because it is compiled multiple times within a single execution.

3.5.4 Analysis

Beyond raw solver performance, there are other differences that may influence the choice of one language over another.

While Rosette can make use of Z3's optimization capabilities, it remains a satisfaction-oriented language. For problems where that is sufficient, Rosette would be preferable, and will almost certainly outperform Scimitar. However, as we see here optimization is not its focus, and for such problems Scimitar will be more performant.

Another reason to use Rosette is that its implementation incorporates a rich subset of Racket's features, while the current implementation of Scimitar is less complete.

Scimitar and Rosette naturally model data differently in accordance with the underlying solvers, with Scimitar being built on top of primitive MILP concepts, while Rosette's data is restricted to types supported by SMT. Accordingly, data in Scimitar is built on top of procedures and vectors over scalar sets, while Rosette's core data types are booleans, integers, reals, bitvectors and uninterpreted functions. While both languages track the types of variables, Scimitar infers the types of variables and expressions, while Rosette requires type ascription and does not support typing of expressions.

Finally, the applications are quite distinct, and despite considerable overlap, they target different classes of problems. Those given by Rosette's developers include formal verification of systems such as JIT compilers, synthesis of GPU kernels, program repair, and model checking. On the other hand, Scimitar targets applications such as logistics problems, resource allocation, and design optimization.

3.6 Related Work

MILP Encoding Popular encoding techniques are diverse [Bra12, Dom18, HV22, GAM22, HV19, RJG+12, Wik24, Par88] and implemented on a case-by-case basis in various MILP frontends. While Scimitar does not implement every technique listed, it aims to provide the programmer with an environment where they do not have to think about the details of encodings.

Embedded Optimization Problems Systems such as cvxpy [DB16], JuMP [DHL17] and Matlab [The20] embed mathematical programming within a general purpose language. These systems are focused on dynamically constructing an optimization problem piece by piece, not on modeling a general purpose language. In contrast, Scimitar is not an embedded language. Instead, it reasons directly about programs themselves, rather than constructing a problem out of pieces.

The classic constraint language is AMPL [FGK87]. Although featureful, it does not attempt to model high-level language features.

Unique Computing Paradigms Rosette [TB13] explored embedding SMT solvers in a functional programming language, and was an inspiration for Scimitar. Both are frameworks that use symbolic expressions translated to constraints then submitted to a solver via an angelic execution query. Both enable developers to write and reason symbolically about programs using high-level

abstractions in the traditional functional style, including constraints over symbolic expressions. The modeling language matches the host language, so there is less cognitive load when transitioning across the host-solver boundary. While inspired by similar principles, Rosette is focused on verification and synthesis, while Scimitar is meant for optimization.

Differentiable programming [AP20, Man12, BRNR17] is an approach to numeric programming that uses automatic differentiation [BPRS18, HP13, PS08] to generate the derivative of a program for use in algorithms such as gradient descent. This is a parallel to Scimitar's approach—it directly applies an alternative interpretation of a program, expanding that program's semantics to include the result. To do this, it must syntactically analyze the input program to generate the derivative. Similar to Scimitar, this technique requires unique handling of conditionals, which are normally discontinuous.

We designed Scimitar as a standalone language because of the unique demands of the runtime system. The HANSEI [KS09] language is an example of using a similar approach to embed its special computing paradigm, probabilistic programming, while making use of host language features. As opposed to cvxpy and JuMP, the embedding is much more shallow, and HANSEI programs hardly look different than programs in a standalone probabilistic language. Scimitar cannot be similarly embedded because it must reason about global properties like control flow and variable use that require an ability to reason about the implementation of every function.

Embedded Functional Program Some recent efforts have been made to encode high-level language features using constraints. These languages compile programs in an imperative syntax to constraints, which allows for the possibility of deciding satisfaction or performing optimization, without requiring programmers to work directly in the constraint language.

BFDL [Bra12] is a high level language derived from Fairplay [MNPS04]. It offers encodings of several language features into a constraint system including non-recursive function calls, fixed iteration loops, conditionals, boolean algebra, integer arithmetic, comparison, user-defined structs, and constant index array access. This is vaguely similar to Scimitar's optimization language, except that BFDL is focused on constraint satisfaction, whereas Scimitar is meant for optimization.

CirC [OBW20] is a compiler framework for proof systems that uses ILP as one of its backends, and can use that to discover output-maximizing inputs for a constraint set. It supports a wide array of language features, as demonstrated by several frontends. The key distinctions are that Scimitar optimizes over any program variable including ones that are used across nested functions, and that solver invocations are nested within normal functional programs, which allows the virtual machine to direct control flow according to optimization results.

High Level Constructs within Solvers MiniZinc [NSB+07] is a high-level, typed, mostly first-order, functional constraint optimization modeling language. It offers facilities for abstraction such as let-bindings and defining predicates and functions, but unlike Scimitar it cannot manipulate these as first class. MiniZinc also has a stronger compile-time/run-time distinction compared to Scimitar. MiniZinc compiles to the simpler FlatZinc format before being handed off to solvers. Scimitar iteratively unfolds a problem to handle an a priori unbounded number of variables,

whereas MiniZinc is more suited to problems with a well specified fixed domain.

3.7 Summary

Optimization-aided programming integrates numerical optimization solving into programs in a natural style familiar to programmers. It abstracts away the requirements, restrictions, and limitations inherent to underlying optimization algorithms and tools, so users can focus on the domain of their own problem. This goal is achieved by automatically encoding many language features users would expect such as solve-time conditionals, bounded inlining, and loop unrolling, which the author of an optimization problem would normally have to write by hand. Merely by adding constraints and cost functions to their program, programmers can leverage these powerful tools and encodings to find optimal values for their computations without worrying about how these values are derived. Our benchmarks demonstrate the benefit of using a dedicated optimization solver in domains such as logistics and resource allocation as compared to a more general SMT solver.

Scimitar as presented in this Chapter is however limited to the minimize query, and optimizing over locally declared symbolic variables. In the next Chapter, we extend Scimitar with additional queries, making it into a complete program synthesis system.

Chapter 4

Optimization Aided Counterexample-guided Inductive Synthesis

4.1 Introduction

While many program synthesis applications would find any solution sufficient that satisfies its specification, there are times when a user prefers one solution to another. To fulfill such a desire, the synthesis system must generate programs satisfying additional properties difficult to state with assertions alone, such as minimizing or maximizing some quantity. Examples include fewest disk writes, best worst-case packing for a buffer, least energy used, lines of code generated, etc. Such heuristics, while feasible, are difficult for users to encode via existing approaches, and would benefit from a paradigm explicitly supporting such an operation.

Optimization-aided programming is such a paradigm. It integrates mixed integer linear programming (*MILP*) constraint optimization problems into functional programs, which enables programming using user-specified objective functions that encode application goals.

In Chapter 2, we discussed sketch-based program synthesis using counterexample-guided inductive synthesis (CEGIS) [STB+06], then Chapter 3 introduced optimization aided programming using the Scimitar language. In this chapter, we combine these two techniques into a new *optimization-aided synthesis* system, Synthitar. This system extends Scimitar with holes, universally quantified inputs, and an optimizing synthesis query, thereby allowing users to directly express these more complex properties. To integrate them, we introduce the *Optimization Aided Counterexample-guided Inductive Synthesis (OACIS)* algorithm that generalizes and extends the CEGIS algorithm to use an optimization solver to handle objective functions. Section 4.2 gives two examples that illustrate Synthitar and walk through a high level trace through OACIS.

To create Synthitar, we extend the Scimitar language with synthesis constructs including holes (`??`), a universal quantifier expression (`(forall (...) ...)`), and a synthesis query `minsynth`, and extend the virtual machine with an OACIS loop that makes use of these constructs. Section 4.3 gives a grammar showing the precise changes from Scimitar.

A significant design decision in Synthitar is how to define optimality in the context of a synthesis problem. In Scimitar, objective functions are solved for their minimum or maximum values, which is a unique global optimum. In Synthitar, however, there is a local optimum for every combination of holes and inputs. There are different possible interpretations to what it means to have an optimal value. Section 4.4 discusses four such strategies, each with its own benefits and drawbacks. It compares them by example, and justifies Synthitar's chosen strategy, which selects the optimum that exceeds all others on at least one input.

OACIS requires three major extensions to CEGIS to be suitable for optimization problems. We adapt the CEGIS loop into a more abstract version we call *General CEGIS* to allow those modifications. Recall that traditional CEGIS iteratively generates programs by determining values for holes using two SMT solvers. One solver, called the *synthesizer*, finds candidate hole assignments that meet a user-supplied specification. The second solver, called the *verifier*, attempts to find counterexample program inputs to these solutions that violate the specification. To create OACIS, first we replace the synthesizer's SMT solver with a MILP solver. Second, we introduce a new algorithm to construct MILP complement problems. Third, we modify the verifier to handle the internal (or *local*) symbolic variables our encoding requires.

Searching for candidate hole assignments is similar enough to CEGIS that we only have to replace the solver, but verification is not so simple. The need for local variables to capture dependencies across constraints requires a strictly more powerful solver than a MILP solver alone. Furthermore, we cannot simply negate an optimization problem and solve to find counterexamples as CEGIS does, because problems are encoded as a flat conjunction of constraints—MILP lacks the built-in disjunction and negation available in an SMT encoding. Our complement problem's construction makes use of encoding techniques including the use of a Δ variable to model strict inequality, giving rise to our *Maximum Variable Delta (Max-Δ) Verification* approach. We leverage and extend concepts pioneered in the framework of delta-decision procedures [KSLG18], a relaxation which allow one-sided numerical errors. Max-Δ finds worst-case counterexamples by forcing the solver to violate some constraint from the original problem by the greatest margin Δ possible. Section 4.5 describes the algorithm's motivation in detail and gives the implementation for General CEGIS. We use it to recover traditional CEGIS (to show it is at least as powerful), then to construct the rest of OACIS. Afterward, we then give the implementation of Max-Δ.

To evaluate Synthitar, we develop a suite of benchmarks that utilize various aspects of the language to perform basic tasks. These range from simple mathematical expressions up to an implementation of an algorithm to calculate square roots. We measure time spent in each phase of OACIS, which we use to motivate a discussion about its performance drivers and the convergence properties of Max-Δ. Section 4.6 presents our observations about OACIS's characteristics and behavior on these benchmarks. We find that performance is linked primarily to the size of the input domain and the structure of the complement problem.

```
(minsynth (forall (y) (test-sqrt (: y nneg)))
(define max-iterations (??))

(define (test-sqrt y)
  (let ((n (babylonian y)))
    (assert (<= (abs (- (* n n) y)) error-tolerance)))
  max-iterations)

(define (babylonian y)
  (letrec
    ((go (lambda (x_n gas)
           (if (< (abs (- (* x_n x_n) y)) error-tolerance)
             (begin (assert (= 0 gas)) x_n)
             (go (* (??) (+ x_n (/ y x_n))) (- gas 1))))))
    (let ((gas0 (symbolic)))
      (begin
        (assert (<= gas0 max-iterations))
        (go (??) gas0))))))
```

Figure 4.1: Sketch for Babylonian square root method

4.2 Overview

To motivate Synthitar, we showcase these new additions using the example of synthesizing an algorithm to calculate square roots. After that, to further illustrate how OACIS works, we trace through its execution on an example synthesizing a loop guard.

4.2.1 Babylonian Method

The *Babylonian method* [Wik25] (also called Heron's method) is the first known algorithm for approximating square roots. The algorithm starts with some initial estimate of the root, and continues until the result is within some tolerance of the true root. It iteratively approaches the true root of a non-negative real number y by averaging the previous estimated root x_n and the quotient of y and x_n:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{y}{x_n}\right)$$

But it is reasonable to ask—is $\frac{1}{2}$ the best value for the averaging factor, or would scaling by a different value improve the algorithm? And how do we choose the estimated initial value? We can answer both of these questions by posing a synthesis problem: across all possible inputs, what assignment to these two values leads to convergence in the fewest overall iterations?

Synthitar programs are written in an extension of the Scimitar language. It adds a new synthesis construct, `minsynth`, a universal quantifier expression (`forall (...) ...`), and holes (`??`). Recall that we also rely on symbolic variables to create local existential variables, which can be assigned freely by the solver to satisfy the program for a given assignment of the holes and inputs. For a discussion of the grammar, see Section 4.3.

Figure 4.1 shows a Synthitar implementation of the Babylonian method. The synthesis

problem has three holes: the maximum number of iterations to converge, the averaging factor, and the initial estimate. The objective is to minimize the number of loop iterations needed to converge, inducing the synthesizer to choose values for the initial estimate and averaging factor that lead to the quickest convergence.

The example is contained in a `minsynth` expression on line 1. To specify our objective, we use the function `test-sqrt`, which we minimize for all inputs `y`. The type of `y` is ascribed to be `nneg`, the set of non-negative numbers (up to a user-defined upper bound). On line 2 we define `max-iterations`, the number of iterations required to converge, and assign to it a new hole. The hole syntax (`??`) wraps the double question mark syntax used by Sketch in parentheses to emphasize that it dynamically constructs a new hole.

The function `test-sqrt` encodes the specification for square root by calling `(babylonian y)` on line 5 and then ensuring on line 6 that the square of the result is within `error-tolerance` of the input. It then returns `max-iterations`, thereby causing it to be minimized by the synthesizer.

Inside our implementation on line 9, we have an inner function `go` that takes the current estimate `x_n` and the variable `gas` that represents the number of iterations remaining. Depending on whether the square of the estimate is within `error-tolerance` of the input `y` (line 12), either it returns successfully with zero `gas` (line 13) or iterates, averaging the estimate according to some unknown factor, and decrementing `gas` (line 14). The initial call to `go` on line 18 takes the initial unknown estimate and the symbolic variable `gas0` which line 17 constrains to be at most `max-iterations`. This variable is needed because `max-iterations` is a high watermark, and there are some values of `y` that need fewer iterations—directly using `max-iterations` here causes the benchmark to fail to synthesize.

To combine synthesis and optimization into a single algorithm, we must reconcile two conflicting paradigms, which requires answering two complementary questions. First, what is the meaning of "optimal" in the context of a synthesis system? Second, how can algorithms designed for logical specifications be adapted to handle numeric specifications?

Traditional MILP optimization problem have only one optimum, because all variables are existential. A solver's goal is to find *whatever* variable assignment optimizes the objective. By adding universally quantified inputs, optimization-aided synthesis runs into the issue that there is rarely one single optimum for the entire input domain. The mechanism for selecting which optimum to return is a crucial design decision for the synthesis system. In this example, the optimum we want is the smallest number of `max-iterations` such that the roots of all values of `y` are calculated with satisfactory precision—in other words, the minimum worst case. That lines up perfectly with Synthitar's definition of optimality, which we call *Best of the Best* (BB). For a more detailed discussion of definitions of optimality and why Synthitar chose BB, see Section 4.4.

Because Synthitar's encoding is a conjunction of linear inequality constraints, it lacks native disjunction and negation. The other key difference is the necessity for local existential variables. To address these, OACIS uses a novel algorithm to find counterexamples, which we call Maximum Variable Delta (Max-Δ) Verification. In general, a counterexample to an optimization-aided synthesis problem is *any* input assignment that causes the original problem to be infeasible—a point in the problem's complement space. In contrast, the Max-Δ approach locates the *worst*

```
1 (minsynth
2   (forall ((: x (interval 0.0 1.0)))
3     (let ((h (??)) (g (symbolic)))
4       (assert (= g (+ (* 2.0 h) x)))
5       (if (> g 1.0)
6         (assert h)
7         (assert (<= 0.5 x)))
8       h)))
```

Figure 4.2: Sketch for a loop guard

possible candidate counterexample given some candidate hole assignment. In this example, that means it should theoretically suggest counterexamples for `y` whose square roots are most out of tolerance or that require the greatest additional iterations. After identifying such a candidate, it is checked to ensure that there is no local variable assignment that could rule out this candidate. If it can be ruled out, MAX-Δ will produce the next worst candidate, and so on, until either a true counterexample is selected or no more candidates can be identified. We explain precisely what that means, the rationale, and give an algorithm for constructing that space in Section 4.5.

Running OACIS on this problem will yield an optimal hole assignment whose exact values will depend on the user's desired precision and the extent of the input domain `nneg`, but we should expect the desired precision set by the user to correlate quadratically to the number of iterations, and that the averaging factor will be a value very close to 0.5 as in the original algorithm. The tighter the precision, the more error will appear when taking square roots of small values, so the assignment to the initial value will be correspondingly smaller to compensate. In Section 4.6, we run the benchmark with the relatively tight range `nneg`=$[0, 5]$ and high `error-tolerance` of $\frac{1}{9}$, which results in an optimal hole assignment of `max-iterations`=3, an averaging factor of 0.49–0.5, and 1.73–2.26 as the initial value, which roughly corresponds with our intuition.

4.2.2 Example OACIS Trace

To illustrate how OACIS behaves, we will now trace through the synthesis of the simple contrived example given in Figure 4.2.

This example takes a single input `x`, whose type is ascribed to be the range $[0.0, 1.0]$. On line 3 it declares a hole `h` and a local variable `g`. Then, line 4 constrains `g` to be a linear combination of `h` and `x`. While that constraint makes the inclusion of the intermediate variable `g` redundant in principle, in practice the SYNTHITAR compiler actually creates an analogous fresh local variable anyhow, and so we make it explicit for the sake of demonstrating how OACIS uses it. Next, depending on the value of `g`, we either require `h` to equal 1, or constrain `x` to have 0.5 as its lower bound. Because `h` is used directly in the **`assert`** on line 6, we can infer it must be a Boolean, represented by the set {0,1}. From the types of `x` and `h`, we can further infer `g`'s type to be the range $[0.0, 3.0]$. Finally, on line 8 the example uses `h` as its objective to minimize.

To synthesize, SYNTHITAR encodes the entire program as a single optimization problem, which it submits to OACIS. As in traditional CEGIS, OACIS starts by fixing some arbitrary value for the input `x` and then solves for the holes. With a concrete hole assignment candidate in hand, it

then must verify it by looking for a counterexample input. If it cannot find one, the hole assignment solves the problem, otherwise the counterexample is used in the next round of synthesis.

As stated above, the approach to finding counterexamples is one of the key differences between CEGIS and OACIS. To understand Max-Δ's behavior in practice, we now walk through synthesizing the problem in Figure 4.2. For simplicity, we can set aside the context and trim the problem statement to the two core expressions:

```
(assert (= g (+ (* 2.0 h) x)))
(if (> g 1.0) (assert h) (assert (<= 0.5 x)))
```

Note that for the sake of readibility, we preserve the high-level constructs `if` and `>`, which are compiled away in the encoding the solver actually sees.

OACIS begins solving for `h` by randomly assigning `x`. For the purpose of this explanation, we will choose `x`=0.5:

```
(assert (= g (+ (* 2.0 h) 0.5)))
(if (> g 1.0) (assert h) (assert (<= 0.5 0.5)))
```

This program is satisfiable for both `h`=0 and `h`=1. Because the objective is to minimize `h`, the synthesizer will choose `h`=0, which forces `g`=0.5. Plugging these into the original problem gives:

```
(assert (= 0.5 (+ (* 2.0 0) x)))
(if (> 0.5 1.0) (assert 0) (assert (<= 0.5 x)))
```

Now OACIS begins searching for counterexamples using those assignments. In Max-Δ, we are most interested in the value *furthest away* from the set of feasible solutions to the original problem. In this case specifically, the first constraint is violated by any value of `x` other than 0.5, while the second is only violated when `x` is anything less than 0.5. As the range of `x` is $[0.0, 1.0]$, and 0.5 is the midpoint of that range, the assignments that most violate the constraints are the two extremes, `x`=0.0 and `x`=1.0. We will choose `x`=1.0 as the counterexample candidate. However, unlike traditional CEGIS, OACIS is not yet able to determine if this is a true counterexample. To "double check" the counterexample, we plug the value back into the original problem along with the hole, while leaving the local variable `g`, and submit this problem to the solver:

```
(assert (= g (+ (* 2.0 0) 1.0)))
(if (> g 1.0) (assert 0) (assert (<= 0.5 1.0)))
```

In this case, we discover that when `g`=1.0, there is a valid path through the code, so `x`=1.0 is a false positive and cannot be a counterexample after all. With this knowledge, we can rerun Max-Δ which chooses `x`=0.0 as the counterexample candidate. Double checking this assignment reveals that as both branches of the conditional lead to unsatisfiable asserts, there is no value of `g` that works for `x`=0.0, which is therefore a true counterexample.

In the next step of synthesis, we extend our problem with a new copy, substituting the value of the new counterexample. This new copy is akin to a different possible path through the program, and accordingly, we have to be sure to give `g` a fresh name in the new copy, or the two versions will be in conflict:

```
(assert (= g (+ (* 2.0 h) 0.5)))
(if (> g 1.0) (assert h) (assert (<= 0.5 0.5)))
(assert (= g_1 (+ (* 2.0 h) 0.0)))
(if (> g_1 1.0) (assert h) (assert (<= 0.5 0.0)))
```

$$
\begin{array}{lll}
e ::= & x \mid v \mid ?? \mid \text{symbolic} \mid \text{letrec } f \Leftarrow e \text{ in } e & \mathcal{C} ::= \; e \preceq e \\
\mid & \text{if } e \text{ then } e \text{ else } e \mid e\,;\ldots;\,e \mid \lambda\, x\,.\, e \mid e\; e & v ::= \; () \mid n \mid \alpha \mid \langle\, e\,,\ldots,\,e\,\rangle \\
\mid & \text{for } x \Leftarrow e \text{ do } e \mid \text{sum } x \Leftarrow e \text{ of } e & o ::= \; e \mid \forall\; x\; o \\
\mid & e + e \mid e \cdot e \mid \text{ref } e\; e \mid (\, e\,,\ldots,\,e\,) & \\
\mid & \text{assert } \mathcal{C} \mid \text{optimum-ref } x\; e & x, y \in \text{vars} \quad n \in \mathbb{Z} \quad \alpha, \beta \in \mathbb{R} \\
\mid & \text{minimize } o\; e \mid \text{minsynth } o\; e & f, g \in \text{funcs}
\end{array}
$$

Figure 4.3: The Synthitar source language, with differences from Scimitar highlighted in blue

This program is only satisfiable for h=1, which forces g=2.5 and g_1=2.0 so we substitute and search for a new counterexample. This time, no matter what counterexample candidate is proposed, the checker validates it. We conclude that h=1 is the satisfying hole assignment.

4.3 Language

As stated above, the Synthitar language is an extension to Scimitar, whose grammar we reproduce in Figure 4.3. We will focus on the Synthitar extensions highlighted in blue, and only include the base grammar in gray for context. The core difference from Scimitar is the new minsynth $o\ e$ term (as well as maxsynth, without loss of generality). This term invokes OACIS to synthesize the expression e, while minimizing the objective function o. To construct their synthesis problem, users can include holes using ??. Akin to the symbolic term, ?? does not introduce names—to name these variables, they have to be bound by the user.

In Scimitar, we used a restricted objective function grammar to parallel the traditional optimization objective structure. Such a formulation is overly restrictive to represent Synthitar problems with universally quantified variables, so we expand the grammar to allow arbitrary expressions. Additionally, we extend objectives with the form $\forall x\ o$ that allows users to introduce a named universally quantified variable x that is in scope in the nested objective expression o. These are the input variables used by OACIS. By nesting $\forall$ terms, we can introduce an arbitrary number of input variables. Because this form is only available in objective functions, users cannot introduce universally quantified variables at any other point in their program, guaranteeing well formed OACIS problems. When all input variables are introduced, the objective function can be completed using an arbitrary expression e.

4.4 Synthesizing Using Objective Functions

As mentioned in Section 4.2, a crucial design decision for optimization-aided synthesis is how to define optimality. Feasible MILP problems have a unique global optimum. Feasible Synthitar programs, on the other hand, have a local optimum for every possible combination of inputs and holes. It is therefore necessary to choose which local optimum to return, and by extension, the hole assignment that leads to that optimum.

In this section we will discuss four different possible definitions for the global optimum of a synthesis problem: *Best Overall* (BO), *Case by Case* (CC), *Best of the Worst* (BW), and Synthitar's choice, *Best of the Best* (BB). While other definitions are possible, we feel that these four are the

Table 4.1: Hypothetical results showing response time to different medication dosages. The first column gives the dosage. The optimal value $r^\star$ is the best case number of hours for that dosage to take effect. The pessimal value $r^\dagger$ gives the worst-case number of hours for that dosage to take effect.

Dosage	$r^\star$	$r^\dagger$
$2.0\,\mathrm{mg}$	$3\,\mathrm{h}$	$5\,\mathrm{h}$
$1.5\,\mathrm{mg}$	$6\,\mathrm{h}$	$9\,\mathrm{h}$
$1.0\,\mathrm{mg}$	$7\,\mathrm{h}$	$8\,\mathrm{h}$
$0.5\,\mathrm{mg}$	$7\,\mathrm{h}$	—

most intuitive and give the most meaningful results. With the help of variations on a simple example, we will analyze each approach's strengths and weaknesses, and justify SYNTHITAR's decision to use BB.

Medication Dosing. To assist our discussion, we will make use of one more small example, a hypothetical SYNTHITAR program encoding a model of a medical trial to determine medication dosing. The program's objective is to choose a recommended dosage of a new medication that across all patient profiles takes effect in the minimum time while avoiding adverse reactions in any patient. The main hole in the model is thus the dosage, while as input it takes the metabolic parameters for a given patient, which are used to model how the medication takes effect.

In Table 4.1 we show some hypothetical data that the solver may contend with to make its decision. The first column gives four possibilities for dosing, from $0.5\,\mathrm{mg}$ to $2.0\,\mathrm{mg}$. The next columns give the optimal value $r^\star$, which represents the number of hours, ranging from 3 to 7, the best case patient took to respond to the medication, and pessimal value $r^\dagger$, the response time of the worst-case patient, ranging from 5 hours to no observed effect.

Regardless of the optimization strategy, the first step of synthesis is to select the optimal value for any hole assignment for any input. In this case, the initial solution would be a $2.0\,\mathrm{mg}$ dose where $r^\star = 3$. From here, however, the behavior of different strategies will diverge. We will see how the differences between each definition of optimality may lead to different dosing guidance. As part of this exploration, we will also see how each strategy responds to variations on the example using modified data.

4.4.1 Best Overall

The first strategy we will consider is choosing a global optimum that is head-and-shoulders better than all others. For the dosage example, the initial solution of $2.0\,\mathrm{mg}$ has a pessimal objective value $r^\dagger = 5$, which is smaller than $r^\star$ for the other possible hole assignments, making this dosage the overall winner for the BO strategy.

More rigorously, to uncover the BO global optimum, we need two criteria: (1) whether this hole assignment has any inputs for which the program is infeasible (correctness counterexamples), and (2) whether the objective function's *pessimal* result is worse than the optimum of some other hole assignments (an overlap counterexample). This pessimal result is the value reached when reversing the direction of optimization, i.e., if we are minimizing an objective function, then its

pessimal value will be the maximum of an objective function (potentially infinite), and vice versa. After finding that value, we merely need to find one other hole assignment with a conflicting optimum—if none exists, then this hole assignment is the global optimum for the BO strategy. By ensuring that one hole assignment's pessimal objective value is better than the optimums for all other assignments, we guarantee that this assignment is unambiguously better than all others.

It may appear that BO should be the obvious choice for a synthesis system, but it is not so clear cut; to understand why, we will make a slight change to our medical trial. Say that a dosage of $2.0\,\mathrm{mg}$ actually causes an adverse reaction, which leads to the model failing for some person. This reaction would result in a correctness counterexample, and upon rerunning the solver would choose a dosage of $1.5\,\mathrm{mg}$. However, this new solution is *not* the global optimum for the BO strategy—its pessimal value $r^\dagger = 9$ is larger than the optimum of the other possible hole assignments, which are overlap counterexamples. Because of that adverse reaction, the global optimum for the BO strategy *does not exist*, and synthesis fails.

From this we can conclude that the BO strategy is powerful but fragile—if a global optimum is chosen, it is as good or better than any result from any other strategy, but if there is no clear winner, synthesis will fail. While it is an intuitive strategy, this drawback makes it a poor choice for Synthitar.

4.4.2 Case by Case

The next strategy of interest chooses a global optimum whose hole assignment's objective function is better for each input individually than that of any other hole assignment. Returning to the dosage example, while the $2.0\,\mathrm{mg}$ dosage is also the CC optimum, let us suppose we were to modify the data such that $2.0\,\mathrm{mg}$ and $1.5\,\mathrm{mg}$ both cause adverse reactions. If that were true, according to this strategy the choice of a $1.0\,\mathrm{mg}$ dosage will actually succeed, as it is better on a case-by-case basis than the $0.5\,\mathrm{mg}$ dosage (assuming that the data is internally monotonic). However, when only the $2.0\,\mathrm{mg}$ dosage causes adverse reactions as in the BO strategy, the CC global optimum does not exist.

To succeed, after finding a candidate the CC strategy must guarantee that there is no other satisfying hole assignment where the objective function is better for some particular input value. More formally, for the objective function $f(h, x)$ (over hole h and input x) and candidate hole assignment h_1, it must be the case that $\forall h, x. f(h_1, x) \preceq f(h, x)$. Therefore an overlap counterexample only occurs if some other hole assignment $h_2 \neq h_1$ exists where the objective value on some concrete x improves over h_1's objective value for the same x. This condition means that every BO optimum is also a CC optimum, but unlike with BO, the results from overlapping objective functions do not necessarily lead to synthesis failing.

The benefit of the CC approach is that if the global optimum exists, it uses the best hole assignment no matter what individual input value is supplied. The approach's largest drawback is that it is computationally intensive, requiring a potentially intractable level of verification. We decided that this made it a poor fit for Synthitar, and we did not explore it further.

4.4.3 Best of the Worst

One less-intuitive strategy worth exploring is Best of the Worst, which chooses the global optimum to avoid a worst-case scenario. Specifically, BW finds the optimum for the hole assignment whose pessimal objective value is *less* pessimal than the pessimal value for all other hole assignments.

Recall that the CC strategy had no global optimum in the dosage example if only the $2.0\,\mathrm{mg}$ dosage causes adverse reactions. A strategy that only considers the pessimal value will succeed in spite of that reaction, choosing the $1.0\,\mathrm{mg}$ dosage. This makes the "least worst" a sensible choice in the context of a drug trial.

The BW strategy generalizes the CC strategy by relaxing the individual optimality requirement to not apply to the objective function value for every input, but only to apply only to the objective function's pessimal value, regardless of which input it is for. Formally, for the objective function f, candidate hole assignment h_1, and input value x_1, this strategy requires that $\forall h.\exists x_2 \forall x.f(h_1,x) \preceq f(h_1,x_1) \wedge f(h,x) \preceq f(h,x_2) \wedge f(h_1,x_1) \preceq f(h,x_2)$. Observe that the value x_1 and existentially quantified x_2 used in this formula are both inputs that yield the pessimal objective value for the respective hole assignment. This condition means that every CC optimum is also a BW optimum, but unlike with CC, the global optimum is also allowed to be entirely contained by some other result. Accordingly, this approach no longer has overlap counterexamples.

The appeal is that by choosing a hole assignment with the best pessimal objective value, it guarantees that no matter what the input is to the program, the worst result will still be better than it would have been for any other hole assignment. The downside is that the BW strategy is a *minimax* problem, the complexity of which far exceeds that of other strategies. Like with CC, this led us to decide that it was a poor fit for Synthitar.

4.4.4 Best of the Best

The final strategy we will consider is the one used by Synthitar: Best of the Best, which directly uses the initial solution as the global optimum without any further analysis. In the dosage example modified so that the $2.0\,\mathrm{mg}$ dosage causes adverse reactions, this strategy chooses the $1.5\,\mathrm{mg}$ dose. It makes this choice because $r^\star = 6\,\mathrm{h}$ is shorter than the other dosages without adverse effects.

Conceptually, BB chooses a global optimum where the hole assignment's objective function is better for at least *some* input than for *all other* holes for *all* inputs. That is, for the objective f, candidate hole assignment h_1, and input value x_1, the formula $\forall h, x.f(h_1,x_1) \preceq f(h,x)$ must hold. This generalizes the CC strategy to only apply to the optimal value regardless of input, similar (but opposite) of BW. This condition means that whenever the BO or CC optimum exists, it will be the same as the BB optimum, but BB does not perform any further checks, meaning the internal distribution of the objective function does not matter—it is chosen not by some group property, but purely using its optimum as a group representative.

The largest appeal of BB in Synthitar is that by avoiding further analysis, it is theoretical much cheaper while still giving similar results. The downside of the BB approach is that the hole assignment chosen may only do really well in one particular case, and may by pessimal otherwise.

We deemed this tradeoff to be acceptable, which is why BB is Synthitar's chosen strategy.

4.4.5 Optimizing the Babylonian method

Let us revisit Section 4.2.1's Babylonian method example to reflect on how each of these strategies would behave. Recall that the example's objective was to minimize the worst-case number of iterations to convergence.

The full hole assignment from the initial solve depends on the user-defined bound for the input domain `nneg`, but the assignment to `max-iterations` is guaranteed to be 0, meaning the square root can be made to converge on the first try, because the solver will cheat and chose an estimated initial x_n equal to the root of the input. For non-degenerate definitions of `nneg` where the upper bound is strictly greater than the lower bound, this will immediately lead to a correctness counterexample, and the number of required iterations will shoot up. However, because the objective is to optimize the hole `max-iterations`, once the verifier can no longer find a correctness counterexample we are guaranteed to have found the global optimum for all four strategies, as a hole assignment is a constant value, making the objective's pessimal value equal to the optimum.[1]

As we did for the medication dosage example, let's see what happens when we tweak the problem. Say we change our Babylonian method implementation to minimize the formula `(- (* 2 max-iterations) gas y)`. This objective has the effect of trying to balance minimizing the worst case while also improving results for small inputs. That is, it ensures that as `y` increases, `gas` also rises closer to `max-iterations`, meaning that large inputs are more likely to require more iterations to solve than small ones. Thus, for any given hole assignment, for `nneg`=$[0, n]$, the objective function will range from the optimal value `(- max-iterations n)` (where `y`=`n` and `gas`=`max-iterations`) to the pessimal value `(* 2 max-iterations)` (with `y`=`gas`=0). A quick check reveals that the objective function's range for `max-iterations`=0 is $[-n, 0]$, for `max-iterations`=1 is $[1-n, 2]$, for `max-iterations`=2 is $[2-n, 4]$, etc. As these data points show, as `max-iterations` increases, the optimum increases from $-n$ while the pessimal value increases from zero twice as fast. Thus, every assignment to `max-iterations` will create an objective function range that overlaps with its successor, and therefore there would be no global optimum for the BO strategy. However, because this relative increase is monotonic, we know that the first solution that does not have a correctness counterexample will be the global optimum for the CC, BW, and BB strategies.

Conversely, consider what would happen if the objective's extent were to shrink, that is, if the pessimal and optimal values were to approach each other as `max-iterations` increases. While such cases are not the usual target for traditional optimization, non-linear behavior is much more common in a software context. A programmer may make use of such an objective, for example, to ensure that values in the middle of the input domain use fewer iterations (we will not propose a specific objective, which can take many forms). The optimal hole assignment would have a pessimal point that was worse than other less optimal hole assignments, and so synthesis would always fail for the CC strategy, while the BW and BB strategies would pick different solutions for the global optimum, at opposite ends of the distribution!

[1] This solution will actually be part of a family of equally valid results; because of numeric variation the solver may return values for the initial estimate and averaging factor that are within a small perturbation of each other. This variance is expected due to the nature of the problem.

4.5 OACIS

The traditional CEGIS algorithm synthesizes programs by alternating between two solvers: the *synthesizer* and the *verifier*. The synthesizer is tasked with proposing assignments to holes, while the verifier tries to disqualify a given hole assignment by finding an input counterexample that violates the program specification. This is an *exists-forall* problem:

$$\exists h \forall x . P(h, x)$$

This problem asks whether some hole assignment h exists such that all possible values of x satisfy the formula $P(h, x)$. The synthesizer fills the role of the *exists* solver, while the verifier takes that of the *forall* solver.

Solving this problem is straightforward for SMT, but naïvely substituting a MILP solver as the verifier runs into a problem. While the SMT representation can be encoded only in terms of its holes and inputs, MILP problems use additional symbolic local variables both to allow for more flexibility for users to model their domain using variables created with the symbolic term, and to encode internal program structure during compilation.[2] Due to the intrinsic structure of MILP problems, we cannot completely avoid adding variables when making such encodings—they are necessary to implement control flow, which spreads its state across multiple constraints, and are also used by variable multiplication, vector indexing, and several other syntactic transformations. Accordingly, regardless of whether a user creates one, all useful programs will contain many local variables once compiled. The reason CEGIS using SMT does not require these variables is because its internal model encodes program structure using named nodes, which are not solver variables and do not need to be quantified over.

Accounting for local existential variables is not possible for the traditional CEGIS verifier, which can only handle universally quantified inputs. Including local variables with no other changes would allow the verifier not only to violate the specification, but to break the program itself. To solve this problem, we must replace the original problem's *forall* quantifier with an inner *forall-exists* quantifier pair:

$$\exists h \forall x \exists g . P(h, x, g)$$

Put another way, if any input violates the specification for *all possible* local variable assignments, it is a counterexample, and this hole assignment is not valid.

4.5.1 General CEGIS Loop

To address this problem, we must adapt CEGIS by abstracting out the individual steps of the loop. We call this generalized algorithm *General CEGIS*, shown in Figure 4.4a. Just like traditional CEGIS, given some program sketch P, GenCEGIS searches for an assignment σ that satisfies P's specification. The difference from the original CEGIS loop is that rather than invoking fixed synthesizer and verifiers subroutines, it takes these as the parameters Learner and Oracle, whose names we borrow from the OGIS framework [JS17]. Additionally, it requires the problem preprocessing

[2] These are variables visible only to the solver, and are not to be confused with normal variables in the user's program.

Input: P: The original problem
Input: Learner: The solver that searches for candidates
Input: Oracle: The solver that invalidates candidates
Input: Xform: This function transforms the problem for the solver
Output: The variable assignment σ or INFEASIBLE token

```
function GenCEGIS(P, Learner, Oracle, Xform)
    P_x ← Xform(P)
    Problems ← P_x
    while true do
        σ ← Learner(Problems)
        if σ ≠ INFEASIBLE then
            P' ← P[σ ↾_S]
            σ' ← Oracle(P')
            if σ' ≠ INFEASIBLE then
                Problems ← P_x[σ' ↾_R] ∧ Problems
            else
                return σ
            end if
        else
            return INFEASIBLE
        end if
    end while
end function
```

(a) The General CEGIS Loop

Input: P: The original problem
Output: The hole assignment or INFEASIBLE token

```
function CEGIS(P)
    GenCEGIS(P, SolveSMT, SolveSMT ∘ Not, Identity)
end function
```

(b) The traditional SMT configuration of the CEGIS Loop

Input: P: The original problem
Output: The hole assignment or INFEASIBLE token

```
function OACIS(P)
    GenCEGIS(P, SolveMILP, Max-Δ, Identity)
end function
```

(c) OACIS: The GenCEGIS Loop specialized for synthesizing MILP holes

Input: P: The original problem
Output: The input counterexample or INFEASIBLE token

```
function Max-Δ(P)
    GenCEGIS(P, VerifyMILP, SolveMILP, InvertMILP)
end function
```

(d) Max-Δ: The GenCEGIS Loop specialized for verifying MILP inputs

Figure 4.4: The General CEGIS Loop and its specializations

function XFORM. When supplied, these specialize GENCEGIS for a particular application.

To track the attempts to satisfy the specification, it creates the variable "Problems" and adds P_x, the result of XFORM(P). Note that each specialization performs additional initialization of Problems excluded from the figure for simplicity. Then, it invokes the LEARNER to get a candidate assignment σ. If σ is feasible, it contains values for not only the solver variables in the search set, S, but also for the remaining solver variables in Problems. Using σ in its entirety would lead to an error, so General CEGIS restricts it to just the variables in the search set. The algorithm substitutes this subset, $\sigma \restriction_S$, into the original problem P to create P' which it then submits to the ORACLE to search for a counterexample assignment σ'. If it finds no such assignment, the LEARNER has successfully found a satisfying assignment σ. However, if σ' is feasible, the algorithm restricts it to the set of refuted variables R then substitutes $\sigma' \restriction_R$ into the transformed problem P_x and adds its result to Problems. This accrues the counterexamples so that in the next loop, the LEARNER has to find an assignment that satisfies all of them. This loop continues until either the search space is exhausted and the algorithm produces the special INFEASIBLE token, or a solution is found. All assignments returned include values for all variables in P.

While the use of XFORM is somewhat awkward here, we chose this way to ensure that the original problem P is what is handed to the ORACLE, the importance of which we will see in Section 4.5.4.

4.5.2 Recovering Traditional CEGIS

The General CEGIS loop is nearly identical to the traditional SMT CEGIS loop, which we recover in Figure 4.4b. Clearly, the LEARNER corresponds to the synthesizer, while the ORACLE is the verifier. This specialization uses SOLVESMT as its synthesizer, and no preprocessing due to using the IDENTITY function. As mentioned, CEGIS additionally initializes Problems using some initial assignment σ_0 to the inputs, to get the synthesis problem started; this way the solver only sees a plain exists problem:

$$\text{Problems}_0 = \exists h.P_x(h, \sigma_0(x))$$

Synthesis yields the hole assignment σ which, when substituted into the initial problem:

$$P = \exists h \forall x.P(h, x)$$

produces the problem:

$$P' = \forall x.P(\sigma(h), x)$$

However, SOLVESMT is an exists solver, so it cannot directly handle the forall. This is why the verifier is the composition SOLVESMT $\circ$ NOT, which in the context of Figure 4.4a will solve the complement of the problem P':

$$\neg P' = \neg \forall x.P(\sigma(h), x) = \exists x.\neg P(\sigma(h), x)$$

This formula can then be passed to SOLVESMT to find a counterexample.

4.5.3 OACIS

Using MILP as a solver backend requires a different approach. Recall our modified quantified formula from above:

$$\exists h \forall x \exists g . P(h, x, g)$$

Because there are two quantifier alternations, we can no longer use a forall solver by itself. The solution we adopt in OACIS requires both an inner and an outer GenCEGIS loop. The specialization of the outer GenCEGIS is shown in Figure 4.4c.

Similar to traditional CEGIS, OACIS uses a plain SolveMILP to search, and Identity for preprocessing. Unlike CEGIS, we treat inputs as regular existential variables in the first round:

$$\text{Problems}_0 = \exists h, x, g . P_x(h, x, g)$$

Picking a fixed initial value prevents OACIS from correctly implementing the BB strategy because inputs can participate in the objective function, and we must ensure that the solver is free to assign to them whatever value optimizes the objective. Subsequent iterations use the corresponding counterexamples; the existential copy of the inputs takes on different values according to whatever optimum is reached. To refute, it invokes the lower GenCEGIS loop, Max-Δ, on the problem:

$$P' = \forall x \exists g . P(\sigma(h), x, g)$$

4.5.4 Maximum Variable Delta Verification

The Maximum Variable Delta Verifier algorithm Max-Δ is shown in Figure 4.4d. This algorithm specializes the GenCEGIS loop by using the plain SolveMILP as the Oracle. The algorithm hinges on maximizing and interpreting Δ, which is encapsulated by the Learner function VerifyMILP, shown in Figure 4.7 and discussed in detail further below. We also finally see need for preprocessing, which is specialized to the function InvertMILP, which is charged with constructing the MILP complement of the synthesis target.

Contextualizing this specialization into Figure 4.4a, we can see that InvertMILP is called at the start to precompute the complement P_x, used to initialize Problems. There is an important extra step before we can find a counterexample: the candidate hole assignment σ_h discovered by OACIS in the outer loop included assignments to locals, with which we initialize the complement:

$$\text{Problems}_0 = \exists x . P_x(x, \sigma_h(g))$$

Next, the initialized complement is handed to VerifyMILP, which identifies a counterexample candidate σ. When a candidate is identified, we can plug it to derive the equation:

$$P' = \exists g . P(\sigma(x), g)$$

which must be then double checked by SolveMILP to detect false counterexamples. Any local variable assignment found in this step is evidence that the candidate input counterexample is actually not a counterexample. If a valid local assignment σ' is found, we add the complement problem substitution $P_x[\sigma' \restriction_R]$ to extend Problems. Finally, if no local assignment is found, the candidate input counterexample is promoted to a *true* counterexample—this is then returned from Max-Δ to the outer synthesis loop, and the synthesis process moves on to the next candidate hole

assignment. When no candidate input counterexample can be found for the identified paths, then MAX-Δ has failed to find a counterexample for the hole assignment, and OACIS succeeds.

The MAX-Δ approach is a large departure from traditional CEGIS, which only tasks verifier with solving the lone problem $\neg P'$. At its root, the difference in approaches is a consequence of the difference between SMT and MILP. As stated previously, the existence of local variables in MILP requires the use of a *forall-exists* solver. As MAX-Δ is defined in terms of GENCEGIS, it works by accumulating valid local assignments into the variable Problems. If the algorithm were to invert the Problems variable in its entirety rather than substituting the local assignments into copies of the complement, VERIFYMILP would only have to satisfy a single local assignment, rather than all of them. By substituting them individually, we guarantee that all local assignments observed so far will be taken into account when proposing candidate input counterexamples.

Properties of Complements. In the algorithm for MAX-Δ in Figure 4.4d above, we needed the preprocessing function INVERTMILP to individually compute the complement problem for each counterexample. This operation is the MILP equivalent of SMT's logical negation. Inversion of an SMT formula is trivial—negate the term, then distribute within the formula. Finding a MILP problem's complement is not so easy, but INVERTMILP's fundamental goal remains the same: to construct a problem whose solutions are variable assignments that are infeasible in the original problem. This is different than testing for a completely infeasible problem, as finding an infeasible point does not imply that the problem is either feasible or infeasible. We merely require a point that causes at least one of the original problem's constraints to be violated. To guide our implementation of INVERTMILP, let us consider the properties we want it to have.

In principle, a valid inversion function f for problems P with possible solutions σ should satisfy these properties:

- *well-definedness*, i.e., that all problems have complements:
$$\forall P. \exists P'. f(P) = P'$$
- *complementarity*, i.e., the original problem and complement have no solutions in common:
$$\forall P. \{\sigma \mid \textsc{Solve}(P) = \sigma\} \cap \{\sigma' \mid \textsc{Solve}(f(P)) = \sigma'\} = \emptyset$$
- *covering*, i.e., that all possible solutions either solve the original problem or its complement:
$$\forall P, \sigma. \textsc{Solve}(P) = \sigma \vee \textsc{Solve}(f(P)) = \sigma$$
- *involution* under solving, i.e., that the set of solutions to the complement of a problem's complement is the set of solutions to the original problem[3]:
$$\forall P. \{\sigma \mid \textsc{Solve}(P) = \sigma\} = \{\sigma' \mid \textsc{Solve}(f(f(P))) = \sigma'\}$$

All of these properties are met by logical negation. Next, we investigate whether it is possible to fulfill them when inverting a MILP problem.

[3]Note that, while a nice property for reasoning about the algorithm, involution under solving does not require f itself to be involutive, i.e. $P = f(f(P))$.

$$
\begin{aligned}
e &::= x \mid I \mid v \mid e + e \mid e \cdot e \\
s &::= \mathcal{C} \mid s\,;\, s \\
op &::= \leq \mid = \mid \geq \\
\mathcal{C} &::= \text{assert } e\ op\ e \\
v &::= n \mid \alpha \\
n &\in \mathbb{Z} \quad \alpha \in \mathbb{R} \quad x, I \in \text{variables}
\end{aligned}
$$

Figure 4.5: The constraint language

Encoding the Complement. Before delving into the details of the algorithm, we briefly review the constraint language it operates over, specified in Figure 4.5. The language consists of programs s, which are either sequences $s\,;\,s$ or constraints $\mathcal{C}$ of the form assert $e_1\ op\ e_2$, over expressions e and inequalities op. Expressions include variables x, values v, addition $e + e$, and multiplication $e \cdot e$, while inequalities include the usual less-than-or-equal, equal, and greater-than-or-equal. Values are integers or reals. We distinguish variables I to serve as indicator variables in the algorithm.

With the language in hand, we now discuss the algorithm for encoding MILP problem complements, given in Figure 4.6. Recall that MILP problems are conjunctions of linear inequalities over continuous and integer variables; for the problem to be feasible, the variables must be assigned values such that all constraints are simultaneously feasible. That means that the complement space is exactly the space of variable assignment such that at least one constraint is not feasible.

It is critical to note that while a constraint must be violated, the variable bounds must be preserved. Otherwise, our verifier could produce out-of-range candidate counterexamples. In fact, allowing bound violations would trivialize the algorithm, as we could ignore all of the original problem constraints!

Because MILP does not include any built-in way to specify disjunction or recognize constraint violation, we will require a special encoding, which we implement starting with the overall problem complement constructor InvertMILP, given in Figure 4.6a. Given some original program's body of constraints s, it delegates to the function Invert, finding corresponding complement constraints s' and violation indicator variables $I_0 + \ldots + I_n$. This is the first of the encoding's three key design insights. Each I_j represents a failure mode for some constraint in s—that is to say, $I_j = 1$ if and only if the assignment the solver returns when solving s' violates some original constraint s_k. These are explained in further detail below. InvertMILP constrains the sum of these indicator variables to be at least one, thereby guaranteeing that we find a point in the complement space, provided that one exists.

Figure 4.6b has two syntax-driven cases. The first, Invert$(s_1; s_2)$, simply inverts the two constraint groups and sums their violation indicators. This implementation is sufficient because violation of any one constraint does not depend on any others, and is thus compositional overall. Put another way, a violation of either constraint group individually is a violation of their combination.

The second case, Invert(assert $e\ op\ 0$), gives us the inverse of a single constraint. Without loss of generality, this formulation merely re-arranges the general assert $e_1\ op\ e_2$ form for

$$\textsc{InvertMILP}(s) = s'; I_0 + \ldots + I_n \geq 1$$
$$\textbf{where } \{s', I_0 + \ldots + I_n\} \leftarrow \textsc{Invert}(s)$$

(a) The overall problem complement constructor InvertMILP

$$\begin{aligned}
\textsc{Invert}(s_1; s_2) &= \{s_1'; s_2', I_0 + \ldots + I_j + I_k + \ldots + I_n\} \\
&\textbf{where } \{s_1', I_0 + \ldots + I_j\} \leftarrow \textsc{Invert}(s_1) \\
&\qquad \{s_2', I_k + \ldots + I_n\} \leftarrow \textsc{Invert}(s_2) \\
\textsc{Invert}(\textsf{assert } e \; op \; 0) &= \{s', I_0 + \ldots + I_n\} \\
&\textbf{where } I_{lt}, I_{eq}, I_{gt} \leftarrow \textbf{fresh} \in \{0, 1\} \\
&\qquad I_0 + \ldots + I_n \leftarrow \textsc{Violation}(op, I_{lt}, I_{gt}) \\
&\qquad s' \leftarrow \textsf{assert } I_{lt} + I_{eq} + I_{gt} = 1; \\
&\qquad\qquad \textsf{assert } I_{lt} \cdot (e + \Delta) \leq 0; \\
&\qquad\qquad \textsf{assert } I_{eq} \cdot \; e \qquad = 0; \\
&\qquad\qquad \textsf{assert } I_{gt} \cdot (e - \Delta) \geq 0
\end{aligned}$$

(b) The constraint complement constructor Invert

$$\begin{aligned}
\textsc{Violation}(\leq, I_{lt}, I_{gt}) &= I_{gt} \\
\textsc{Violation}(=, I_{lt}, I_{gt}) &= I_{lt} + I_{gt} \\
\textsc{Violation}(\geq, I_{lt}, I_{gt}) &= I_{lt}
\end{aligned}$$

(c) The violation equation constructor Violation

Figure 4.6: An algorithm for encoding MILP problem complements

Input: P: The complement problem
Output: The input counterexample or INFEASIBLE token
function VERIFYMILP(P)
$\sigma \leftarrow$ MAXIMIZE(Δ, P)
if $\sigma \neq$ INFEASIBLE $\wedge\ \sigma(\Delta) \leq \epsilon$ **then**
return INFEASIBLE
end if
return σ
end function

Figure 4.7: The verification solver VERIFYMILP

simplicity. The encoding's second key design insight is that each constraint has three domains: "less-than," "equal-to," and "greater-than." In other words, exactly one of these mutually exclusive situations will always be true for any variable assignment, either: (1) $e < 0$, (2) $e = 0$, or (3) $e > 0$. Depending on the original constraint's operator *op*, either one or two cases will signal success, while the others are violations. In other words, if the original constraint is $e \geq 0$, then $e < 0$ would be a violation; for $e \leq 0$, $e > 0$ is a violation; and if $e = 0$, then $e < 0$ and $e > 0$ would both be violations. We create three fresh indicator variables I_{lt}, I_{eq}, and I_{gt}, one corresponding to each case. One or both of I_{lt} or I_{gt} will indicate a violation, as captured by the function VIOLATION, shown in Figure 4.6c.

Finally, we reach the heart of the algorithm: the construction of the complement for a single constraint. Each complement subproblem contains four parts: the mutual exclusion constraint that ensures exactly one indicator is active in each complement subproblem, and one constraint for each of the three domains. The first thing to notice in the domain constraints is that because MILP does not include strict inequality, we have to model it through the use of the Δ-offset encoding. This is the third key design insight, and is where MAX-Δ gets half of its name. The variable Δ is global to all constraints in the complement, and its type is constrained to be positive. We can see that $e + \Delta \leq 0$ guarantees that $e < 0$, and conversely that $e - \Delta \geq 0$ guarantees $e > 0$. Now we can take the product of the Δ-offset encoding with the indicator encoding to conditionalize each domain constraint. Thus, the constraint assert $I_{lt} \cdot (e + \Delta) \leq 0$ encodes that either $I_{lt} = 0$ or $e + \Delta \leq 0$ or both. That is to say, if $I_{lt} = 1$, the constraint is *active* and must hold, but otherwise it is *inactive* and it may or may not hold; conversely, if the constraint cannot hold, then it must be the case that $I_{lt} = 0$, otherwise it can be either. This is a one-way implication, but earlier we stated that $I_j = 1$ if and only if the assignment the solver returns when solving s' violates the original constraint s_k. This additional requirement is why the construction includes all three domains: the mutual exclusion constraint forces exactly one indicator from each trio to be 1, and the corresponding constraint must hold. These indicators are what transform a nominally conjunctive MILP formulation into the disjunction required for a complement—because only one $I_j = 1$ is necessary, the other constraints can be safely made inactive by setting their corresponding indicators to 0.

Maximizing Δ. Now that we have an encoding, we can turn our attention to the solver that handles it, VerifyMILP, shown in Figure 4.7. This function invokes a MILP solver on a complement problem P, maximizing the variable Δ as its objective function. By doing so, it finds the variable assignment which most violates the original problem.

Using the constraint assert $I_{lt} \cdot (e + \Delta) \leq 0$ to demonstrate, assume that $I_{lt} = 1$, and that Δ is as large as possible. This would mean that the expression e is as negative as possible—if it could be made more negative, then Δ would be correspondingly larger. Likewise, e must be the most extreme value over all others—if a different constraint could be made more extreme, by enabling it the solver could find a larger value of Δ, and would then set the indicator I_{lt} to 0, disabling our original constraint. While it is true that any $\Delta > 0$ is enough to signal a violation, searching for the largest Δ is a useful strategy for programs with linear constraints, as it simultaneously pushes the solver away from the trivial solution $\Delta = 0$ while increasing the liklihood of eliminating large swaths of the search space, an effect that we observed in some benchmarks.

In theory, this problem should always be satisfiable, because when $\Delta = 0$ we are able to recover the original problem, which would mean that the solution is not a counterexample. When this circumstance is detected, OACIS terminates. In practice we choose a concrete value $\Delta \leq \epsilon$ as a cutoff to give users better control over the verifier's performance, as well as to avoid concerns about numerical tolerances and precision inherent to the solver algorithm. The value of ϵ is application dependent and up to the user, and should be smaller than any number needed by their program but no smaller to help lead to efficient termination.

Numeric Limitations of MILP. In practice, the complement properties we established above are stronger than necessary for a correct inversion function. Only complementarity is strictly required for correctness—well-definedness, covering, and involution are only to guarantee completeness. If we relax well-definedness, some programs will be inadmissible. If we relax covering, then there will be some variable assignments that are infeasible for Solve(P), but that Solve($f(P)$) will be unable to find. If we relax involution, then the set of solutions to $f(f(P))$ will be a subset of the set of solutions to P rather than an identical set.

To design a practical inversion function, we must account for real world constraints imposed by existing numerical solvers that require InvertMILP to relax covering and involution. Due to precision issues inherent in floating point, relaxing covering means that there may be variable assignments that real solvers will be unable to discover in either the original problem or its complement, for example, because of numerical tolerances needed by the solver to converge on a solution. Relaxing involution in this context means these issues are worsened, as this undiscoverable set of assignments in the complement is further increased in the complement of the complement.

Thankfully, Max-Δ can easily sidestep the issue in the case of involution by keeping P around, as we have done in General CEGIS. We are not so lucky with covering, however, as MILP's lack of strict inequalities necessitates the use of Δ and ϵ, which creates the possibility that some points are excluded from the complement's solution space. Additionally, Synthitar's implementation relaxes well-definedness, although only very narrowly, due to additional design decisions

in SCIMITAR's constraint language. In practice, this means that some problems cannot be inverted, and others may have counterexamples that can't be identified because they are outside the precision specified by the user, which we discuss further in Section 4.6.2.

4.6 Evaluation

To evaluate SYNTHITAR, we supplement the examples presented in Section 4.2 with a suite of microbenchmarks that exercise various aspects of the language. We measured each benchmark's median run time performance in milliseconds during each phase of OACIS, along with number of iterations it took to solve. We use our measurements to observe OACIS on different kinds of tasks, and analyze its behavior and the implications.

4.6.1 Benchmarks

We group our benchmarks into five categories, described below: *triv*, *eq*, *math*, *ite*, and *sqrt*.

Triv. The *triv* category is a group of microbenchmarks limited to using holes and inputs independently of each other. This group includes the benchmarks *triv-prim*, *triv-unit*, *triv-bu*, and *triv-bit*.

Eq. The *eq* microbenchmarks include constraints on the holes and inputs using simple expressions including data structures, and is made up of the benchmarks *eq-bit*, *eq-int*, *eq-vec*, *eq-tuple*, and *eq-unit*.

Math. We group microbenchmarks that constrain the holes and inputs via more complex mathematical and programming relations into the *math* category. These are slightly more complex than the preceding categories, and include the benchmarks *math-sub*, *math-mul*, *math-square*, *math-lb*, and *math-fun*.

ITE. The *ite* benchmark group explores the behavior of loop guards. We previously discussed the example *ite-scale* from Figure 4.2, to which we add a variation *ite-choice* that uses Boolean logic in the guard.

Sqrt. We give a simplified version for *sqrt* in Figure 4.1. The real version has several minor differences from the one presented. While most are superficial, the one significant change is that SYNTHITAR does not directly support floating point variable-variable multiplication or division, and the benchmark uses helper functions to emulate it. For our measurements, we choose the relatively tight range $[0, 5]$ for `nneg` and high `error-tolerance` of $\frac{1}{9}$, to speed up synthesis. These choices result in an optimal hole assignment of `max-iterations`=3, an averaging factor of 0.49–0.5, and 1.73–2.26 as the initial value.

Table 4.2: Solve time in milliseconds and iterations for each phase of OACIS for each SYNTHITAR benchmark, segmented by category

	t_s	t_v	t_c	n_s	n_v
triv-prim	**0.5** ms	0.4 ms	—	1	1
triv-unit	0.2 ms	**0.3** ms	—	1	1
triv-bu	0.2 ms	**7.3** ms	0.2 ms	1	2
triv-bit	0.1 ms	**6.4** ms	0.1 ms	1	2
eq-int	**0.5** ms	0.2 ms	—	1	1
eq-vec	**0.2** ms	0.1 ms	—	1	1
eq-tuple	0.7 ms	**8.0** ms	0.2 ms	3	1
eq-unit	**0.1** ms	0.1 ms	—	1	1
eq-bit	0.2 ms	**5.3** ms	0.2 ms	1	2
math-mul	1.0 ms	**587** ms	0.2 ms	3	1
math-square	**0.5** ms	0.1 ms	—	1	1
math-lb	**0.2** ms	0.1 ms	—	1	1
math-fun	0.1 ms	**9.1** ms	0.1 ms	1	2
math-sub	0.2 ms	**0.8** ms	0.2 ms	1	2
ite-choice	2.4 ms	**9.0** ms	0.6 ms	2	1
ite-scale	1.1 ms	**6150** ms	48 ms	2	74
sqrt	**60 586** ms	24 ms	87 ms	5	2

4.6.2 Results

Our evaluation focuses on two specific dimensions: performance of the different phases and convergence of the algorithm.

Precision and Correctness. OACIS's resulting feasible hole assignment is guaranteed to optimize a problem's objective function within the precision of a user supplied margin of error and of Gurobi's internal tolerances. The correctness of the results of MAX-Δ is guaranteed up to the granularity specified by the previously discussed user-specified parameter ϵ, and so all optimal hole assignments are guaranteed to be free from input counterexamples up to the resolution of ϵ. This definition deviates from the usual definition of soundness in synthesis systems, but does not present a problem provided that users ensure that they design their problems accordingly. The restriction on correctness is a consequence of the nature of discrete algorithms solving for continuous values. Indeed, if MAX-Δ did not impose this qualification on its counterexamples, the user would nevertheless be subject to the definition of floating point and all of its accompanying soundness issues.

Performance. First, we look at the performance of each phase of OACIS. Depending on the nature of user's program, the algorithm may spend more time in synthesis, verification, or counterexample checking.

Columns 2–4 of Table 4.2 give the median run time performance in milliseconds of each benchmark during each phase. As with SCIMITAR, our evaluation used the popular off-the-shelf

Gurobi MILP solver [Gur23]. All measurements were taken on a 3.2 GHz AMD Ryzen 5 1600 system with 32 GB of RAM using Racket's `current-inexact-milliseconds` function.

More specifically, the t_s column gives the sum of the running time of all calls to the synthesizer across all benchmarks in the category for a single execution of the benchmark suite. The measurement shown is the median of 11 runs. Likewise for the time t_v spent in the verifier, and t_c in the checker. Because we take the median across all runs for each phase independently of the others, the three measurements for a given benchmark may not necessarily correspond to any single run. As the bolded columns show, verification time vastly dominates most microbenchmarks, with the most extreme examples in each category being *triv-bit* (97.0%), *eq-bit* (93.4%), *math-mul* (99.8%), and *ite-scale* (99.2%), while *sqrt* spends virtually all of its time (99.8%) in the synthesizer. No benchmark spent any significant time in the checker.

Convergence. Next, we break down the convergence of the algorithm. This is a measure of how many calls were made to the synthesizer, and the worst case number of calls to the verifier within a single execution of Max-Δ.

The fifth and sixth columns of Table 4.2 measure the number of steps to get a result in each benchmark. The n_s column lists the number of iterations through the OACIS loop (calls to the synthesizer) before synthesis succeeded for the median run. The n_v column gives the high watermark for number of iterations through the Max-Δ loop (calls to the verifier) for the median run. We choose the high watermark rather than the total because in most of the benchmarks we investigated, the verifier almost always finds true counterexamples on the first try if one exists, but may run many times otherwise, which means that most verifier runs happen after finding the optimal hole assignment. We do not give the number of calls to the checker, which is always $n_v - 1$, as the only time Max-Δ does not run the checker is if the verifier did not find a counterexample, which only happens once. Because most of our microbenchmarks are quite straightforward, most converge within two calls to either phase, so we will devote the most discussion to *math-mul* (3 synthesis iterations), *ite-scale* (74 worst-case verifier calls), and *sqrt* (5 synthesis and 2 worst-case verifier calls).

4.6.3 Analysis

To explain our results, we must consider the root cause of their behavior. To do so, we must understand a little bit about three different kinds of complement problems, which we dub *empty*, *pathological*, and *tractable*. *Empty* complements are infeasible—either they contain constraints that could never be feasible, such as $1 = 0$, or combinations of constraints that lead to a contradiction, meaning there are no input counterexamples. *Pathological* complements arise when some input is constrained directly to a local variable. These local variables are given concrete assignments during verification, resulting in a complement that searches for a counterexample to just that value. For example, for the constraint `(assert (= x g))`, the assignment `g`=5.0 would mean that the complement would only need to find a counterexample to `(assert (= x 5.0))`. Such constraints are extremely easy to violate because there could be a large number of possible input counterexample candidates (is this constraint, anything in the domain of `x` other than 5.0). Unfor-

tunately, the better the hole assignment, the more of those will be false counterexamples, meaning that the checker will reject them and trigger another MAX-Δ iteration. In the worst case, MAX-Δ will explore the entire input domain looking for an input counterexample. Even a single such constraint is enough to make the complement pathological. *Tractable* constraints are those that involve primarily inputs and holes, so named because these improve the rate of progress through the algorithm, as they increase the odds that the verifier finds true counterexamples.

Of the sixteen microbenchmarks, synthesis dominates six: *triv-prim*, *eq-int*, *eq-vec*, *eq-unit*, *math-square*, and *math-lb*. In each of these cases, the checker does not run even once, meaning that the first call to the synthesizer picked a hole assignment for which the verifier can find no counterexamples. The foremost reason for this behavior is that the five *eq* and *math* complements are all empty, making any hole assignment that satisfies the synthesizer an optimal one. This situation arises because for each of these, the holes are constrained independently of the inputs, which are either unused or have a tightly constrained domain. In the case of *triv-prim*, the complement is actually technically pathological, but the input domain is so restricted that this does not create a problem in practice.

The remaining ten microbenchmarks were dominated by the verifier. The *triv-unit*, *triv-bu*, *triv-bit*, *eq-bit*, *math-fun*, and *math-sub* benchmarks are all simple problems and synthesize very quickly, but their complements are all pathological; however, their input domains are sufficiently restricted that verification is still expedient. Both *eq-tuple* and *math-mul* are tractable problems—while they synthesize quickly, the verifier can immediately find true counterexamples, so they each require a few attempts before finding a solution. The *ite-choice* and *ite-scale* benchmarks have the most complex synthesis problems so far, but their pathological complements drive up the overall verifier time.

Of the microbenchmarks, *math-mul* and *ite-scale* are the most deserving of further discussion. Unlike most other microbenchmarks, these have a comparatively large input domain, and use their inputs and holes in more elaborate constraints which are more demanding for the verifier. The *math-mul* complement contains constraints that impose a non-linear relationship between the inputs and holes—specifically, they include expressions that take the product of one input with the square of the other—which results in the largest complement problem of all our benchmarks. This added complexity works to our advantage, because it means that although each verifier call may take more time, it is more likely to uncover a true counterexample, which is why in the worst case $n_v = 1$. The effect of the combination of a pathological constraint and a large input domain is especially evident in *ite-scale*, where no individual verifier run takes more than $461\,\mathrm{ms}$, but it finds $n_v - 1 = 73$ false counterexamples in its last iteration before giving up and accepting the hole assignment.

Finally, the *sqrt* benchmark is overwhelmingly dominated by synthesis. The increase in synthesis time is easily explained by the fact that the problem submitted to the solver contains over ten times as many constraints and ten times as many variables than either *ite*, which in turn have around ten times as many constraints and ten times as many variables as the remaining microbenchmarks. Conversely, while its complement problem is pathological, the input domain is small so counterexample candidates are chosen and double checked quickly. Furthermore, the time

spent in the synthesis phase appears to rise exponentially with each additional counterexample, with each call taking from four to ten times longer than the previous. This unexpected behavior hints at some sort of underlying combinatorial conflict between the holes when confronted with multiple input values to satisfy.

There are several high-level observations we can derive from this analysis. The most significant is that although the MAX-Δ approach explores the counterexample search space efficiently, pathological complements can negate that efficiency. For example, although *math-fun* includes both pathological and tractable constraints, the verifier's behavior is determined primarily by the pathological ones. To make matters worse, in addition to user-specified constraints, pathological constraints can appear during a SYNTHITAR program's compilation. This behavior is a weakness of the implementation that could be improved with further engineering, but it remains a hard problem to avoid in general. Another observation is that speed of checking is generally negligible, meaning that the cost of the MAX-Δ algorithm comes primarily from the number of times the verifier is invoked. Moreover, the final call to MAX-Δ will almost always dominate the previous calls, as the odds are good that the correct hole assignment will produce a lot of false counterexamples, especially in pathological complements.

Possible future improvements to OACIS could ameliorate these concerns.

4.7 Related Work

Optimization in Synthesis To our knowledge, SYNTHITAR is the first work that successfully encodes programs as MILP problems for general purpose optimization-aided synthesis, but others have explored other methods and applications of optimization in synthesis.

Most prominently, SYNAPSE [BTGC16] employs the strategy of running multiple traditional CEGIS loops in parallel with a central controller to propose candidate solutions, which it judges according to some user-defined cost function κ expressible as a term in a decidable theory $\mathcal{T}$, then directs further search according to some gradient function. Unlike previous work, a SYNAPSE cost function can reason both about the program's syntax and semantics, similar to SYNTHITAR. The work remains more general though, and assumes existing encodings for any $\mathcal{T}$, so can not exploit their structure as SYNTHITAR does.

RESYN [KWPH19b] introduces resource-guided synthesis, a technique that uses a refinement-based type system to specify cost functions that perform amortized resource analysis. The work focuses on type-driven synthesis of programs that are efficient with respect to memory, time, or other domain specific resource metrics. While the type system is based on constrained linear inequalities, they make use of SMT in a custom CEGIS-like algorithm rather than use numeric solvers.

STITCH [BOW+23] is a tool for synthesizing library functions that uses a generic utility function containing an compression objective that seeks to maximize the product of the size of the abstraction and the number of locations where the abstraction can be used. The theory behind the approach is that an objective function that measures how compressed the program is will lead to compact or even minimal code generation. Unlike other approaches, it does not employ CEGIS

or solvers, but introduces a technique called *corpus-guided top-down search*, which iteratively constructs and refines syntax trees that meet the specification and maximize the utility measure.

Euler [CSL12] is a system for solving unconstrained optimization problems posed as sketches. While it is not a complete synthesis system, it includes many of the necessary core concepts, including holes and objective functions. The grammar and semantics are a subset of what Synthitar provides, and the algorithm's end results are akin to a single pass through OACIS. They accomplish their optimization via smooth interpretation, discussed in more detail below.

Generalizing CEGIS Synthitar is not the first attempt to generalize CEGIS. Abstract learning frameworks [LMN16, NSM16, NGM$^+$18] introduce a formalism that encompasses several CEGIS formulations as well as other synthesis algorithms not generally viewed as CEGIS, and demonstrate its application to novel domains. The framework formalism generalizes synthesizers and verifiers in terms of *learners* and *teachers* (or *oracles*), along with their respective *sample* and *hypothesis* spaces. OGIS [JS17] creates a conceptually similar formulation. Both of these approaches are more formal than our work, and provide a more complete treatment. The goal of General CEGIS is not to create an independent framework, just to increase the flexibility of CEGIS to adapt it a new domain, and can be regarded as a specialization of these approaches.

Synthesizing Numeric and Constraint Programs Many attempts have been made to synthesize programs that include numeric equations and arithmetic constraints. One example is Synmio [WGW23], a syntax guided synthesis (SyGuS) method that uses CEGIS to convert arbitrary Boolean logic formulas by synthesizing MILP constraints that solve more efficiently than encodings such as the ones used by Synthitar. Another, DSSynth [ABC$^+$20] uses CEGIS to synthesize digital controllers for physical systems which include A/D and D/A converters using linear time invariant models expressed as ordinary differential equations. DryadSynth [HQSW20b] approaches SyGuS problems over conditional linear integer arithmetic (CLIA) by employing a novel divide-and-conquer strategy employing CEGIS. nay [HCDR20b] goes even further, proving that CLIA SyGus problems over finitely many examples is decidable. Unlike Synthitar, none of these make use of optimizing objective functions during synthesis, instead just focusing on correctness.

The smooth interpretation [CSL10b] algorithm uses Gaussian smoothing combined with Nelder-Mead simplex search to synthesize optimal control parameters for embedded control applications by creating a continuous approximation of a discrete program then employing an optimization objective that minimizes the error of the synthesized result. While it makes use of optimization as a tool, it only does so within the algorithm, and does not accept arbitrary objectives as Synthitar does.

Synthesizing Floating Point Programs CEGIS synthesis approaches often struggle in the presence of floating point holes and inputs. Most existing work is concerned with correctness of floating point theories in the context of SMT, but there has been some work specifically addressing synthesizing floating point values. Systems such as FPSyn [NJR22] and PINE [IDS20] focus on guaranteeing correct roundoff error behavior via rigorously modeling floating point in

restricted domains of predicates and inductive invariants.

Sketch's approach to floating point [KCISL16, SLAT+07b, SLB25, IGKSL18] uses four different encodings for different circumstances. First, it encodes float literals as a lookup table using a one-hot encoding augmented with some hard-coded ϵ, which is efficient when making decisions about inputs and holes that can be drawn from that concrete set. Second, in some contexts it can assign float variables to scaled temporary bitvectors, which allows reasoning about all values up to the scaling resolution in the relatively small range of the bitvector width. Third, in contexts where the actual floating point logic is not important, it can replace the entire computation with an equisatisfiable problem over uninterpreted functions, or, since the reals form a field, over some other field that is more amenable to synthesis. Finally, there has been exploratory work to support numerical optimization via symbolically computing derivatives of the sketch then running gradient descent combined with Newton's method.

These approaches are completely different than Synthitar's approach to floating point—our ability to synthesize floating point values is a natural consequence arising purely from the use of a MILP solver.

Large Language Models Ever since the introduction of the Transformer model[VSP+17], no discussion of a novel program generation technique is complete without acknowledging its relationship to Large Language Models such as GPT, Claude, and Gemini. The state of the art includes a wide range of quality vs compute [SLB25], but it can be roughly divided into three tiers:

- *Retail.* This includes tools such as OpenAI's ChatGPT, Microsoft's Copilot, and Google's Gemini. These products are generally capable of reproducing or creating small variations of code where many prior examples exist in its training set.
- *Agentic.* These are professional tools such as Devin[4] and Cursor.[5] While more powerful than the retail tier, they struggle with more routine tasks like maintaining or adding features to existing code, and excel at automating rare but formulaic activities like creating projects and scripts. Their success relies on a large amount of search (often dozens of minutes to hours) and good test cases—at their core, though, they use the same models as the retail tier.
- Google DeepMind[LCC+22, LGA+23, Goo24, RPBN+24, MMM+25] remains in a category by itself. Over the last three years this group has demonstrated increasing ability on diverse tasks, succeeding in feats like ranking highly against humans in code and math competitions. Their success has a caveat though—it often requires so much compute that it is not feasible for others to reproduce, and therefore hard to characterize what it is and isn't capable of.

Gu et. al [GJL+25] present a detailed survey of the current abilities of these offerings. They present a taxonomy of software tasks and define measures to evaluate them. According to their rubric, the Babylonian square root example from Figure 4.1 would be considered a function scope, medium-high logical complexity code generation and optimization task. Of these sorts of problems, the authors claim that:

[4] `https://devin.ai`
[5] `https://cursor.com`

> Optimizing real-world systems poses significant challenges due to the large scope and high logical complexity of the task [...]. Code optimization often has a large search and solution space with competing objectives [...].

Later on, they further comment on challenges faced by programs in SYNTHITAR's target domain:

> Tasks such as writing [...] performance optimizations have a high logical complexity, often proving difficult for even the best human coders. Similar to solving research-level math problems, these out-of-distribution domains are very hard for LLMs.

Part of the difficulty, they conclude, is that these tasks are largely absent from the training sets of modern LLMS, as they are unique, domain-specific, and quite challenging for human beings.

Further complicating the matter is the complexity of knowing how to coax models to produce desirable answers. While SYNTHITAR allows for precise logical specification of a goal function, methods for explaining an objective to the LLM in the *natural language to code* paradigm are so mysterious they have launched an independent field of study, known as *prompt engineering*. This topic has attracted attention from both academic researchers trying to characterize models [YSW+25, SBA+25, KdONML25] and the general public that want to make the best use of them [DAI]. This situation hinders direct comparison between LLM approaches and symbolic approaches like SYNTHITAR, as the tools necessary to design good prompts are still in their infancy, and users currently lack the ability to know whether the LLM is unable to complete a task, or if their prompt is merely insufficient.

Finally, LLMs perform especially poorly [SLB25] on over-constrained synthesis problems with very few solutions, discrete problems which lack the benefit of gradients, and problems in unique domains which lack the benefits of pretraining. It is this author's opinion that the best way to address this problem lies in a hybrid approach, by giving LLMs access to existing symbolic tools and training them how to use them effectively. Some efforts have been made in this direction, with the development of standard approaches such as Anthropic's *Model Context Protocol (MCP)* [Ant24], which gives an LLM secure access to external applications. The agentic tools mentioned above are already starting to take advantage of this ability, and we expect its use to only increase. Whether such approaches will be the ultimate solution to the problem, or whether LLMs catch up on their own, only time will tell.

4.8 Summary

Optimization-aided synthesis merges optimization-aided programming with sketch-based synthesis, allowing users to specify objective functions that can encode application goals that are difficult to state with assertions alone. These include syntactic or semantic properties they want to optimize, such as program structure, resource usage, or other domain specific behavior. The OACIS algorithm employed by SYNTHITAR enables this merger by adapting CEGIS to synthesize mixed integer linear programs, employing MAX-Δ verification to find the most extreme counterexample. Our results show that this approach can be used to generate optimal programs, particularly numeric algorithms and ones including floating point values.

Chapter 5

Conclusion and Future Work

This dissertation introduced two novel approaches to leverage information within a sketch to improve synthesis. First we presented Sketcham, which modularly decomposes program sketches using information gleaned from their own test suites to break them apart into mocks without needing the user to supply any additional information. Next we created the optimization-aided language Scimitar, which combines functional and symbolic reasoning by leveraging a mixed integer linear program solver to allow users to write programs including objective functions that can choose optimal values for program variables. Finally, we extend Scimitar with synthesis language constructs including holes, universally quantified inputs, and an optimizing synthesis query, to create the full fledged Synthitar optimization-aided synthesis system, which employs the new OACIS and Max-Δ algorithms. These efforts support our thesis statement from the Introduction:

> The central claim of this dissertation is that augmenting the information about program structure and objectives available to these solver aided systems leads to better results, both when synthesizing programs and when searching for optimal program values.

By modularizing sketches, Sketcham enables users to reason separately about callers and callees. Constructing new subproblems using mocks allows each function to be synthesized in relative isolation, leading to performance gains of as much as $5\times$. Scimitar frees users from needing expertise about encoding techniques that must normally be written by hand, including solve-time conditionals, bounded inlining, and loop unrolling. Its quick solve performance supports the argument in favor of a dedicated MILP solver for optimization applications. Synthitar's integration of optimization with sketching gives users the power to direct synthesis with goals difficult to state with assertions alone. Objective functions specifying such syntactic and semantic properties can influence the size, value, or shape of program values and structure, letting users precisely express how the system should prefer one solution to another.

We see several opportunities to extend and improve on this work. First, Sketcham's transformation currently excludes some more complex asserts that could potentially be usable for mocks, e.g., those including conjunctions of otherwise unrelated terms, and additional Sketch features such as complex harness types. Supporting these would drastically broaden the kinds of programs the algorithm could be applied to.

A meaningful research direction for Scimitar would be to explore recursive algebraic data structures, particularly variable-length lists. Support for these types would allow for a more complete investigation of higher-order functions and optimizing of entire data structures. Another line of research would be on the language's type system. The typing algorithm is powerful but informal, and we believe that creating a formalized type inference system built on MILP is both possible and desirable. One other possible direction is to move past MILP to more powerful, nonlinear approaches such as using a Quadratically Constrained Quadratic Programming (*QCQP*) solver. Doing so would drastically increase the number of functions that could be modeled without a linear relaxation, not least of which is float–float variable multiplication, for which it is not practical to use a MILP solver. Beyond expanding the language design, it would also be worthwhile to perform a user study on the benefits and drawbacks of Scimitar as compared to the traditional optimization problem approach.

Synthitar presents many avenues worth pursuing for further development. The most significant is to integrate more techniques used by state of the art *exists-forall-exists* solvers into OACIS. A prominant example of this would be to upgrade the Max-Δ algorithm to proactively eliminate local variables during preprocessing via Skolemization. While it would be unable to eliminate them entirely, Skolemization would drastically cut down on the number of pathological constraints. Another would be to modularize the sketch in a way similar to Sketcham, by grouping constraints into mocks that fully determine some aspect of the original problem, thereby decreasing the complement's search space by limiting the number of possible false counterexamples. Also, it would be interesting to explore definitions of optimality other than Best of the Best, further increasing the power users can wield in the search of their perfect program. Finally, given the recent advancements in program generation using LLMs, it is important to consider how our work might compare to a potential LLM-based approach to optimal synthesis. Performing such an evaluation could help determine with more certainty whether LLMs are capable of generating programs with objective functions, and if so how their results and performance compare with ours.

We believe this work is evidence that synthesizing modular and optimal programs is not only possible, but worthwhile, and that the approaches we present are valuable contributions toward achieving that purpose.

Appendix A

SKETCHAM Benchmarks

The following is a selection of SKETCHAM's benchmark suite. For a detailed discussion of each benchmark, see Section 2.4.

```
1  generator int constant() { int m = ??; return {| (-)? m |}; }
2  int abs(int n) {
3      if ({| n (< | > | <= | >= | == | !=) constant() |}) {
4          return pos(n);
5      } else {
6          return neg(n);
7      }
8  }
9  generator int xform(int n) { return {| n (+ | - | * | /) constant() |}; }
10 int pos(int n) { return xform(n); }
11 int neg(int n) { return xform(n); }
12
13 harness void ha(int n) { assert abs(n) >=  0; assert abs(-n) >= 0; }
14 harness void hp()      { assert pos(1) ==  1; }
15 harness void hn()      { assert neg(1) == -1; }
```

Figure A.1: The *absval* benchmark

```
pragma options "--bnd-inbits 6 --bnd-inline-amnt 10";

int fib_slow(int n) {
    if (n <= 1) { return 1; }
    return fib_slow(n - ??) + fib_slow(n - ??);
}

harness void h_fib_slow() {
    assert fib_slow( 0) ==    1;
    assert fib_slow( 1) ==    1;
    assert fib_slow( 2) ==    2;
    assert fib_slow( 3) ==    3;
    assert fib_slow( 4) ==    5;
    assert fib_slow( 5) ==    8;
    assert fib_slow( 6) ==   13;
    assert fib_slow( 7) ==   21;
    assert fib_slow( 8) ==   34;
    assert fib_slow( 9) ==   55;
}

int fib_fast(int m) {
    int fib_inner(int n, int accum2, int accum1) {
        if(n <= 1) { return accum1; }
        return fib_inner(n - 1, accum1, accum2 + accum1);
    }
    return fib_inner(m, ??, ??);
}

int fib_hack(int n) {
    return fib_slow(n);
}

harness void h_fib(int n) {
    assume n < 10;
    assert fib_hack(n) == fib_fast(n);
}
```

Figure A.2: The *fib* benchmark

```
include "intops.skh";

adt StrAdt { Str {int n; char[n] s;} }
generator Str str([int n], char[n] s) { return new Str(n = n, s = s); }

int levenshtein([int a_n, int b_n], char[a_n] a_s, char[b_n] b_s) {
    Str a = str(a_s), b = str(b_s);
    int pad = ??;
    int[b.n + pad][a.n + pad] memo = { {0}};
    for (int i = 0; i < a.n + pad; i++) {
        for (int j = 0; j < b.n + pad; j++) {
            if(i == 0 || j == 0) {
                memo[i][j] = i + j;
            } else {
                bit eqInd = a.s[i - 1] != b.s[j - 1];
                memo[i][j] = min(min(memo[i - 1][j] + 1,
                                     memo[i][j - 1] + 1),
                                 memo[i - 1][j - 1] + eqInd);
            }
        }
    }
    return memo[a.n][b.n];
}

harness void h_levenshtein([int n, int m], char[n] s1, char[m] s2) {
    // This case is not covered by the spec below
    assert levenshtein("mother", "father") == 2;
    // There are no really exemplary weak specs for edit distance
    assume n >= 0 && n <= UNROLL_BOUND - 1;
    assume m >= 0 && m <= n;
    if (s1 == s2) { assert levenshtein(s1, s2) == 0; }
    Str sa = str(s1), sb = str(s2);
    int j = 0;
    for(int i = 0; i < sa.n; i++) {
        assume j <= sb.n;
        if (j < sb.n && sa.s[i] == sb.s[j]) { j++; }
    }
    int delta_n = sa.n - sb.n;
    if (j == sb.n) {
        assert levenshtein(s1, s2) == delta_n;
        assert levenshtein(s2, s1) == delta_n;
    } else {
        assert levenshtein(s1, s2)  > delta_n;
        assert levenshtein(s2, s1)  > delta_n;
    }
}
```

Figure A.3: The Levenshtein edit distance algorithm, used by both *spellcheck* and *minpair*

```
pragma options "--slv-nativeints --bnd-unroll-amnt 9";
#define UNROLL_BOUND 9
#include "levenshtein.sk"

Str[n] spellcheck([int n], Str[n] sentence) {
    Str[4] dictionary = {
        str("etch"),
        str("skeet"),
        str("set"),
        str("sketchy")
    };
    Str[n] result = sentence;
    for(int i = 0; {| i (< | <= | ==) n |}; i++) { // i < n
        Str original = sentence[i];
        for(int j = 0; j < ??; j++) { // j < 4
            Str dictword = dictionary[j];
            int score = levenshtein(original.s, dictword.s);
            if (score <= ??) { // score <= 1
                result[i] = dictword;
            }
        }
    }
    return result;
}

harness void h_spell() {
    Str[3] original_sentence = {
        str("It"), str("is"), str("sketch")
    };
    Str[3] fixed_sentence = {
        str("It"), str("is"), str("sketchy")
    };
    assert spellcheck(original_sentence) == fixed_sentence;
}
```

Figure A.4: The *spellcheck* benchmark

```
pragma options "--slv-nativeints --bnd-unroll-amnt 10";
#define UNROLL_BOUND 10
#include "levenshtein.sk"

int min_edit_pair([int n], Str[n] strs, ref Str a, ref Str b) {
    assert n >= 2;
    int best_score = -1;
    for (int j = ??; j < n; j++) {     // 0 or 1
        for (int i = ??; i < j; i++) { // 0
            Str si = strs[i];
            Str sj = strs[j];
            int score = levenshtein(si.s, sj.s);
            if (best_score == -1 || score < best_score) {
                a = si;
                b = sj;
                best_score = score;
                if (best_score == 0) {
                    return best_score;
                }
            }
        }
    }
    return best_score;
}

harness void h_min1() {
    Str a, b;
    Str[5] ss = {
        str("etch"), str("set"),
        str("sketch"), str("sketches"),
        str("sketchy") };
    int score = min_edit_pair(ss, a, b);
    assert score == 1;
    assert a.s == "sketch";
    assert b.s == "sketchy";
}

harness void h_min2() {
    Str a, b;
    Str[4] ss = {
        str("daughter"), str("son"),
        str("mother"), str("father") };
    int score = min_edit_pair(ss, a, b);
    assert score == 2;
    assert a.s == "mother";
    assert b.s == "father";
}
```

Figure A.5: The *minpair* benchmark

```
pragma options "--slv-nativeints --bnd-inbits 2 --bnd-unroll-amnt 4";
include "generators.skh";

int[n] insertion([int n], int[n] vs) {
    for(int i = ??; i < n; ++i) { // 1
        int v = vs[i];
        for(int j = expr({i, n, v}, {PLUS, MINUS});   // i - 1
            exprBool({j, i, n}, {}) &&                // j >= 0
            exprBool({j, i, n, v, vs[j]}, {}); --j) { // vs[j] > v
            vs[j + 1] = vs[j];
            vs[j] = v;
        }
    }
    return vs;
}

generator bit elem([int n], int m, int[n] vs) {
    for(int i = 0; i < n; ++i) { if(vs[i] == m) { return true; } }
    return false;
}

harness void h_sort([int n], int[n] vs) {
    int[n] svs = insertion(vs);
    for(int i = 0; i < n;     ++i) { assert elem(vs[i], svs); }
    for(int i = 0; i < n;     ++i) { assert elem(svs[i], vs); }
    for(int i = 0; i < n - 1; ++i) { assert svs[i] <= svs[i + 1]; }
}

int[n] dedup([int n], int[n] vs, ref int sz) {
    int[n] svs = insertion(vs);
    int[n] result;
    sz = ??; // 0
    for(int i = ??; i < n; ++i) { // 0
        int j = expr({sz, i}, {PLUS, MINUS}); // sz - 1
        if(sz == ?? || {| svs[i] (> | >= | < | <= | != | ==) result[j] |} ) { // 0 >
            result[sz] = svs[i];
            sz = expr({sz, i}, {PLUS, MINUS}); // sz + 1
        }
    }
    return result;
}

harness void h_dedup([int n], int[n] vs) {
    int sz = 0;
    int[n] dvs0 = dedup(vs, sz);
    assert n == 0 || sz > 0;
    int[sz] dvs = dvs0[0::sz];
    for(int i = 0; i < n;      ++i) { assert elem(vs[i], dvs); }
    for(int i = 0; i < sz;     ++i) { assert elem(dvs[i], vs); }
    for(int i = 0; i < sz - 1; ++i) { assert dvs[i] < dvs[i + 1]; }
}
```

Figure A.6: The $dedup_i$ benchmark using insertion sort

```
pragma options "--slv-nativeints --bnd-inbits 2 --bnd-unroll-amnt 4";
include "generators.skh";

int[n] merge([int n], int[n] vs) {
    if (n > 1) {
        int mid = n/2, rem = n - n/2, i = 0, j = 0;
        int[mid] vas = merge(vs[0::mid]);
        int[rem] vbs = merge(vs[mid::rem]);
        while(exprBool({i, j, n}, {PLUS})) { // i + j < n
            if(i < mid && (j == rem || vas[i] < vbs[j])) {
                vs[i + j] = vas[i]; i++;
            } else {
                vs[i + j] = vbs[j]; j++;
            }
        }
    }
    return vs;
}

generator bit elem([int n], int m, int[n] vs) {
    for(int i = 0; i < n; ++i) { if(vs[i] == m) { return true; } }
    return false;
}

harness void h_sort([int n], int[n] vs) {
    int[n] svs = merge(vs);
    for(int i = 0; i < n;     ++i) { assert elem(vs[i], svs); }
    for(int i = 0; i < n;     ++i) { assert elem(svs[i], vs); }
    for(int i = 0; i < n - 1; ++i) { assert svs[i] <= svs[i + 1]; }
}

int[n] dedup([int n], int[n] vs, ref int sz) {
    int[n] svs = merge(vs);
    int[n] result;
    sz = ??; // 0
    for(int i = ??; i < n; ++i) { // 0
        int j = expr({sz, i}, {PLUS, MINUS}); // sz - 1
        if(sz == ?? || {| svs[i] (> | >= | < | <= | != | ==) result[j] |} ) { // 0 >
            result[sz] = svs[i];
            sz = expr({sz, i}, {PLUS, MINUS}); // sz + 1
        }
    }
    return result;
}

harness void h_dedup([int n], int[n] vs) {
    int sz = 0;
    int[n] dvs0 = dedup(vs, sz);
    assert n == 0 || sz > 0;
    int[sz] dvs = dvs0[0::sz];
    for(int i = 0; i < n;      ++i) { assert elem(vs[i], dvs); }
    for(int i = 0; i < sz;     ++i) { assert elem(dvs[i], vs); }
    for(int i = 0; i < sz - 1; ++i) { assert dvs[i] < dvs[i + 1]; }
}
```

Figure A.7: The $dedup_m$ benchmark using merge sort

Appendix B

Formalization of Scimitar's Source Language

B.1 Types

Figure B.1 shows the Scimitar type system. Types provide guarantees about what the solver receives in the compiled code, and enable several important features (see Section 3.4).

Types τ range over a scalar interval $\mathcal{I}$ with a size or "measure" μ, the unit type, tuples $\tau \times \tau$, records $\langle x : \tau, \ldots \rangle$, and functions $\tau \to \tau$.

Tensors are given the type $\mathcal{I}^{\mu}$; vectors are a special case for $\mu = k$. While any closed interval $\mathcal{I}$ of the reals is a valid scalar type, the intervals of most interest to us are the reals $\mathbb{R}$ and the non-negatives $\mathbb{R}_+$, the natural numbers $\mathbb{Z}$, and the set $\{0, 1\}$, $\mathbb{I}$.

Types are built on the idea that we can model any value with a non-recursive algebraic type as a vector of type $\mathbb{R}^k$; other types are sugar on top of this basic type. The implementation flattens tensors, tuples, and records according to the dimensionality and range of their type.

$$
\begin{aligned}
\mathcal{I} &::= \mathbb{R} \mid \mathbb{R}_+ \mid \mathbb{I} \mid \mathbb{Z} \\
\mu &::= k \mid k \times \mu \\
\tau &::= \mathcal{I}^{\mu} \mid () \mid \tau \times \tau \\
&\quad \mid \langle x : \tau, \ldots \rangle \mid \tau \to \tau \\
k &\in \mathbb{Z}
\end{aligned}
$$

Figure B.1: The Scimitar types

B.2 Additional Semantics

In Section 3.3.1 discussed the interface between host and solver semantics for Scimitar. The intuition behind the functional solver semantics can be laid out using a complete trace-based semantics, as presented in Figures B.2, B.3, and B.4. Because the whole program is evaluated at once, we can't analyze individual asserts in isolation to determine whether they will be violated. Even something as simple as $a = b$ could for example make the program infeasible, force a solution,

$$\frac{\alpha, \beta \text{ are global constants}}{\min \alpha \cdot x + \beta; \top \vdash x \rightsquigarrow x}\ \textsc{Var} \qquad \frac{}{\min 0; \top \vdash v \rightsquigarrow v}\ \textsc{Val}$$

$$\frac{\begin{array}{c}\min o_g; \mathcal{C}_g \vdash e_g \rightsquigarrow x_g \qquad \min o_t; \mathcal{C}_t \vdash e_t \rightsquigarrow x_t \qquad \min o_f; \mathcal{C}_f \vdash e_f \rightsquigarrow x_f \\ x \text{ is fresh} \qquad \mathcal{C}_x \equiv x = x_g * x_t + (1 - x_g) * x_f \\ \mathcal{C}_{tg} \equiv x_g \Longrightarrow \mathcal{C}_t \qquad \mathcal{C}_{fg} \equiv (1 - x_g) \Longrightarrow \mathcal{C}_f \\ \mathcal{C} \equiv \mathcal{C}_g \wedge \mathcal{C}_{tg} \wedge \mathcal{C}_{fg} \wedge \mathcal{C}_x\end{array}}{\min o_g + o_t + o_f; \mathcal{C} \vdash \textsf{if } e_g \textsf{ then } e_t \textsf{ else } e_f \rightsquigarrow x}\ \text{ITE}$$

$$\frac{\begin{array}{c}\min o_i; \mathcal{C}_i \vdash e_i \rightsquigarrow y_i \qquad \mathcal{C}_y \equiv y_i = \langle y_1, \ldots, y_n \rangle \qquad n < \text{unroll limit} \\ \min o_1; \mathcal{C}_1 \vdash e_b[x \mapsto y_1] \rightsquigarrow x_1 \\ \vdots \\ \min o_n; \mathcal{C}_n \vdash e_b[x \mapsto y_n] \rightsquigarrow x_n \\ \mathcal{C} \equiv \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_n \wedge \mathcal{C}_i \wedge \mathcal{C}_y\end{array}}{\min o_1 + \ldots + o_n + o_i; \mathcal{C} \vdash \textsf{for } x \Leftarrow e_i \textsf{ do } e_b \rightsquigarrow x_n}\ \textsc{For}$$

$$\frac{\begin{array}{c}\min o_i; \mathcal{C}_i \vdash e_i \rightsquigarrow y_i \qquad \mathcal{C}_y \equiv y_i = \langle y_1, \ldots, y_n \rangle \qquad n < \text{unroll limit} \\ x_s \text{ is fresh} \qquad \mathcal{C}_s \equiv x_s = x_1 + \ldots + x_n \\ \min o_1; \mathcal{C}_1 \vdash e_b[x \mapsto y_1] \rightsquigarrow x_1 \\ \vdots \\ \min o_n; \mathcal{C}_n \vdash e_b[x \mapsto y_n] \rightsquigarrow x_n \\ \mathcal{C} \equiv \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_n \wedge \mathcal{C}_i \wedge \mathcal{C}_y \wedge \mathcal{C}_s\end{array}}{\min o_1 + \ldots + o_n + o_i; \mathcal{C} \vdash \textsf{sum } x \Leftarrow e_i \textsf{ of } e_b \rightsquigarrow x_s}\ \textsc{Sum}$$

Figure B.2: Scimitar's functional solver semantics for variables, values, and loops. The judgment form used here is trace-based: $\min o; \mathcal{C} \vdash e \rightsquigarrow x$

$$\frac{n \geq \text{inline/unroll limit}}{\phi; n; \text{PC} \Longrightarrow \bot \vdash e \Downarrow e} \textsc{InlineFail}$$

$$\frac{}{\min 0; \top \vdash \lambda x \,.\, e \rightsquigarrow (\!|\lambda x \,.\, e|\!)} \textsc{Lambda}$$

$$\frac{\begin{array}{c}\phi = \{f \mapsto e_f\} \\ \phi; n+1; \mathcal{C} \vdash e[f \mapsto e_f] \Downarrow e' \\ n < \text{inline/unroll limit}\end{array}}{\phi; n; \mathcal{C} \vdash e \Downarrow e'} \textsc{Inline}$$

$$\frac{\begin{array}{c}\min o_1; \mathcal{C}_1 \vdash e_1 \rightsquigarrow (\!|\lambda x_f \,.\, e_f|\!) \\ \min o_2; \mathcal{C}_2 \vdash e_2 \rightsquigarrow x_2 \\ \min o_f; \mathcal{C}_f; \vdash e_f[x_f \mapsto x_2] \rightsquigarrow x\end{array}}{\min o_1 + o_2 + o_f; \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \mathcal{C}_f \vdash e_1 e_2 \rightsquigarrow x} \textsc{Apply}$$

$$\frac{\begin{array}{c}\min o_f; \mathcal{C}_f \vdash e_f \rightsquigarrow v_f \\ \phi = \{f \mapsto v_f\} \;\; \phi; 0; \mathcal{C}'_f \vdash v_f \Downarrow v'_f \\ \min o; \mathcal{C} \vdash e[f \mapsto v'_f] \rightsquigarrow x \quad \mathcal{C}' \equiv \mathcal{C}_f \wedge \mathcal{C}'_f \wedge \mathcal{C}\end{array}}{\min o_f + o; \mathcal{C}' \vdash \textsf{letrec } f \Leftarrow e_f \textsf{ in } e \rightsquigarrow x} \textsc{Letrec}$$

Figure B.3: Scimitar's functional solver semantics: functions

or have no effect. Solver semantics that evaluate to concrete values prove difficult to specify, as doing so is tantamount to presenting an entire solver algorithm, e.g. branch and cut. However, our trace-based approach can give us some insight as to what the solver sees before it begins its decision procedure.

The judgment form $\min o; \mathcal{C} \vdash e \rightsquigarrow x$ states that given some MILP problem that minimizes some objective o subject to constraints $\mathcal{C}$, we can find an equivalent Scimitar program expression e that leads to the variable or value x. By "leads to," we mean specifically that building the MILP problem equivalent to the operation e requires the solver to be aware of x, which the problem may use in other constraints. It represents knowledge that the solver has about this expression.

Using these rules, we can explore informally how a user should expect a program to behave within a solve expression. The basic solver semantic concepts are variables and constraints, and our objective is to accumulate these to feed to the solver. The judgment form is read as follows. For an expression e on the right hand side of the turnstile, we introduce constraints $\mathcal{C}$ and a solver objective $\min o$ on the left side. Together, these lead to either a variable x or a value v, where x represents how the solver interprets the constraint(s) introduced by the expression.

Figure B.2 gives the rules for variables, values, conditionals and loops. Var is the only case where an objective is introduced, an affine formula over the variable used in the statement of the problem. Values (Val) can only lead to themselves. Conditionals (ITE) use the objectives of all subexpressions, and the conditional behavior is represented in the constraints: first, we

$$\frac{\min o_1; \mathcal{C}_1 \vdash e_1 \rightsquigarrow x_1 \quad \ldots \quad \min o_n; \mathcal{C}_n \vdash e_n \rightsquigarrow x_n}{\min o_1 + \ldots + o_n; \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_n \vdash e_1; \ldots; e_n \rightsquigarrow x_n} \textsc{Seq}$$

$$\frac{x \text{ is fresh} \qquad \min o_1; \mathcal{C}_1 \vdash e_1 \rightsquigarrow x_1 \qquad \min o_2; \mathcal{C}_2 \vdash e_2 \rightsquigarrow x_2}{\min o_1 + o_2; \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge x_1 + x_2 = x \vdash e_1 + e_2 \rightsquigarrow x} \textsc{Plus}$$

$$\frac{x \text{ is fresh} \qquad \min o_1; \mathcal{C}_1 \vdash e_1 \rightsquigarrow v_1 \qquad \min o_2; \mathcal{C}_2 \vdash e_2 \rightsquigarrow x_2}{\min o_1 + o_2; \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge v_1 \cdot x_2 = x \vdash e_1 \cdot e_2 \rightsquigarrow x} \textsc{TimesVal}$$

$$\frac{x \text{ is fresh} \qquad \min o_1; \mathcal{C}_1 \vdash e_1 \rightsquigarrow x_1 \qquad \min o_2; \mathcal{C}_2 \vdash e_2 \rightsquigarrow x_2}{\min o_1 + o_2; \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge [\![x_1 \cdot x_2]\!] = x \vdash e_1 \cdot e_2 \rightsquigarrow x} \textsc{TimesVar}$$

$$\frac{y \text{ is fresh} \quad \min o; \mathcal{C} \vdash e \rightsquigarrow x \quad \min o_i; \mathcal{C}_i \vdash e_i \rightsquigarrow i}{\min o + o_i; \mathcal{C} \wedge \mathcal{C}_i \wedge \mathsf{ref}\ x\ i = y \vdash \mathsf{ref}\ e\ e_i \rightsquigarrow y} \textsc{VecIxVal}$$

$$\frac{y \text{ is fresh} \quad \min o; \mathcal{C} \vdash e \rightsquigarrow x \quad \min o_i; \mathcal{C}_i \vdash e_i \rightsquigarrow x_i}{\min o + o_i; \mathcal{C} \wedge \mathcal{C}_i \wedge [\![\mathsf{ref}\ x\ x_i]\!] = y \vdash \mathsf{ref}\ e\ e_i \rightsquigarrow y} \textsc{VecIxVar}$$

$$\frac{\min o_1; \mathcal{C}_1 \vdash e_1 \rightsquigarrow x_1 \qquad \min o_2; \mathcal{C}_2 \vdash e_2 \rightsquigarrow x_2}{\min o_1 + o_2; \mathcal{C}_1 \wedge \mathcal{C}_2 \wedge x_1 \preceq x_2 \vdash \mathsf{assert}\ e_1 \preceq e_2 \rightsquigarrow 0} \textsc{Assert}$$

$$\frac{\begin{array}{c} x \text{ is fresh} \qquad \min o_1; \mathcal{C}_1 \vdash e_1 \rightsquigarrow x_1 \quad \ldots \quad \min o_n; \mathcal{C}_n \vdash e_n \rightsquigarrow x_n \\ \mathcal{C} \equiv \mathcal{C}_1 \wedge \ldots \wedge \mathcal{C}_n \wedge x = \langle x_1, \ldots, x_n \rangle \end{array}}{\min o_1 + \ldots + o_n; \mathcal{C} \vdash \langle e_1, \ldots, e_n \rangle \rightsquigarrow x} \textsc{Tuple}$$

Figure B.4: Scimitar's functional solver semantics: other ops

introduce a constraint $\mathcal{C}_x$ that governs what the overall expression leads to. This equation reads that depending on the binary variable x_g, either x_t or x_f are non-zero. In $\mathcal{C}_{tg}$ and $\mathcal{C}_{fg}$ we use the variable x_g to enable or disable the constraint sets $\mathcal{C}_t$ and $\mathcal{C}_f$ respectively. This description is meant to just give an intuition—our actual implementation is more subtle. For the exact encoding and its explanation, see Figure 3.8. We structure our constraints like this to allow backwards reasoning about x_g. For and Sum are relatively straightforward. Sometimes we can statically decide what the loop upper bound is and unroll the loop or sum completely. Otherwise the exact behavior depends on reaching the unroll limit; when reached (discussed in Section 3.3.4), which we can decide during solve time, a conditional contradiction constraint is introduced, which makes the overall problem infeasible.

Figure B.3 shows the semantics for function definition, anonymous functions, and function application. Like the unroll limit for loops, we make use of a dynamic inlining limit while inlining functions, which we depict using a special judgment form. Given some function definition ϕ, an

iteration limit n, and a conditional contradiction constraint $\mathcal{C}$, INLINE recursively replaces uses of ϕ in the expression e, yielding the completely inlined expression e'. We stop inlining when the inline limit is reached (INLINEFAIL), recognizing that from the solver's perspective this point is not reached due to recursive execution but when a certain constraint over the path condition PC is uniquely feasible. I.e., the solver decides that the variables representing the path condition can only take on the values corresponding to the situation where the inlining limit is reached. For simplicity we take the path condition for granted, and exclude its construction from these judgments.

Note that during function definition (LETREC) we pre-inline all functions up to this inlining limit, and then inline it into the body of the letrec. The expression e_f must be a function; this restriction is verified during type inference. The LAMBDA rule evaluates to function values; no environment is included, as all variables and values have been pre-substituted. Note that functions only constrain the overall problem if invoked in an application. Otherwise, their constraints are dropped. The APPLY rule checks that the value of e_1 is a function value[1] before substituting the actual into the body and applying it. The case where e_1 is not a function cannot occur, as is caught during type inference.

In Figure B.4 we give the rest of the rules. Most of these are self-explanatory, but we want to draw attention to three. In variable multiplication, TIMESVAR, one of the two positions must be an integer variable. For the complete encoding of integer multiplication, see Figure 3.7. Dynamic vector indexing, VECIXVAR, is given in the encoding in Figure 3.8. In an ASSERT rule, after analyzing the subexpressions, we directly add the constraint without the expression leading to anything.

[1]We hand wave a little here—these semantics allow for the possibility that e_1 is actually a variable, i.e., they do not guarantee a function value. Because of restructuring during solve expression processing, a function value is actually guaranteed here.

Appendix C

Formalization of $\mathcal{O}$

The optimization problem language is shown in Figure C.1. Although we give $\mathcal{O}$ a concrete syntax, allowing users to code in it directly, the intention is for users to program exclusively in Scimitar.

With the exception of primitive declarations d, $\mathcal{O}$ is a strict subset of the grammar presented in Figure 3.4. The primitive declaration $f\ (x_1, \ldots, x_n)\ (r)\ (y_1, \ldots, y_m)\ s$ introduces some primitive f with inputs x, result r and local variables y, each of some type τ. Note that all variables in a primitive must have type ascriptions, unlike in Scimitar's functional language. Although that program was already type checked, we retain these types to assist in lowering to the solver format. A primitive's body is fundamentally a conjunction of constraints $\mathcal{C}$. This form is an augmented version of a typical mathematical programming problem $\min c^T x \,\mathrm{s.t.}\, Ax \preceq b$ augmented with the explicit distinction between input, result, and local variables.

The semantics of $\mathcal{O}$ differ from Scimitar's in several important ways.

First, the looping constructs in $\mathcal{O}$ must have known finite bounds, and recursion is disallowed. Loops can not have undetermined ranges with system-wide upper bounds, as this capability is taken care of when compiling Scimitar. Additionally, no expression can contain any constraints, sequences, or for loops, which are now higher level statements s.

$$
\begin{aligned}
e &::= x \mid v \mid e\ e \mid (\ e\ , \ldots,\ e\ ) \\
&\quad \mid \ e + e \mid e \cdot e \mid \mathsf{ref}\ e\ e \mid \mathsf{sum}\ x \Leftarrow e\ \mathsf{of}\ e \\
s &::= \mathcal{C} \mid s\ ; \ldots ; s \mid \mathsf{for}\ x \Leftarrow e\ \mathsf{do}\ s \\
\mathcal{C} &::= \mathsf{assert}\ e \preceq e \\
v &::= () \mid n \mid \alpha \mid \langle\ v\ , \ldots,\ v\ \rangle \\
d &::= f\ (\ x : \tau \ldots x : \tau\ )(\ r : \tau\ )(\ y : \tau \ldots y : \tau\ )\ s \\
n &\in \mathbb{Z} \quad \alpha \in \mathbb{R} \quad f, x, r, y \in \text{variables}
\end{aligned}
$$

Figure C.1: The optimization problem language $\mathcal{O}$

Notably absent expressions are lambdas, let bindings, and conditionals. These have been completely stripped while compiling Scimitar. Lambdas are omitted because primitives in $\mathcal{O}$ must be named, which avoids the complexity of scope and closed over variables. Let bindings are undesirable for similar reasons. While conditionals might be useful at this level, we omit them to keep the design of $\mathcal{O}$ closer to a constraint-focused optimization problem representation. Finally, we can't include solver blocks, as the solver is already $\mathcal{O}$'s target. Of note, the dynamic vector indexing and McCormick envelope encodings are performed in $\mathcal{O}$ rather than Scimitar.

Appendix D

Scimitar Benchmarks

The following is a selection of Scimitar's benchmark suite. For a detailed discussion of each benchmark, see Section 3.5.

```
(letrec
  ((go (lambda (prev upper)
         (if (= prev upper)
           upper
           (go upper
               (optimum-ref in
                 (maximize (: result real)
                   (define in (symbolic))
                   (define result (symbolic))
                   (letrec
                     ((foo (lambda x
                             (if (> x 3)
                               (if (< x upper)
                                 (+ x 2)
                                 3)
                               (* 2 x)))))
                     (assert
                       (= result
                          (foo in)))))))))))
  (go (: 8 (ival 0 10))
      (: 7 (ival 0 10))))
```

Figure D.1: The *bounce* benchmark

```
(optimum-ref new-total
  (minimize new-memory-usage
    (define new-total (symbolic))
    (define new-memory-usage (symbolic))
    (define load-balance (symbolic))
    (assert
      (= (: new-memory-usage nneg)
         (sum ([i (range (racket (bucket-count)))])
           (* (vec-ref (racket bucket-sizes) i)
              (vec-ref (: new-total (nnegty (bucket-count))) i)))))
    (assert
      (= (racket (floor (/ (bucket-count) 2)))
         (sum ([i (range (racket (bucket-count)))])
           (vec-ref (: load-balance (bitty (bucket-count))) i))))
    (for ([i (range (racket (bucket-count)))])
      (if (> (vec-ref (racket pending) i) 0)
        ;; the logic of this constraint is that we want to have a few
        ;; slack slots because this might be a busy bucket
        (assert
          (>= (vec-ref (: new-total (nnegty (bucket-count))) i)
              (+ (vec-ref (racket new-used) i)
                 (* (racket (grow-factor))
                    (vec-ref (racket pending) i)))))
        (begin
          ;; prefer removing from buckets that have been unused
          ;; overall as compared to the size the last time.
          (assert
            (>= (vec-ref (: new-total (nnegty (bucket-count))) i)
                (* (racket (shrink-factor)) (vec-ref (racket total) i))))
          (assert
            (>= (vec-ref (: new-total (nnegty (bucket-count))) i)
                (if (> (vec-ref (racket new-allocs) i) 0)
                  (+ (vec-ref (racket new-used) i)
                     (* (vec-ref (: load-balance (bitty (bucket-count))) i)
                        (vec-ref (racket new-allocs) i)))
                  (vec-ref (racket new-used) i)))))))))
```

Figure D.2: The optimization core of the *malloc* benchmark

```
(product tacos 1 1.00)
(truck foodtruck 0.1 1000)
(city downtown 100 10000 (tacos 11000))
(city uptown   200 2000  (tacos 1000))
(road downtown uptown 10 9)
```

Figure D.3: The *logistics-s* benchmark

```
(product apples 3 0.50)
(product banana 2 0.25)
(product orange 4 0.75)
(product grapes 2 1.25)

(city smallville 1000 14000
      (apples 400)
      (orange 600)  (grapes 50))
(city mediumville 1200 22000
      (apples 1750) (banana 500)
      (orange 3000) (grapes 2000))
(city bigville 900 40000
      (apples 4000) (banana 5500)
      (orange 5500) (grapes 4000))
(city nowheresville 100 15000
      (apples 50)   (orange 40))

(road smallville mediumville 40 20)
(road smallville bigville 60 10)
(road mediumville bigville 30 40)
(road nowheresville smallville 100 0)

(truck mega-rig     0.33 2000)
(truck the-monster  0.32 2000)
(truck big-bertha   0.30 1800)
(truck keep-truckin 0.25 1400)
(truck lonely-road  0.20 600)
(truck dusty-trail  0.19 600)
(truck little-guy   0.16 300)
(truck putt-putt    0.15 300)
```

Figure D.4: The *logistics-h* benchmark

```
(optimum-ref (rec-count rec-scheduled rec-attendance)
 (minimize rec-count
  (define rec-count (symbolic))
  (define rec-scheduled (symbolic))
  (define rec-attendance (symbolic))
  (define student-attendance (symbolic))
  (for ([i (range (: rec-count (ival 0 (time-slots))))])
   (begin
    ;; attendance for the recitation; time slot must be between the minimum and the capacity
    (assert
     (<= (racket (rec-minimum-attendance))
         (vec-ref (: rec-attendance (vecty (ival 0 (rec-capacity)) `(,(time-slots)))) i)))
    (assert
     (>= (racket (rec-capacity))
         (vec-ref (: rec-attendance (vecty (ival 0 (rec-capacity)) `(,(time-slots)))) i)))
    ;; different rec-scheduled entries are not the same; unscheduled ones should have zero attendees
    (if (< (+ 1 i) rec-count)
     (assert
      (<= (+ 1 (vec-ref (: rec-scheduled (vecty (ival 0 (sub1 (time-slots))) `(,(+ 1 (time-slots))))) i))
          (vec-ref (: rec-scheduled (vecty (ival 0 (sub1 (time-slots))) `(,(+ 1 (time-slots))))) (+ 1 i))))
     '())
    (if (> (+ 1 i) rec-count)
     (assert (= 0 (vec-ref (: rec-scheduled (vecty (ival 0 (sub1 (time-slots))) `(,(+ 1 (time-slots))))) i)))
     '())
    ;; the sum of students available for the selected scheduled recitation must exceeds the attendance
    (assert
     (>= (sum ([j (range (racket (student-count)))])
          (vec-ref (racket (student-availability))
           `(,j ,(vec-ref (: rec-scheduled (vecty (ival 0 (sub1 (time-slots))) `(,(+ 1 (time-slots))))) i))))
         (vec-ref (: rec-attendance (vecty (ival 0 (rec-capacity)) `(,(time-slots)))) i)))
    ;; the sum of all students in attendence must equal the attendance for that recitation
    (assert
     (= (sum ([j (range (racket (student-count)))])
         (vec-ref (: student-attendance (bitty (student-count) (time-slots)))
          `(,j ,(vec-ref (: rec-scheduled (vecty (ival 0 (sub1 (time-slots))) `(,(+ 1 (time-slots))))) i))))
        (vec-ref (: rec-attendance (vecty (ival 0 (rec-capacity)) `(,(time-slots)))) i)))))
  ;; the number of recitations must exceed the number of time slots that have a non-zero attendance
  (assert
   (>= rec-count
       (sum ([i (range (racket (time-slots)))])
        (< 0 (sum ([j (range (racket (student-count)))])
              (vec-ref (: student-attendance (bitty (student-count) (time-slots))) `(,j ,i)))))))
  ;; This excludes the possibility that the student isn't available and did attend
  (for ([i (range (racket (time-slots)))])
   (for ([j (range (racket (student-count)))])
    (assert
     (= 1 (let ((a (- 1 (vec-ref (: student-attendance (bitty (student-count) (time-slots))) `(,j ,i))))
                (b (vec-ref (racket (student-availability)) `(,j ,i))))
            (- (+ a b) (* a b)))))))
  ;; the sum of all student attendances must be at least the attendance requirement * student count
  (assert
   (>= (sum ([i (range (racket (time-slots)))])
        (sum ([j (range (racket (student-count)))])
         (vec-ref (: student-attendance (bitty (student-count) (time-slots))) `(,j ,i))))
       (racket (* (student-attendance-requirement) (student-count)))))))
```

Figure D.5: The optimization core of the *recitation* benchmark

Bibliography

[ABC+20] Alessandro Abate, Iury Bessa, Lucas Cordeiro, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Automated formal synthesis of provably safe digital controllers for continuous plants. *Acta Informatica*, 57(1):223–244, 2020. URL: https://pmc.ncbi.nlm.nih.gov/articles/PMC7056743/, doi:10.1007/s00236-019-00359-1.

[ADG16] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 789–801, New York, NY, USA, January 2016. Association for Computing Machinery. doi:10.1145/2837614.2837628.

[Ant24] Anthropic. Introducing the Model Context Protocol, 2024. URL: https://www.anthropic.com/news/model-context-protocol.

[AP20] Martín Abadi and Gordon D. Plotkin. A simple differentiable programming language. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, January 2020. URL: https://dl.acm.org/doi/10.1145/3371106, doi:10.1145/3371106.

[ASMS19] Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. Augmented example-based synthesis using relational perturbation properties. *Proceedings of the ACM on Programming Languages*, 4(POPL):56:1–56:24, December 2019. doi:10.1145/3371124.

[BCG+10a] Rastislav Bodík, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 339–352, New York, NY, USA, 2010. ACM. doi:10.1145/1706299.1706339.

[BCG+10b] Rastislav Bodík, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 339–352, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1706299.1706339.

[BFRSL21] Nate F. F. Bragg, Jeffrey S. Foster, Cody Roux, and Armando Solar-Lezama. Program sketching by automatically generating mocks from tests. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*, page 808–831, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-81685-8_38.

[BFZ24a] Nate F. F. Bragg, Jeffrey S. Foster, and Philip Zucker. Scimitar: Functional programs as optimization problems. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '24, page 96–112, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3689492.3690051.

[BFZ24b] Nate F. F. Bragg, Jeffrey S. Foster, and Philip Zucker. Scimitar: Onward! 2024 artifact, 2024. URL: https://zenodo.org/doi/10.5281/zenodo.13625532, doi:10.5281/ZENODO.13625532.

[Bjø10] Nikolaj Bjørner. Linear Quantifier Elimination as an Abstract Decision Procedure. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, Lecture Notes in Computer Science, pages 316–330, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-14203-1_27.

[Bjo22] Nikolaj Bjorner. Advanced topics, 2022. Accessed: 2023-04-12. URL: https://microsoft.github.io/z3guide/docs/optimization/advancedtopics/.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977. doi:10.1145/359842.359859.

[BM07] Aaron R. Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer, Berlin, 2007. OCLC: 255687662.

[BOW+23] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-Down Synthesis for Library Learning. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571234.

[BPRS18] Atilim Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018. URL: http://jmlr.org/papers/v18/17-468.html.

[Bra12] Benjamin Braun. Compiling computations to constraints for verified computation. *UT Austin Honors Thesis HR-12-10*, 2012.

[BRNR17] Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a Differentiable Forth Interpreter. In *International Conference on Machine Learning*, pages 547–556, July 2017. ISSN: 2640-3498 Section: Machine Learning. URL: http://proceedings.mlr.press/v70/bosnjak17a.html.

[BTGC16] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 775–788, New York, NY, USA, January 2016. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2837614.2837666, doi:10.1145/2837614.2837666.

[CP34] C. J. Clopper and E. S. Pearson. The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial. *Biometrika*, 26(4):404–413, December 1934. Publisher: Oxford Academic. doi:10.1093/biomet/26.4.404.

[CSL10a] Swarat Chaudhuri and Armando Solar-Lezama. Smooth interpretation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 279–291, Toronto, Ontario, Canada, June 2010. Association for Computing Machinery. doi:10.1145/1806596.1806629.

[CSL10b] Swarat Chaudhuri and Armando Solar-Lezama. Smooth interpretation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 279–291, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1806596.1806629.

[CSL12] Swarat Chaudhuri and Armando Solar-Lezama. Euler: A System for Numerical Optimization of Programs. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification*, pages 732–737, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-31424-7_57.

[CSM12] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Using program synthesis for social recommendations. In Xue-wen Chen, Guy Lebanon, Haixun Wang, and Mohammed J. Zaki, editors, *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pages 1732–1736, Hawaii, USA, 2012. ACM. URL: http://dl.acm.org/citation.cfm?id=2396761, doi:10.1145/2396761.2398507.

[DAI] DAIR.AI. Prompt Engineering Guide — Nextra. [Online; accessed 20-December-2025]. URL: https://www.promptingguide.ai/.

[DB16] Steven Diamond and Stephen Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016. URL: http://jmlr.org/papers/v17/15-408.html.

[DHL17] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM review*, 59(2):295–320, 2017. doi:10.1137/15M1020575.

[Dom18] John Dombrowski. Mccormick envelopes, Nov 2018. URL: https://doi.org/10.21985/N29T8M.

[ENP+19] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, Execute, Assess: Program Synthesis with a REPL. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 9169–9178. Curran Associates, Inc., 2019. URL: http://papers.nips.cc/paper/9116-write-execute-assess-program-synthesis-with-a-repl.pdf.

[ERSLT18] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to Infer Graphics Programs from Hand-Drawn Images. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 6059–6068. Curran Associates, Inc., 2018.

[EWN+21] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 835–850, New York, NY, USA, June 2021. Association for Computing Machinery. doi:10.1145/3453483.3454080.

[FCD15] 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 '15, pages 229–239, New York, NY, USA, June 2015. Association for Computing Machinery. doi:10.1145/2737924.2737977.

[FGK87] Robert Fourer, David M Gay, and Brian W Kernighan. *AMPL: A mathematical programming language*. AT & T Bell Laboratories Murray Hill, NJ, 1987.

[FGO20] Mattia Fazzini, Alessandra Gorla, and Alessandro Orso. A Framework for Automated Test Mocking of Mobile Apps. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), NIER track*, pages 1204–1208, September 2020. ISSN: 2643-1572.

[Fil03] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Technical report, Research Report 1366, LRI, Université Paris Sud, 2003. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.3200&q=A%20Certified%20Multi-prover%20Verification%20Condition%20Generator.

[FMW+17] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-Based Synthesis for Complex APIs. *ACM SIGPLAN Notices*, 52(1):599–612, January 2017. doi:10.1145/3093333.3009851.

[GAM22] GAMS Development Corporation. Generalized disjunctive programs (gdps), 2022. URL: https://www.gams.com/latest/docs/UG_EMP_DisjunctiveProgramming.html.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, October 2007. doi:10.1145/1297105.1297033.

[GJJ+20] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, January 2020. URL: https://dl.acm.org/doi/10.1145/3371080, doi:10.1145/3371080.

[GJL+25] Alex Gu, Naman Jain, Wen-Ding Li, Manish Shetty, Yijia Shao, Ziyang Li, Diyi Yang, Kevin Ellis, Koushik Sen, and Armando Solar-Lezama. Challenges and Paths Towards AI for Software Engineering, March 2025. arXiv:2503.22625 [cs]. URL: http://arxiv.org/abs/2503.22625, doi:10.48550/arXiv.2503.22625.

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 62–73, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993506.

[Goo24] Google DeepMind. AI achieves silver-medal standard solving International Mathematical Olympiad problems, May 2024. URL: https://deepmind.google/blog/ai-solves-imo-problems-at-silver-medal-level/.

[Gur23] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL: https://www.gurobi.com.

[HCDR20a] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1128–1142, London UK, June 2020. ACM. URL: https://dl.acm.org/doi/10.1145/3385412.3385979, doi:10.1145/3385412.3385979.

[HCDR20b] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 1128–1142, New York, NY, USA, June 2020. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3385412.3385979, doi:10.1145/3385412.3385979.

[HP13] Laurent Hascoët and Valérie Pascual. The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3), may 2013. doi:10.1145/2450153.2450158.

[HQSW20a] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 1159–1174, New York, NY, USA, June 2020. Association for Computing Machinery. doi:10.1145/3385412.3386027.

[HQSW20b] Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 1159–1174, New York, NY, USA, June 2020. Association for Computing Machinery. doi:10.1145/3385412.3386027.

[HV19] Joey Huchette and Juan Pablo Vielma. A combinatorial approach for small and strong formulations of disjunctive constraints. *Mathematics of Operations Research*, 44(3):793–820, 2019. doi:10.1287/moor.2018.0946.

[HV22] Joey Huchette and Juan Pablo Vielma. Nonconvex piecewise linear functions: Advanced formulations and simple modeling tools. *Operations Research*, 2022. doi:10.1287/opre.2019.1973.

[HZZK19] Jinru Hua, Yushan Zhang, Yuqun Zhang, and Sarfraz Khurshid. Edsketch: Execution-driven sketching for java. *Int. J. Softw. Tools Technol. Transf.*, 21(3):249–265, June 2019. doi:10.1007/s10009-019-00512-8.

[IDS20] Anastasiia Izycheva, Eva Darulova, and Helmut Seidl. Counterexample- and Simulation-Guided Floating-Point Loop Invariant Synthesis. In David Pichardie and Mihaela Sighireanu, editors, *Static Analysis*, pages 156–177, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-65474-0_8.

[IGKSL18] Jeevana Priya Inala, Sicun Gao, Soonho Kong, and Armando Solar-Lezama. REAS: Combining Numerical Optimization with SAT Solving, February 2018. arXiv:1802.04408 [cs]. URL: http://arxiv.org/abs/1802.04408, doi:10.48550/arXiv.1802.04408.

[IPP+21] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. Cyclic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 944–959, New York, NY, USA, June 2021. Association for Computing Machinery. doi:10.1145/3453483.3454087.

[IPQ+17] Jeevana Priya Inala, Nadia Polikarpova, Xiaokang Qiu, Benjamin S. Lerner, and Armando Solar-Lezama. Synthesis of Recursive ADT Transformations from Reusable Templates. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS*

2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I, volume 10205 of *Lecture Notes in Computer Science*, pages 247–263, 2017. doi:10.1007/978-3-662-54577-5_14.

[ISSL16] Jeevana Priya Inala, Rohit Singh, and Armando Solar-Lezama. Synthesis of Domain Specific CNF Encoders for Bit-Vector Solvers. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 302–320. Springer, 2016. doi:10.1007/978-3-319-40970-2_19.

[JQFD+16] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. Synthesizing framework models for symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 156–167, New York, NY, USA, May 2016. Association for Computing Machinery. doi:10.1145/2884781.2884856.

[JQSLF15] Jinseong Jeon, Xiaokang Qiu, Armando Solar-Lezama, and Jeffrey S. Foster. JSketch: Sketching for Java. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE), Tool Demo Track*, pages 934–937, Bergamo, Italy, September 2015. ACM.

[JS17] Susmit Jha and Sanjit A. Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54(7):693–726, November 2017. doi:10.1007/s00236-017-0294-5.

[KCISL16] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 711–726, New York, NY, USA, June 2016. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/2908080.2908117, doi:10.1145/2908080.2908117.

[KdONML25] Ranim Khojah, Francisco Gomes de Oliveira Neto, Mazen Mohamad, and Philipp Leitner. The Impact of Prompt Programming on Function-Level Code Generation. *IEEE Transactions on Software Engineering*, 51(8):2381–2395, August 2025. URL: https://ieeexplore.ieee.org/abstract/document/11077752, doi:10.1109/TSE.2025.3587794.

[KHDR21] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages*, 5(POPL):30:1–30:32, January 2021. doi:10.1145/3434311.

[KMM+20] Paul Krogmeier, Umang Mathur, Adithya Murali, P. Madhusudan, and Mahesh Viswanathan. Decidable Synthesis of Programs with Uninterpreted Functions. In

Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 634–657, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-53291-8_32.

[KMPS10] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 316–329, New York, NY, USA, 2010. ACM. doi:10.1145/1806596.1806632.

[KS09] Oleg Kiselyov and Chung-chieh Shan. Embedded probabilistic programming. In *IFIP Working Conference on Domain-Specific Languages*, pages 360–384. Springer, 2009. doi:10.1007/978-3-642-03034-5_17.

[KSLG18] Soonho Kong, Armando Solar-Lezama, and Sicun Gao. Delta-Decision Procedures for Exists-Forall Problems over the Reals. *arXiv*, 2018. URL: https://dspace.mit.edu/handle/1721.1/137908.

[KWPH19a] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2019*, pages 253–268, Phoenix, AZ, USA, 2019. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3314221.3314602, doi:10.1145/3314221.3314602.

[KWPH19b] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 253–268, New York, NY, USA, June 2019. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/3314221.3314602, doi:10.1145/3314221.3314602.

[LCC+22] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, December 2022. URL: https://www.science.org/doi/abs/10.1126/science.abq1158, doi:10.1126/science.abq1158.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-17511-4_20.

[Leu17] Alan Leung. *Constructing Parsers by Example via Interactive Program Synthesis*. PhD thesis, UC San Diego, 2017. URL: https://escholarship.org/uc/item/5m96s9r5.

[LGA+23] Rémi Leblond, Felix Gimeno, Florent Altché, Alaa Saade, Anton Ruddock, Corentin Tallec, George Powell, Jean-Bastien Grill, Maciej Mikula, Matthias Lochbrunner, and others. Alphacode 2 Technical Report. 2023. URL: https://storage.googleapis.com/deepmind-media/AlphaCode2/AlphaCode2_Tech_Report.pdf.

[LMN16] Christof Löding, P. Madhusudan, and Daniel Neider. Abstract Learning Frameworks for Synthesis, May 2016. arXiv:1507.05612 [cs]. URL: http://arxiv.org/abs/1507.05612, doi:10.48550/arXiv.1507.05612.

[LSO16] Mina Lee, Sunbeom So, and Hakjoo Oh. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, pages 70–80, New York, NY, USA, October 2016. Association for Computing Machinery. URL: http://doi.org/10.1145/2993236.2993244, doi:10.1145/2993236.2993244.

[Man12] Oleksandr Manzyuk. A simply typed λ-calculus of forward automatic differentiation. *Electronic Notes in Theoretical Computer Science*, 286:257–272, 2012. URL: https://www.sciencedirect.com/science/article/pii/S1571066112000473, doi:10.1016/j.entcs.2012.08.017.

[MFP+17] Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. Synthesizing bijective lenses. *Proceedings of the ACM on Programming Languages*, 2(POPL):1:1–1:30, December 2017. doi:10.1145/3158089.

[MMM+25] Aditi Mavalankar, Hassan Mansoor, Zita Marinho, Mariia Samsikova, and Tom Schaul. AuPair: Golden Example Pairs for Code Repair. June 2025. URL: https://openreview.net/forum?id=GmqZ3WvkeV.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, 2004.

[MRX+19] Benjamin Mariano, Josh Reese, Siyuan Xu, ThanhVu Nguyen, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. Program synthesis with algebraic library specifications. *Proc. ACM Program. Lang.*, 3(OOPSLA):132:1–132:25, October 2019. doi:10.1145/3360558.

[MW79] Z. Manna and R. Waldinger. Synthesis: Dreams → Programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, July 1979. Conference Name: IEEE Transactions on Software Engineering. URL: http://ieeexplore.ieee.org/document/1702636/, doi:10.1109/TSE.1979.234198.

[NGM+18] Daniel Neider, Pranav Garg, P. Madhusudan, Shambwaditya Saha, and Daejun Park. Invariant Synthesis for Incomplete Verification Engines. In Dirk Beyer and

Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 232–250, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-89960-2_13.

[NJR22] Thanh Son Nguyen, Ben Jones, and Zvonimir Rakamarić. Synthesis of Rigorous Floating-Point Predicates. In Owolabi Legunsen and Grigore Rosu, editors, *Model Checking Software*, pages 44–60, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-15077-7_3.

[NSB+07] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*, pages 529–543. Springer, 2007. doi:10.1007/978-3-540-74970-7_38.

[NSM16] Daniel Neider, Shambwaditya Saha, and P. Madhusudan. Synthesizing Piece-Wise Functions by Learning Classifiers. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 186–203, Berlin, Heidelberg, 2016. Springer. doi:10.1007/978-3-662-49674-9_11.

[NWKF17] ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. Connecting Program Synthesis and Reachability: Automatic Program Repair Using Test-Input Generation. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 301–318, Berlin, Heidelberg, 2017. Springer. doi:10.1007/978-3-662-54577-5_17.

[OBW20] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. Cryptology ePrint Archive, Paper 2020/1586, 2020. URL: https://eprint.iacr.org/2020/1586.

[Par88] P.M. Pardalos. Linear complementarity problems solvable by integer programming. *Optimization*, 19(4):467–474, 1988. doi:10.1080/02331938808843365.

[PKSL16] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016*, pages 522–538, Santa Barbara, CA, USA, 2016. ACM Press. doi:10.1145/2908080.2908093.

[PS08] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):1–36, 2008. doi:10.1145/1330017.1330018.

[RDK+15] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, volume 9207, pages 198–216. Springer International Publishing, Cham, 2015. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-319-21668-3_12, doi:10.1007/978-3-319-21668-3_12.

[RJG+12] Juan P Ruiz, Jan-H Jagla, Ignacio E Grossmann, Alex Meeraus, and Aldo Vecchietti. Generalized disjunctive programming: Solution strategies. In *Algebraic Modeling Systems*, pages 57–75. Springer, 2012. doi:10.1007/978-3-642-23592-4_4.

[RPBN+24] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, January 2024. URL: https://www.nature.com/articles/s41586-023-06924-6, doi:10.1038/s41586-023-06924-6.

[SA19] Calvin Smith and Aws Albarghouthi. Program synthesis with equivalence reduction. In Constantin Enea and Ruzica Piskac, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 24–47, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-11245-5_2.

[SAPE05] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 114–123, New York, NY, USA, November 2005. Association for Computing Machinery. doi:10.1145/1101908.1101927.

[SBA+25] E. G. Santana, Gabriel Benjamin, Melissa Araujo, Harrison Santos, David Freitas, Eduardo Almeida, Paulo Anselmo da M. S. Neto, Jiawei Li, Jina Chun, and Iftekhar Ahmed. Which Prompting Technique Should I Use? An Empirical Investigation of Prompting Techniques for Software Engineering Tasks, June 2025. arXiv:2506.05614 [cs]. URL: http://arxiv.org/abs/2506.05614, doi:10.48550/arXiv.2506.05614.

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 313–326, New York, NY, USA, January 2010. Association for Computing Machinery. doi:10.1145/1706299.1706337.

[SKR19] Malavika Samak, Deokhwan Kim, and Martin C. Rinard. Synthesizing replacement classes. *Proceedings of the ACM on Programming Languages*, 4(POPL):52:1–52:33, December 2019. doi:10.1145/3371120.

[SL08] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html.

[SL20] Armando Solar-Lezama. The Sketch Programmers Manual. Technical report, MIT, February 2020.

[SLAT+07a] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 167–178, San Diego, California, USA, June 2007. Association for Computing Machinery. doi:10.1145/1250734.1250754.

[SLAT+07b] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. Sketching stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 167–178, San Diego, California, USA, June 2007. Association for Computing Machinery. doi:10.1145/1250734.1250754.

[SLB25] Armando Solar-Lezama and Nate F.F. Bragg. Discussion of Floating Point Encodings Within Sketch and the State of Program Synthesis using Large Language Models, June 2025.

[Sol09] Armando Solar-Lezama. The sketching approach to program synthesis. In Zhenjiang Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 4–13. Springer, 2009. doi:10.1007/978-3-642-10672-9_3.

[SRHN19] Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. Synthesizing Datalog Programs using Numerical Relaxation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 6117–6124, Macao, China, August 2019. International Joint Conferences on Artificial Intelligence Organization. URL: https://www.ijcai.org/proceedings/2019/847, doi:10.24963/ijcai.2019/847.

[SSL19] Kensen Shi, Jacob Steinhardt, and Percy Liang. FrAngel: component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages*, 3(POPL):73:1–73:29, January 2019. doi:10.1145/3290386.

[SSX+14] Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. Modular synthesis of sketches using models. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 395–414. Springer, 2014. doi:10.1007/978-3-642-54013-4_22.

[STB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006. doi:10.1145/1168857.1168907.

[TB13] Emina Torlak and Rastislav Bodík. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software - Onward! '13*, pages 135–152, Indianapolis, Indiana, USA, 2013. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=2509578.2509586, doi:10.1145/2509578.2509586.

[The20] The MathWorks, Inc. intlinprog, Copyright 2013-2020.

[VGND+20] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. Synthesizing JIT Compilers for In-Kernel DSLs. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, volume 12225, pages 564–586. Springer International Publishing, Cham, 2020. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-030-53291-8_29.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 6000–6010, Red Hook, NY, USA, December 2017. Curran Associates Inc. URL: https://dl.acm.org/doi/abs/10.5555/3295222.3295349, doi:10.5555/3295222.3295349.

[WADM18] Xinyu Wang, Greg Anderson, Isil Dillig, and K. L. McMillan. Learning Abstractions for Program Synthesis. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, volume 10981, pages 407–426. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/978-3-319-96145-3_22, doi:10.1007/978-3-319-96145-3_22.

[WDS18] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, January 2018. URL: https://dl.acm.org/doi/10.1145/3158151, doi:10.1145/3158151.

[WGW23] Jingbo Wang, Aarti Gupta, and Chao Wang. Synthesizing MILP Constraints for Efficient and Robust Optimization. *Proc. ACM Program. Lang.*, 7(PLDI):184:1896–184:1919, June 2023. URL: https://dl.acm.org/doi/10.1145/3591298, doi:10.1145/3591298.

[Wik24] Wikipedia contributors. Big m method — Wikipedia, the free encyclopedia, 2024. [Online; accessed 28-August-2024]. URL: https://en.wikipedia.org/w/index.php?title=Big_M_method&oldid=1227030411.

[Wik25] Wikipedia contributors. Square root algorithms — Wikipedia, The Free Encyclopedia, 2025. [Online; accessed 20-December-2025]. URL: https://en.wikipedia.org/w/index.php?title=Square_root_algorithms&oldid=1326623125.

[WK20] Thomas Welsch and Vitaliy Kurlin. Synthesis through unification genetic programming. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 1029–1036, Cancún Mexico, June 2020. ACM. URL: https://dl.acm.org/doi/10.1145/3377930.3390208, doi:10.1145/3377930.3390208.

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. doi:10.1017/S1471068411000494.

[WWD18] Yuepeng Wang, Xinyu Wang, and Isil Dillig. Relational program synthesis. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–27, October 2018. URL: https://dl.acm.org/doi/10.1145/3276525, doi:10.1145/3276525.

[YKDC16] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing Transformations on Hierarchically Structured Data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 508–521, New York, NY, USA, June 2016. Association for Computing Machinery. doi:10.1145/2908080.2908088.

[YSW+25] Sixiang Ye, Zeyu Sun, Guoqing Wang, Liwei Guo, Qingyuan Liang, Zheng Li, and Yong Liu. Prompt Alchemy: Automatic Prompt Refinement for Enhancing Code Generation. *IEEE Transactions on Software Engineering*, 51(9):2472–2493, September 2025. URL: https://ieeexplore.ieee.org/document/11082010, doi:10.1109/TSE.2025.3589634.

[YWDD17] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. SQLizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):63:1–63:26, October 2017. doi:10.1145/3133887.