Automated Approaches for Program Verification and Repair

William Triest Hallahan
Yale University Graduate School of Arts and Sciences, billthallahan@gmail.com

Follow this and additional works at: https://elischolar.library.yale.edu/gsas_dissertations

Recommended Citation
Hallahan, William Triest, "Automated Approaches for Program Verification and Repair" (2022). Yale Graduate School of Arts and Sciences Dissertations. 602.
https://elischolar.library.yale.edu/gsas_dissertations/602

This Dissertation is brought to you for free and open access by EliScholar – A Digital Platform for Scholarly Publishing at Yale. It has been accepted for inclusion in Yale Graduate School of Arts and Sciences Dissertations by an authorized administrator of EliScholar – A Digital Platform for Scholarly Publishing at Yale. For more information, please contact elischolar@yale.edu.
Formal methods techniques, such as verification, analysis, and synthesis, allow programmers to prove properties of their programs, or automatically derive programs from specifications. Making such techniques usable requires care: they must provide useful debugging information, be scalable, and enable automation. This dissertation presents automated analysis and synthesis techniques to ease the debugging of modular verification systems and allow easy access to constraint solvers from functional code. Further, it introduces machine learning based techniques to improve the scalability of off-the-shelf syntax-guided synthesis solvers and techniques to reduce the burden of network administrators writing and analyzing firewalls.

We describe the design and implementation of a symbolic execution engine, G2, for non-strict functional languages such as Haskell. We extend G2 to both debug and automate the process of modular verification, and give Haskell programmers easy access to constraints solvers via a library named G2Q.

Modular verifiers, such as LiquidHaskell, Dafny, and ESC/Java, allow programmers to write and prove specifications of their code. When a modular verifier fails to verify a program, it is not necessarily because of an actual bug in the program. This is because when verifying a function $f$, modular verifiers consider only the specification of a called function $g$, not the actual definition of $g$. Thus, a modular verifier may fail to prove a true specification of $f$ if the specification of $g$ is too weak. We present a technique, counterfactual symbolic execution, to aid in the debugging of
modular verification failures. The approach uses symbolic execution to find concrete counterexamples, in the case of an actual inconsistency between a program and a specification; and abstract counterexamples, in the case that a function specification is too weak. Further, a counterexample-guided inductive synthesis (CEGIS) loop based technique is introduced to fully automate the process of modular verification, by using found counterexamples to automatically infer needed function specifications. The counterfactual symbolic execution and automated specification inference techniques are implemented in G2, and evaluated on existing LiquidHaskell errors and programs.

We also leveraged G2 to build a library, G2Q, which allows writing constraint solving problems directly as Haskell code. Users of G2Q can embed specially marked Haskell constraints (Boolean expressions) into their normal Haskell code, while marking some of the variables in the constraint as symbolic. Then, at runtime, G2Q automatically derives values for the symbolic variables that satisfy the constraint, and returns those values to the outside code. Unlike other constraint solving solutions, such as directly calling an SMT solver, G2Q uses symbolic execution to unroll recursive function definitions, and guarantees that the use of G2Q constraints will preserve type correctness.

We further consider the problem of synthesizing functions via a class of tools known as syntax-guided synthesis (SyGuS) solvers. We introduce a machine learning based technique to preprocess SyGuS problems, and reduce the space that the solver must search for a solution in. We demonstrate that the technique speeds up an existing SyGuS solver, CVC4, on a set of SyGuS solver benchmarks.

Finally, we describe techniques to ease analysis and repair of firewalls. Firewalls
are widely deployed to manage network security. However, firewall systems provide only a primitive interface, in which the specification is given as an ordered list of rules. This makes it hard to manually track and maintain the behavior of a firewall.

We introduce a formal semantics for iptables firewall rules via a translation to first-order logic with uninterpreted functions and linear integer arithmetic, which allows encoding of firewalls into a decidable logic. We then describe techniques to automate the analysis and repair of firewalls using SMT solvers, based on user provided specifications of the desired behavior. We evaluate this approach with real world case studies collected from StackOverflow users.
Automated Approaches for Program Verification and Repair

A Dissertation
Presented to the Faculty of the Graduate School
Of
Yale University
In Candidacy for the Degree of
Doctor of Philosophy

By
William T. Hallahan

Dissertation Director: Ruzica Piskac

May 2022
Copyright © 2022 by William T. Hallahan
All rights reserved.
I’d like to begin by thanking my advisor, Ruzica Piskac. Ruzica has been incredibly supportive from the beginning of my PhD (before she was even my advisor) to the end. I appreciate all the guidance she has given me and how much she has taught me about conducting research, writing papers, and giving presentations. Ruzica constantly looks out for what is best for her students, and encourages us to extend ourselves to our full potential. I also very much appreciated Ruzica’s sense of humor, which made long hours, late into the night, working on papers much more enjoyable.

I appreciate Zhong Shao, Abhishek Bhattacharjee, and Nate Foster for serving on my dissertation committee, and am grateful for their feedback on this dissertation. I have enjoyed my occasional interactions with Zhong and Abhishek, at Yale Computer Science events or when I have run into them in the hallway.

In addition to serving on my dissertation committee, Nate Foster also served as my mentor when I was an intern at Barefoot Networks after my second year of graduate school, and collaborated with me on two papers. It was great to get some early insight on research in industry, and to have Nate guide me through a slightly different research process than I had previously experienced at Yale.

I have also been fortunate to collaborate with Ranjit Jhala twice during my PhD. Ranjit’s enthusiasm and cheerfulness is infectious, and I have enjoyed working with him and learning from him. In particular, I feel like I have learned a lot from Ranjit about how to structure a paper, and I very much appreciate all the support he’s given me.

I was an intern at Galois during the Summer of 2018. I enjoyed working with all the Galwegians I met there, and especially appreciate the support I got from Tristan Ravitch, Erin Chapman, Kevin Quick, and Rob Dockins. It was a terrific opportunity to get more experience in industry, while still almost feeling like I was in academia.

When I first started working with Ruzica, her group consisted of Ennan Zhai and Mark Santolucito. I very much appreciate Ennan collaborating with me on my first project and paper as a PhD student. Along with Ruzica, Ennan was instrumental in helping guide me through the research and writing process. Mark is now a good friend, and was a great source of help, guidance, and inspiration while we shared an office at Yale.

During my time at Yale, Ruzica’s group has grown to include Timos Antonopoulos, Julien Lepiller, Jialu Zhang, Ning Luo, Sam Judson, Ferhat Erata, John Kolesar, Matt
Elacqua, and Ben Chaimberg. I very much appreciate the support from the whole group—ranging from feedback on presentations to late night dinners on submission nights.

During my PhD I was fortunate to collaborate with a number of other graduate students, undergraduates, and even high school students. I would like to thank Eric Campbell, who led the Avenir project to a successful NSDI paper, Anton Xue, who as an undergraduate played a key role in designing and building G2, and Max Levatich, who as an undergraduate spent a summer with us exploring synthesis techniques. I appreciate all the hard work and dedication of Kairo Morton and Elven Shum, two high school students who visited Yale in the Summer of 2019, and did excellent work on the Grammar Reduction Tool. I would also like to extend my thanks to the numerous other collaborators I have had during my PhD, including (among others) Robert Soulé, Jed Liu, Cole Schlesinger, Călin Cașcaval, Hossein Hojjat, and Priya Srikumar. I am very grateful for the excellent advice and help that Lin Zhong gave me when I was beginning my job search.

I'd like to thank a number of professors at my undergraduate college, the College of the Holy Cross. Professor Cristina Ballantine introduced me to research and was my co-author on my first ever publication. Without her guidance, support, and encouragement, I might have never discovered how much fun research is. I am also thankful to Professor King, Professor Walsh, and Professor Hwang, all of whom gave me great advice—both while I was a student at Holy Cross, and after I had left.

My friends have been a constant source of support during my PhD, and I have enjoyed all of our trivia outings, Yale Cab viewings, and game nights. Finally, I would like to thank my family. I would especially like to thank my Mom and Dad, who have been incredibly encouraging, enthusiastic, and caring during my PhD.
## Contents

### 1 Introduction
- 1.1 Symbolic Execution .................................................. 1
  - 1.1.1 Constraint Solving ............................................. 2
  - 1.1.2 Modular Verification ........................................... 2
- 1.2 Syntax-Guided Synthesis Grammar Reduction ......................... 3
- 1.3 Firewall Analysis and Repair ....................................... 4
- 1.4 Contributions .......................................................... 4
- 1.5 Remarks ................................................................. 6

### 2 Non-strict Symbolic Execution
- 2.1 Introduction ............................................................ 7
- 2.2 Overview ............................................................... 8
  - 2.2.1 Goal: Symbolic Execution .................................... 8
  - 2.2.2 Challenge: Lazy Evaluation ................................... 8
  - 2.2.3 Solution: Lazy Symbolic Execution ........................... 9
- 2.3 Lazy Symbolic Execution ............................................ 12
  - 2.3.1 Syntax ............................................................ 12
  - 2.3.2 Symbolic States ................................................ 13
  - 2.3.3 Symbolic Execution Transitions ............................... 14
- 2.4 Implementation ....................................................... 18
- 2.5 Related Work .......................................................... 18

### 3 G2Q: Haskell Constraint Solving
- 3.1 Introduction ........................................................... 20
- 3.2 G2Q for Program Analysis .......................................... 23
- 3.3 Solver-Aided Interface ............................................... 27
  - 3.3.1 The g2 Quasiquoter ............................................ 27
  - 3.3.2 The G2Rep Typeclass ......................................... 27
  - 3.3.3 Strictness and Fairness ..................................... 27
- 3.4 Solver-Aided Backend ............................................... 30
  - 3.4.1 The G2Rep Typeclass ......................................... 31
  - 3.4.2 g2 Quasiquoter Compilation ................................ 31
  - 3.4.3 Fairness and Heuristic Search ............................... 33
6 Grammar Filtering For Syntax-Guided Synthesis 92
6.1 Introduction ........................................ 92
6.2 Background ......................................... 94
6.3 Overview ............................................. 95
6.4 Data Generation ..................................... 95
  6.4.1 Criticality Data ............................... 96
  6.4.2 Timing Data ................................... 96
6.5 Training .............................................. 97
  6.5.1 Predicting criticality ......................... 97
  6.5.2 Predicting time savings ..................... 97
  6.5.3 Combining predictions ...................... 98
  6.5.4 Falling back to the full grammar .......... 98
6.6 Experiments ......................................... 98
  6.6.1 Technical details ............................. 99
  6.6.2 Results ....................................... 100
6.7 Related .............................................. 104
6.8 Conclusions ........................................ 104

7 Automated Verification and Repair of Firewalls 106
7.1 Introduction ........................................ 106
7.2 Preliminaries ....................................... 108
7.3 Motivating Examples .............................. 110
7.4 System Design ..................................... 112
  7.4.1 Encoding Firewalls and Examples as FOL Formulas ... 113
  7.4.2 Firewall Verification ......................... 119
  7.4.3 Adding Rules .................................. 120
  7.4.4 Consistency Checking ....................... 120
  7.4.5 Formal Guarantees for Repaired Firewalls .... 120
  7.4.6 Rate Limiting Rules Generation .......... 121
  7.4.7 Repair Algorithms ........................... 124
  7.4.8 Redundant Rule Detection .................. 126
7.5 Implementation and Evaluation .................. 126
7.6 Related Work ...................................... 131
7.7 Conclusion ......................................... 132

8 Conclusion ............................................. 133
8.1 Future Directions ................................. 134
  8.1.1 Automated verification ..................... 134
  8.1.2 Formal Methods via Machine Learning .... 135
8.2 Remarks ............................................ 135

viii
List of Figures

2.1 Program with assertion over an infinite list that strict analyzers would struggle with. .................................................. 9
2.2 Symbolic Execution Tree for Example from Figure 2.1. ............... 11
2.3 $\lambda_G$ grammar ................................................................. 13
2.4 Symbolic States ................................................................. 13
2.5 Lazy Transition Rules .......................................................... 15
2.6 Symbolic Transition Rules .................................................... 16
2.7 Two impure $\lambda_G$ programs. ........................................... 17

3.1 Finding the solution to \texttt{sumToN} using a direct encoding to a SMT solver. 21
3.2 Encoding the \texttt{sumToN} problem in the \texttt{g2} quasiquoter. ............ 22
3.3 Simple arithmetics language .................................................. 25
3.4 A program inspired from [69] written with the simple arithmetic language shown in Figure 3.3. It accepts a variable "j" as input, and checks an assertion at its end. .................................................. 26
3.5 The grammar accepted by G2Q. $y$ and $\tau$ represent standard Haskell variables and types, respectively. $e$ represents a standard Haskell expression, which must be of type \texttt{Bool}. .................................................. 28
3.6 Here, we show several examples of G2Q’s behavior on infinite lists, for which the quasiquoter will give output. The key requirement is that the predicate require evaluation of only a finite amount of the infinite input. .................................................. 28
3.7 Here, we show two examples of G2Q’s behavior on infinite lists, that will result in non-termination. .................................................. 29
3.8 Here, we show two examples of \texttt{g2} quasiquoter’s, with an output type that is an infinite data structure. When only a finite amount of the output has to be evaluated to check the predicate, the quasiquoter can return such an infinite data structure. However, trying to satisfy a predicate that requires evaluating an infinite amount of the infinite data structure results in non-termination. .................................................. 29
3.9 A quasiquoter that will fail to terminate, because of an unsatisfiable predicate involving a recursive function. .................................................. 30
3.10 The \texttt{G2Rep} typeclass, which converts values to and from G2’s representation. .................................................. 31
3.11 The \texttt{g2Rep} definition for lists. We denote the standard Haskell list constructors as \texttt{:} and \texttt{[]}, and G2's representation of a list as \texttt{g2Cons} and \texttt{g2Nil}, respectively.

3.12 De Bruijn index based lambda calculus and evaluator.

3.13 N-Queens

3.14 A solution to 8-queens produced by G2Q

3.15 Case study running times.

3.16 all\texttt{QueensSafe}' is the same as all\texttt{QueensSafe}, from Figure 3.13, except the constraint on the list length is moved to the end of the function. This change has a dramatic affect on running time. Solving 8-queens with all\texttt{QueensSafe} takes only 2.30 seconds, while using all\texttt{QueensSafe}' requires 36.42 seconds.

4.1 A Dafny program where \texttt{main} fails to verify due to a weak specification for \texttt{incr}.

4.2 \texttt{\lambda_D} types

4.3 \texttt{\lambda_D} precondition and postcondition

4.4 \texttt{\lambda_D} to \texttt{\lambda_G} translation

4.5 Evaluation results for errors reported by LiquidHaskell on student homeworks. \texttt{Con.} is the number of reported concrete counterexamples. \texttt{Abs.} is the number of abstract counterexamples reported by G2. \texttt{Timeout} is the number of times G2 timed out before returning counterexamples. \texttt{Error} is the errors encountered in G2 when generating counterexamples. \texttt{Avg. Time} is the average amount of time taken by all runs of G2 reported in the table.

5.1 A Dafny program, where verification of \texttt{main} fails due to \texttt{double}'s weak post-condition.

5.2 An overview of Lynx's inference algorithm.

5.1 \texttt{traverseCG}_T \texttt{k} synthesizes specifications for use by a modular verifier.

5.3 \texttt{evalCE}(E_U, cex) maps counterexamples to either constraints on the specifications, or returns a counterexample if it indicates a flaw in the users specification.

5.2 \texttt{iterateInfer}_T \texttt{k} uses \texttt{traverseCG}_T \texttt{k} to synthesize environments in \texttt{U}_k \texttt{T}^k.

5.4 LiquidHaskell grammar

5.5 Evaluation of Lynx: \textbf{Main Function} is the benchmark name; \texttt{LoC} and \texttt{Funcs} are the number of lines of code and functions in the benchmark; \texttt{Lvls} is the maximum call-graph depth in the benchmark; \texttt{Time} is the number of seconds needed to infer modular specifications that verify the benchmark; \texttt{Loops} is the number of CEGIS loop iterations needed to infer specifications; \texttt{Lvl Dec} and \texttt{Negated Models} is the number of times the backtracking and negated models heuristics were used.
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Translation of a C loop (in 5.6a) to a Haskell recursive function (in 5.6b.)</td>
<td>88</td>
</tr>
<tr>
<td>5.7</td>
<td>Running times of Lynx on the benchmarks from [70] with a timeout of 10 minutes.</td>
<td>88</td>
</tr>
<tr>
<td>6.1</td>
<td>GRT uses the grammar $G$ and constraints $C$ to predict how critical each function is, and the amount of time that would be saved by eliminating it from the grammar. Then, it outputs a new grammar $G^<em>$, which it expects will speed up synthesis over the original grammar (that is, it expects that $T_{C, G}^</em> &lt; T_{C, G}$).</td>
<td>95</td>
</tr>
<tr>
<td>6.2</td>
<td>The top 20 problems with longest synthesis time for CVC4 (excepting timeouts), and the corresponding synthesis times for GRT+CVC4.</td>
<td>99</td>
</tr>
<tr>
<td>6.3</td>
<td>When the GRT+CVC4 found a different solution than CVC4, it was on average shorter than the solution found with the full grammar.</td>
<td>101</td>
</tr>
<tr>
<td>6.4</td>
<td>Synthesis results over the 30 longest running benchmarks from SyGuS Competition’s PBE Strings track.</td>
<td>103</td>
</tr>
<tr>
<td>7.1</td>
<td>An visualization of a limit, as packets try to match a rule.</td>
<td>110</td>
</tr>
<tr>
<td>7.2</td>
<td>An example of a firewall repair problem.</td>
<td>111</td>
</tr>
<tr>
<td>7.3</td>
<td>The workflow overview of FireMason.</td>
<td>113</td>
</tr>
<tr>
<td>7.4</td>
<td>In a FirewallMap, ACL’s are duplicated for each point that they can be jumped to from.</td>
<td>115</td>
</tr>
<tr>
<td>7.5</td>
<td>Scalability for number of rules.</td>
<td>127</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Analysis, verification, and synthesis techniques allow programmers to check and prove properties of their programs, and automatically derive programs from descriptions of desired behavior. Integration of verification techniques into development workflows allows an increased level of confidence in the correctness of software. Unfortunately, there are significant barriers to widespread use of formal methods techniques, including unintuitive and hard to use tools, insufficiently automated tools, and tools that do not scale to real world problems [62].

Of course, there are some success stories. For example, Microsoft’s Static Device Verifier (the product based on the SLAM research project) [36] applies formal methods techniques to fully automatically verify the absence of certain classes of bugs, or find counterexamples exposing bugs, in Windows device drivers. Perhaps even more ubiquitously, FlashFill [90], a synthesis tool for string transformations, has been integrated into Excel with an easy-to-use interface. Still, by and large, use of verification and synthesis tools currently requires a great deal of domain expertise.

To reduce this burden, this dissertation describes techniques to formalize language semantics- in particular, non-strict semantics and firewall scripting languages- in a way that allows automated logical reasoning via SMT solvers. Further, we show how this enables a range of applications to be built to analyze, verify, and repair programs.

1.1 Symbolic Execution

Symbolic execution is a well known technique for running programs with symbolic variables instead of concrete inputs. A variety of symbolic execution engines have been developed for both imperative [46, 154] and functional languages [189, 190, 143]. Traditionally, symbolic execution has been widely employed to test programs, by symbolically searching for values that violate test cases or assertions.

We describe the design of a symbolic execution engine named G2. Unlike existing symbolic execution engines, G2 specifically targets execution of programs in statically typed non-strict functional languages, such as Haskell. This requires a reformulation of the reduction rules used in symbolic execution to reduce to a newly defined Symbolic
Weak Head Normal Form. G2 benefits from the static types of our language, as it reduces the number of possible instantiations for symbolic variables. Symbolic functions are supported via defunctionalization [169], using in scope functions of the correct type as possible instantiations. We leave implementation of more complete techniques, such as those used in [189, 143], as future work. We build on G2 to extend the use cases for symbolic execution beyond testing and bug finding.

1.1.1 Constraint Solving

Constraint solvers, such as SMT solvers, have a wide variety of use cases [68, 196, 197, 195, 149, 178, 75, 164]. A common technique for a program to interact with an SMT solver is to pass formulas to the solver, as strings in a format called SMT-LIB. Unfortunately, this approach has downsides: it can require redefining datatypes and functions that exist in the original program, and, if working in a statically typed language, it loses type soundness. Further, SMT solvers often struggle with formulas that include quantifiers or recursive functions.

We use G2 to develop a library, G2Q, which allows constraint problems to be written directly as Haskell code. This allows Haskell data types to be directly used in the constraints, and ensures correct typing between the constraints inputs and outputs and the solver. The use of symbolic execution allows the gradual unrolling of recursive functions, thus allowing G2Q to solve problems that are out of scope when directly encoded as SMT formulas. The functionality of G2Q is demonstrated via a number of case studies.

1.1.2 Modular Verification

As another application of symbolic execution, we explore debugging and automating the process of modular verification. Modular verifiers, such as Dafny [128], ESC-/Java [81], and LiquidHaskell [197], allow users to write and prove high-level specifications for their code. They have been widely used both in academia and industry to verify properties of the Curiosity rover’s agent-environment interface [49], model-view-controller web applications [127], encryption libraries [28], SAT solvers [29], mutable data structure libraries [42], red-black trees [156], and widely used Haskell libraries [196].

Unfortunately, modular verifiers error messages are not very intuitive. In particular, error messages may not indicate an actual bug in the code- an error message may instead just indicate that the verifier failed to prove a true specification correct. This is because modular verifiers rely on abstracting callee functions with just those function’s specifications. Consider a function $f$, which calls a function $g$. When trying to verify the function $f$, the verifier considers only the specification- not the definition- of $g$. Such spurious errors do not identify actual bugs in the code- rather, they arise simply because a callee function has a specification that is too weak. There are good reasons for designing a verifier with this behavior. From a theoretical perspective,
Inlining all function calls is impossible in the case of unbounded recursion. From a practical perspective, even without recursive functions or other loops, inlining all functions may result in poor scalability from the verifier. Nonetheless, the solution of abstracting all function calls still leaves users—especially those unfamiliar with or new to modular verification—in a difficult position.

**Debugging** We introduce a technique called *counterfactual symbolic execution* to aid in the debugging of modular verification errors. Counterfactual symbolic execution finds two types of counterexamples. *Concrete counterexamples* show actual inconsistencies between the specifications and the code, by finding inputs which satisfy the preconditions of a function, but violate the postconditions. *Abstract counterexamples* help explain spurious errors, by identifying (1) functions which require stronger specifications and (2) incorrect input/output pairs for those functions, which the correct specification allows, but which a strong enough specification would rule out. We evaluated this approach on a corpus of 7550 errors gathered from users of the LiquidHaskell refinement type system. We show that for 97.7% of these errors, G2 is able to quickly find counterexamples that show how the code or specifications must be fixed to enable verification.

**Automation** Counterfactual symbolic execution improves the debugging experience when working with a modular verifier. Rather than having to reason directly about imprecise error messages, counterexamples allow flawed or weak specifications to be easily identified. However, this still takes some amount of effort from the user to manually correct the error and write specifications. Often, necessary specifications are non-trivial, making this task quite difficult.

We extend counterfactual symbolic execution to automate the process of modular verification. A counterexample-guided inductive synthesis (CEGIS) loop based inference algorithm for modular verification specifications is introduced. The algorithm is parameterized over a verifier, counterexample generator, and constraint guided synthesizer. We show that if each of these three components is sound and complete over a finite set of possible specifications, our algorithm is sound and complete as well. Additionally, we introduce *size-bounded* synthesis functions, which extends our completeness result to an infinite set of possible specifications. In particular, we describe a size-bounded synthesis function for linear integer arithmetic constraints. We evaluate our inference algorithm on a variety of programs.

### 1.2 Syntax-Guided Synthesis Grammar Reduction

Syntax-guided synthesis is a synthesis paradigm in which functions satisfying user-provided constraints are synthesized by searching over a grammar. SyGuS is a widely used technique in a variety of contexts, including synthesis of functions and loop invariants [51, 35, 173, 183]. Motivated by the number of use case, a common specification language for SyGuS problems has been developed [24, 167], and a number of off-the-shelf Syntax-Guided Synthesis solvers have been developed [27, 168, 106, 152].
We describe an application of machine learning to improve the performance of off-the-shelf SyGuS solvers. In particular, we focus on improving the performance of SyGuS tools solving programming-by-example (PBE) problems. In a PBE problem, all the constraints consist of input/output examples. We show that, by preprocessing SyGuS PBE problems with a neural network, we can use a data driven approach to reduce the size of the search space, thus allowing automated reasoning-based solvers to more quickly find a solution analytically. Our system, the Grammar Reduction Tool (GRT), is able to run atop existing SyGuS PBE synthesis tools, decreasing the runtime of the winner of the 2019 SyGuS Competition for the PBE Strings track by 47.65% to outperform all of the competing tools.

1.3 Firewall Analysis and Repair

Firewalls play an important role in networks, as they manage incoming and outgoing network traffic. Ensuring the correctness of a firewall is essential for security. Because enterprise-scale firewalls contain hundreds or thousands of rules, ensuring the correctness of firewalls – that the rules in the firewalls meet the specifications of their administrators – is an important but challenging problem. Thus, a large number of tools have been developed to analyze, verify, and debug firewalls [136, 192, 131, 142, 200, 207, 142, 136].

We describe FireMason, a tool to analyze and automatically repair firewalls. Unlike existing work, FireMason implements a repair by example based approach, which automatically modifies firewalls based on user provided examples. Once an administrator observes undesired behavior in a firewall, they may provide input/output examples that comply with the intended behaviors. Based on the examples, FireMason automatically synthesizes new firewall rules for the existing firewall. This new firewall correctly handles packets specified by the examples, while maintaining the rest of the behaviors of the original firewall. Through a conversion of the firewalls to SMT formulas, we offer formal guarantees that the change is correct. Our evaluation results from real-world case studies show that FireMason can efficiently find repairs.

1.4 Contributions

This dissertation establishes techniques to analyze and repair programs via formalisms of program semantics that enable the use of SMT solvers.

Symbolic Execution We introduce G2, the first symbolic execution engine for statically typed non-strict functional programming languages, such as Haskell. Existing symbolic execution engines for functional languages [189, 190, 143] target strict languages, and thus, if applied to a non-strict language, may find spurious assertion violations or miss actual assertion violations. G2's symbolic execution reduces expressions to Symbolic Weak Head Normal Form, via classical lazy graph reduction
semantics [135, 161, 162] augmented with support for symbolic variables. We formalize the reduction strategy via a collection of reduction rules.

We use G2 to construct a library, G2Q, to allow constraints written as (mostly) ordinary Haskell code to be solved. Alternatively, one could encode constraint problems in SMT-LIB and call an SMT solver directly, but G2Q has a number of advantages over this approach: G2Q allows the reuse of existing Haskell datatypes and functions, ensures type soundness when working with constraints, and enables effective use of recursive functions (which SMT solvers tend to struggle with.) We outline the design of the front end, which allows constraints to be specified via a Haskell language feature called quasiquoters, and the backend, which solves the constraints using symbolic execution, and translates values between regular Haskell values and G2’s intermediate representation. We explore use cases for G2Q and evaluate G2Q’s effectiveness using a number of case studies.

**Debugging and Automation of Modular Verifiers** We introduce counterfactual symbolic execution to aid in the debugging of modular verifiers. An error from a modular verifier may be a true error, indicating that a specification is actually incorrect, or a spurious error, in which case some other specification in the program must be strengthened for verification to succeed. Counterfactual symbolic execution returns two kinds of counterexamples: concrete counterexamples, to explain true errors, and abstract counterexamples to explain spurious errors. To the best of our knowledge, abstract counterexamples are the first technique to be able to identify specific functions that require stronger specifications to avoid spurious errors. We define a translation from refinement types into G2’s language, allowing counterfactual symbolic execution to be used to debug LiquidHaskell specification. We show that G2 is able to find a counterexample for 97.7% of an existing collection of LiquidHaskell errors. We further evaluate our explanations of spurious failures and find that in at least 96.1% of cases, our explanation of how the user can fix the spurious error is correct.

We further extend counterfactual symbolic execution to automatically fix spurious errors by inferring needed specifications from counterexamples. We prove soundness and completeness results for our algorithm, in particular showing that if our synthesizer satisfies certain conditions and if there is some assignment of synthesizable specifications to functions that would allow verification to succeed, we will find that assignment. We show that these conditions are practical by constructing such a synthesizer for specifications drawn from the infinite set of linear integer arithmetic specifications. We evaluate the approach on benchmarks based on existing LiquidHaskell code, as well as translation of existing C loop invariant benchmarks.

**SyGuS Grammar Filtering** We introduce a new technique to improve the performance of Syntax-Guided Synthesis (SyGuS) solvers via a machine learning based preprocessor. The technique works by filtering the grammar in the SyGuS problem to eliminate non-critical functions (functions that do not appear likely to contribute to a solution.) We identify two key factors to predict per function: criticality (whether a
given function is critical) and potential time savings (the amount of time that will be saved by eliminating a given function.) In our evaluation, we empirically show that our technique effectively speeds up a SyGuS solver on an existing benchmark set, and that this success relies on both the criticality and potential time savings predictions.

Firewalls Finally, we define a formalism for firewalls (in particular, the iptables specification language) via a translation to first order logic with linear integer arithmetic and uninterpreted functions. Existing work formalized certain aspects of firewall specification languages [136, 192, 131, 142, 200, 207, 142, 136]. However, to the best of our knowledge, this is the first formalization of rate-limiting (time based) firewall rules. We then use this formalization to build the first tool to automate the repair of firewalls, by using SMT solving for template based synthesis. We evaluate on several real world examples gathered from StackOverflow.

1.5 Remarks

Allowing widespread use of formal methods by programmers who are not formal methods experts requires that formal methods tools be made accessible and scalable. This works outlines techniques to analyze and repair programs, aided by formalisms of the programming language semantics that enables the use of SMT solving.

As part of this goal, we develop backend techniques. For example, we develop an encoding of firewalls into first-order logic, and G2, a symbolic execution engine for Haskell. We then extend these techniques to aid in automation and debugging of verification and repair tasks. Using G2, we develop G2Q, to aid in constraint solving, and systems for modular verification debugging and automation. Using our firewall formalism, we implement FireMason a tool to automatically analyze and repair firewalls. Widespread use of formal methods techniques also requires that they scale to large problems- GRT explores the route of using machine learning to improve the performance of synthesizers, allowing applications to larger and more complex problems.
Chapter 2

Non-strict Symbolic Execution

This chapter describes work completed in collaboration with Anton Xue, Maxwell Bland, Ranjit Jhala, and Ruzica Piskac. This work includes material originally published in [94].

2.1 Introduction

This chapter outlines the design of a lazy symbolic execution engine for languages with non-strict semantics. Existing work on symbolic execution [118, 46, 143] uses laziness as an implementation technique to improve the efficiency of symbolic execution for languages with strict semantics. On the other hand, our work describes symbolic execution of Haskell, a language with non-strict semantics. Consequently, as we show in Section 2.2.2, existing symbolic execution engines can fail to find simple counterexamples that arise with lazy evaluation. Similarly, they can return spurious counterexamples that are avoided by lazy evaluation.

We solve this problem by augmenting classical lazy graph reduction semantics [135, 161, 162] with symbolic variables to reduce terms into Symbolic Weak Head Normal Form, that only computes values as needed, thereby obtaining the first symbolic execution framework for a non-strict language. Our practical implementation, G2, targets symbolic execution of Haskell programs.

In this chapter we outline the design of the core G2 symbolic execution engine. In Chapters 3, 4, and 5, we will explore applications of symbolic execution and G2 to a variety of problems. In Chapter 3, we will show how symbolic execution can ease accessing constraint solvers from a traditional programming language. In Chapters 4 and 5, we will describe approaches to use symbolic execution for the debugging and automating of modular verification.
2.2 Overview

We start with an overview of our goals and the challenges posed by lazy evaluation and show how we solve these challenges via lazy symbolic execution.

2.2.1 Goal: Symbolic Execution

Our first goal is to implement a symbolic execution engine for non-strict languages like Haskell. Such an engine would take as input a program like:

```haskell
intersect :: (Eq a) => [a] -> [a] -> [a]
intersect xs ys = [x | x <- xs , any (x ==) ys]

any :: (a -> Bool) -> [a] -> Bool
any _ [] = False
any p (x:xs) = p x || any p xs
```

together with a property, specified as an assertion about the behavior of the program over some unknown inputs, e.g. that the `intersect` function above was commutative:

```haskell
let xs = ?; ys = ? in assert (xs `intersect` ys == ys `intersect` xs)
```

Our engine then symbolically evaluates all executions of the above program (up to some given number of reduction steps) to find a counterexample, i.e. values for `xs` and `ys` under which the asserted predicate is `False`:

```haskell
counterexample : assert fails when
xs = [0, 1], ys = [1, 1, 0]
```

2.2.2 Challenge: Lazy Evaluation

While there are several symbolic execution engines that can produce the above result [48], including those for functional languages like F# [188], Scala [122], and Racket [189, 191], all of these tools assume strict or call-by-value semantics. This is problematic for a non-strict language (like Haskell.) Strict evaluation can both miss assertion failures, and report spurious failures that cannot occur under lazy evaluation.

**Strictness Reports Spurious Failures** Consider:

```haskell
let f x = 10; g _ = assert False in f (g 0)
```

Under strict evaluation, `g 0` would be computed first, violating the assertion. However, under non-strict semantics, `f` is evaluated first, and immediately returns `10` without evaluating its argument. Thus, as `g 0` is never reduced, the assertion is never evaluated and, hence, does not fail.

**Strictness Misses Real Failures** Even worse, strict symbolic execution can miss errors in code that relies explicitly on lazy evaluation. For example, consider the code in Figure 2.1. The code uses two functions, `!`, which returns the j-th element of the
let xs ! j = case xs of
    h:t -> case j == 0 of
        True -> h
        False -> t ! j-1
    repl n = n : repl (n + 1)
i = ?; k = ?
in assert (repl i ! k == i)

Figure 2.1: Program with assertion over an infinite list that strict analyzers would struggle with.

list xs, and repl, which returns an infinite list starting at n. The code asserts that the k-th element of repl i should be i. Strict symbolic execution will keep unfolding the infinite list corresponding to the term repl i, and thus, will miss that the assertion can be violated by lazily evaluating the asserted predicate on a finite prefix.

2.2.3 Solution: Lazy Symbolic Execution

In this paper we solve the problems caused by strictness by developing a novel lazy symbolic execution algorithm. At a high-level, our algorithm mimics the lazy graph reduction semantics of non-strict languages like Haskell, where terms are only reduced by need, up to Weak Head Normal Form (WHNF), i.e. enough to resolve pattern-match branches. Our key insight is that we can generalize the classical semantics to account for symbolic values that denote unknown inputs, by developing a notion of Symbolic WHNF (SWHNF), where terms are reduced up to symbolic variables whose values are constrained by path constraints that capture the branch information leading up to that point in the execution.

Symbolic States Symbolic execution evaluates a State, which is a triple \((E, H, P)\) comprised of an expression \(E\) being evaluated, a heap \(H\), mapping variables to other expressions, and path constraints \(P\), which are logical formulas constraining the values of symbolic variables in \(E\) and \(H\).

Symbolic Execution Tree Figure 2.2 shows the tree of states resulting from symbolic executing the code in Figure 2.1. Each node is a symbolic state, and has children corresponding to the states that the parent node can transition to.

- Initial State: The initial symbolic state \(S_0\) is comprised of \(E_0\), the source program expression, \(H_0\), the initial empty heap, and \(P_0\), the trivial path constraint (True).

- Variable Binding: \(S_0 \rightarrow S_1\) accounts for the let-bindings, which are not evaluated, but are, instead, bound on the heap as shown in \(S_1\). The symbolic variables \(k\) and \(i\) correspond to the (unknown) input values.
• **Variable Lookup and Application:** $S_1 \hookrightarrow S_2$ looks up and applies the definition of the list index operator $!$ to repl $i$ and $k$. Due to laziness, we create fresh bindings on the heap, rather than evaluate the arguments.

• **Lazy Evaluation to Symbolic WHNF:** $S_2 \hookrightarrow S_3$ looks up $xs_2$ – namely repl $i$ – and lazily evaluates it to SWHNF, *i.e.* precisely enough to determine which of the patterns to branch on. Under strict semantics, the list would have to be completely evaluated *before* picking a case alternative, but since repl generates an infinite list, this evaluation would never terminate.

• **Pattern Matching:** $S_3 \hookrightarrow S_4$ matches the non-empty list against the cons-pattern by introducing fresh binders $h_4$ and $t_4$, and binding them to the respective terms on the heap $H_4$.

• **Symbolic Branching:** At $S_4$, the scrutinized expression is $j_2 = 0$ which, after looking up $j_2$ in the heap, is $k = 0$. This contains a symbolic value $k$ and hence, is in SWHNF, so it could evaluate to True or False. Therefore, there are two possible transitions, to $S_5$ and $S_7$. We strengthen the path constraints $P_5$ and $P_7$ with $k = 0$ and $-k = 0$ respectively, to record the condition under which the transition occurred. $S_5 \hookrightarrow S_6$ looks up $h_4$ to reduce the asserted predicate to a tautology $i = i$, meaning the assertion holds.

• **Recursive Unfolding:** The symbolic execution continues to explore the other branch, $S_4 \hookrightarrow S_7$. Again, the binders are lazily looked up on the heap. Via a sequence of transitions we arrive at $S_{10}$, where the head of the list is bound to the value $h_{10} = i + 1$.

• **Assertion Failure:** Again, at $S_{11}$ we have a symbolic branch on the term $k - 1 = 0$. This time, however, the True branch transitions to $S_{12}$ where the asserted predicate has been reduced to $h_{10} = i$. $S_{11} \hookrightarrow S_{12}$ looks up $h_{10}$ in the heap to find that the asserted predicate, $i + 1 = i$, is not True. Thus, our symbolic execution reports a counterexample to the assertion in Figure 2.1.

We can obtain a satisfying assignment (*i.e.* a model) for the path constraints at the point of violation to obtain concrete values for the symbolic inputs that lead to the failure. This allows us to determine concrete values that violate the assertion. For example, here, the SMT solver tells us that the assertion is violated when $k = 1$ and not, *e.g.* when $k = 0$. 
Figure 2.2: Symbolic Execution Tree for Example from Figure 2.1.
2.3 Lazy Symbolic Execution

Here, we describe a core language $\lambda_G$ (Section 2.3.1), which draws inspiration from GHC’s Core language [160]. We formalize lazy symbolic execution as a novel reduction semantics (Section 2.3.3). We then show how to extend this language with counterfactual branching (Section 4.3), and how to use the resulting framework to localize refinement type errors (Section 4.4).

2.3.1 Syntax

Figure 2.3 summarizes the syntax of our core language $\lambda_G$, a typed lambda calculus extended with special constructs for symbolic execution.

- **Terms** include literals, variables, data constructors, function application, lambda abstraction, let bindings, and case expressions.

- **Case** expressions `case e of \{ a \}` operate on algebraic data types. We refer to `e` as the scrutinee, and to `a` as alternatives, each of which maps a pattern `D \pi` – comprising a constructor `D` and a sequence of (bound) pattern variables `\pi` – to the expression that should be evaluated when the scrutinee matches the pattern. As is standard, Boolean branches correspond to a case-of over the patterns `True` and `False`.

- **Symbolic variables** denote some unknown value. We assume that all symbolic binders are to first order values: higher-order values are orthogonal and can be handled via the approach of [189].

- **Symbolic generator** expressions `? : \tau` are used to introduce new symbolic variables of type `\tau`.

- **Assume** expressions `assume e_1 in e_2 condition` the evaluation of `e_2` upon whether `e_1` evaluates to `True` and cause evaluation to halt otherwise.

- **Assert** expressions `assert e_1 in e_2 check` that `e_1` evaluates to `True` and cause evaluation to CRASH otherwise.

- **Counterfactual** branch expressions `e_1 □ e_2` nondeterministically evaluates to either `e_1` or `e_2`.

**Types** Every expression has a type. We write `$e : \tau$` to denote that `$e$` has type `$\tau$`. Type checking $\lambda_G$ is standard for polymorphic functional languages, e.g., the rules used in System F$^\uparrow_C$ [162], and is omitted for brevity. `assume e_1 in e_2` and `assert e_1 in e_2` require that `e_1` have type `Bool`. In a counterfactual branch, both expressions must have the same type.
2.3.2 Symbolic States

Next, we formalize the notion of lazy symbolic execution by presenting a new symbolic, non-strict operational semantics for $\lambda_G$ formalized via rules that show how a program transitions between symbolic states. Figure 2.4 summarizes the syntax of symbolic states, $S$, which are tuples of the form $(E,H,P)$. The expression $E$ corresponds to the term that is being evaluated. The heap $H$ is a map from (bound) variables $x$ to terms $e$. As is standard, the heap is used to store unevaluated thunks (i.e. unevaluated expressions) until the point at which they are needed. The path constraint $P$ is a conjunction of logical formulas that describes the values that (symbolic) variables must have in order for computation to have proceeded up to the given state. We will use $P$ to capture the conditions under which evaluation proceeds along different case-branches.

Well-formedness Only symbolic variables may occur free in a state, all other variables are bound, either on the heap, or by a lambda, let, or case expressions. We
denote the binding of a variable $x$ to an expression $e$ in the heap $H$ as $H\{x = e\}$. We write $\text{lookup}(H, x)$ for the expression to which $x$ is bound in $H$. If there is no such binding, $\text{lookup}(H, x)$ is not defined.

**Symbolic Variables and Primitive Applications** $\text{Sym}(e)$ checks if an expression is a symbolic variable, or is a primitive application that cannot be concretely reduced:

$$
\text{Sym}(e) = \begin{cases}
\text{True} & e = s \\
\text{True} & e = e_1 \oplus e_2 \land (\text{Sym}(e_1) \land \text{Sym}(e_2)) \\
\text{True} & e = e_1 \oplus l \land \text{Sym}(e_1) \\
\text{True} & e = l \oplus e_2 \land \text{Sym}(e_2) \\
\text{False} & \text{otherwise}
\end{cases}
$$

**Symbolic Weak Head Normal Form** The essence of non-strict semantics, *e.g.* in Haskell, is to reduce expressions to Weak Head Normal Form (WHNF) [162], *i.e.* a literal, lambda abstraction, or data constructor application. Consequently, the heart of our lazy symbolic execution is a notion of Symbolic Weak Head Normal Form (SWHNF), that generalizes WHNF to account for (unknown) symbolic values. Formally, an expression $e$ is in SWHNF if the predicate $\text{SWHNF}(e)$ holds:

$$
\text{SWHNF}(e) = \begin{cases}
\text{True} & e \equiv l \\
\text{True} & e \equiv s \\
\text{True} & e \equiv D\overline{E} \\
\text{True} & e \equiv \lambda x . e \\
\text{True} & e \equiv e_1 \oplus e_2 \land \text{Sym}(e) \\
\text{False} & \text{otherwise}
\end{cases}
$$

### 2.3.3 Symbolic Execution Transitions

We formalize lazy symbolic execution via the transition relation $S \rightarrow S'$ that says that the state $S$ takes a single step to the state $S'$. The transition relation is formalized via the rules in Figures 2.5 and 2.6. For some states, more than one rule applies, or there is more than one way to apply a single rule. From the perspective of a single execution, this requires a nondeterministic decision to apply one of the rules. However, during symbolic execution, we split the state, by applying each potential rule, allowing us to explore all possible program runs up to some bounded number of transitions.

**Lazy Transitions**

We now describe the reduction rules, shown in Figure 2.5, that formalize lazy execution.

**Bindings** and **variables** are implemented via lazy evaluation facilitated by the heap.
\[
\begin{align*}
\text{VAR} & \quad e = \text{lookup}(H, x) \\
\text{SWHNF}(e) & \quad (x, H, P) \leftarrow (e, H, P) \\
\text{VAR-RED} & \quad e = \text{lookup}(H, x) \quad \neg\text{SWHNF}(e) \\
& \quad (e, H, P) \leftrightarrow (e', H', P') \\
\neg\text{SWHNF}(f) & \quad (f, H, P) \leftrightarrow (f', H', P') \\
\text{LET} & \quad (\text{let } x = e_1 \text{ in } e_2, H, P) \leftarrow (e'_2, H \{x' = e'_1\}, P) \\
\text{APP} & \quad (\lambda x . e_1) e_2, H, P) \leftarrow (e'_1, H', P) \\
\text{PR-L} & \quad (e_1 \oplus e_2, H, P) \leftarrow (e'_1 \oplus e_2, H', P') \\
\text{PR-R} & \quad (e_1 \oplus e_2, H, P) \leftarrow (e_1 \oplus e'_2, H', P') \\
\text{CASE-EV} & \quad (\text{case } e \text{ of } \{a\}, H, P) \leftrightarrow (e', H', P') \\
\text{CASE} & \quad (\text{case } D e_1 \ldots \text{ of } \{D \bar{x} \rightarrow e, \ldots\}, H, P) \leftarrow (e[\bar{x}' / \bar{x}], H \{x'_1 = e_1 \ldots\}, P) \\
\text{CASE-SYM} & \quad (\text{case } e \text{ of } \{D \bar{x} \rightarrow e_a, \ldots\}, H, P) \leftarrow (e_a[\bar{x}' / \bar{x}], H, P \land e = D \bar{x}')
\end{align*}
\]

Figure 2.5: Lazy Transition Rules

VAR and VAR-RED lookup a concrete variable, \(x\), in the heap, to find the expression it is mapped to, \(e\). If \(e\) is already in SWHNF, it is simply returned by VAR. Otherwise, VAR-RED reduces \(e\) to an expression, \(e'\), in SWHNF, before both returning \(e'\), and remapping \(x\) to \(e'\) in the heap. Typically, VAR-RED is simply an optimization in case \(x\) is reevaluated: since Haskell is pure, evaluating \(e\) repeatedly would be semantically correct, but inefficient [135]. However, in Section 2.3.3, we will see that during symbolic execution with symbolic generators or counterfactual branching, this rule takes on a new importance.

LET and APP-LAM both bind an expression in the heap, \textit{without} evaluating the expression. APP reduces the function in a function application, without reducing the arguments.
\[
\begin{align*}
(e_1 \square e_2, H, P) & \hookrightarrow (e_1, H, P) & \text{CH-L} \\
(e_1 \square e_2, H, P) & \hookrightarrow (e_2, H, P) & \text{CH-R} \\
\end{align*}
\]

\[
\begin{align*}
s \text{ fresh} & \quad (\text{?} : \tau, H, P) \hookrightarrow (s, H, P) & \text{SYM-GEN} \\
(e_p, H, P) & \hookrightarrow (e'_p, H', P') & \text{ASSUME-EV} \\
(\text{assume } e_p \text{ in } e_b, H, P) & \hookrightarrow (\text{assume } e'_p \text{ in } e_b, H', P') & \\
\text{SWHNF}(e_p) & \quad (\text{assume } e_p \text{ in } e_b, H, P) \hookrightarrow (e_b, H, e_p \land P) & \text{ASSUME} \\
(e_p, H, P) & \hookrightarrow (e'_p, H', P') & \text{ASSERT-EV} \\
(\text{assert } e_p \text{ in } e_b, H, P) & \hookrightarrow (\text{assert } e'_p \text{ in } e_b, H', P') & \\
\text{SWHNF}(e_p) & \quad (\text{assert } e_p \text{ in } e_b, H, P) \hookrightarrow (e_b, H, e_p \land P) & \text{ASSERT} \\
\text{SWHNF}(e_p) & \quad \text{isSMTSat}(\neg e_p \land P) & \text{ASSERT-CRASH} \\
(\text{assert } e_p \text{ in } e_b, H, P) & \hookrightarrow (\text{CRASH}, H, \neg e_p \land P) & \\
\end{align*}
\]

**Figure 2.6:** Symbolic Transition Rules

**Primitive** operations arguments are evaluated to SWHNF by Pr-L and Pr-R. If both of the arguments of a primitive are concrete literals, Pr evaluates the primitive concretely.

**Case** expressions require the *scrutinee* be evaluated to SWHNF, so that the correct alternative can be picked. This evaluation is performed by CASE-EV. If the scrutinee is concrete, CASE continues evaluation on the correct alternative expression. If the scrutinee is a symbolic variable, CASE-SYM nondeterministically chooses an alternative expression.

**Symbolic Transitions**

We now turn our attention to the reduction rules in Figure 2.6, which shows constructs particular to symbolic execution.

**Counterfactual branches** proceed nondeterministically by either CH-L or CH-R, allowing reduction on either \(e_1\) or \(e_2\).

**Symbolic generators** are evaluated using SYM-GEN, which introduces a fresh symbolic value \(s\).

**Assume** expressions are evaluated by first reducing the predicate \(e_p\) to SWHNF using ASSUME-EV. Then, the rule ASSUME adds the predicate to the path constraint, thereby recording that the predicate must hold for computation to proceed.
\[
\text{let } f = \lambda x . x \Box ? \text{ in } f \ 2 \ \ast \ \ast \ f \ 2
\]

(a) A program which evaluates \( f \ 2 \) twice.

\[
\text{let } f = \lambda x . x \Box ? ; \ y = f \ 2 \text{ in } y \ \ast \ y
\]

(b) A program which evaluates \( f \ 2 \) once.

Figure 2.7: Two impure \( \lambda G \) programs.

 Assert expressions are handled similarly in that the predicate is first reduced to SWHNF. Next, we check that the predicate actually evaluates to True—otherwise execution CRASH-es due to an assertion violation. To this end, Assert-Crash queries the SMT solver for satisfying assignments of our symbolic variables, that falsify the predicate, i.e. which cause the predicate to evaluate to False. If the SMT solver finds such an assignment, we can show the user the inputs that cause the assertion violation. If no such assignment can be found, Assert proceeds to evaluate the inner expression \( e_b \) under a strengthened path constraint.

Impurity of Symbolic Transition Rules

Unlike GHC’s Core Haskell, \( \lambda G \) is impure, due to Symbolic Generators and Counterfactual Branching. For instance, consider the \( \lambda G \) program in Figure 2.7a. This program is reducible to four different values in SWHNF: 4, \( 2 \ \ast \ s \), \( s \ \ast \ 2 \), or \( s \ \ast \ s' \) (where \( s \) and \( s' \) are symbolic variables.) The evaluation of \( f \ 2 \) may vary, even in a single reduction.

When symbolically executing the program, we explore each of these 4 branches separately. However, it is often desirable to require two values to match within an individual state. For example, we might wish to ensure that both calls to \( f \ 2 \) either result in 2, or result in the same symbolic value.

We can achieve this with the program shown in Figure 2.7b. In a strict, call by value language, it would be clear why this program achieved the desired result: \( y \) would be computed only once, during the let binding, and before the evaluation of the multiplication. In a lazy setting, \( f \ 2 \) is stored as a thunk, and only computed when forced by the multiplication. A natural question then arises: why does this program work in our lazy setting?

This is a result of us taking advantage of the Var-Red rule. During normal execution, this rule is just an optimization, but during symbolic execution, it allows us to control nonpurity. In the modified program, in Figure 2.7b, it means that, even though the reduction is performed only when needed, the reduction of \( y \) (and thus the reduction of \( f \ 2 \)) is still performed only once. Thus, there are only 2 possible values in SWHNF: 4, and \( s \ \ast \ s \).
Completeness of Symbolic Execution

We write $\leftrightarrow_c$ for the concrete transition relation obtained by replacing the rule $\text{Sym-Gen}$ with $\text{Conc-Gen}$, shown below, which replaces a symbolic generator with some total expression of the suitable type:

$$\exists e' : \tau \quad (\exists e, H, P) \leftrightarrow_c (e', H, P)$$

The concrete transitions correspond exactly to the usual standard non-strict operational semantics; there are no symbolic values anywhere, and the path constraint is just $\text{True}$.

Completeness Let $\leftrightarrow^*$ and $\leftrightarrow_c^*$ respectively denote the reflexive transitive closure of $\leftrightarrow$ and $\leftrightarrow_c$. We can prove by induction on the length of the transition sequences that if the concrete execution can CRASH then so can the symbolic execution:

Theorem 1. $(e, \emptyset, \text{True}) \leftrightarrow^*_c (\text{CRASH}, \cdot, \cdot)$ iff $(e, \emptyset, \text{True}) \leftrightarrow^* (\text{CRASH}, \cdot, \cdot)$.

2.4 Implementation

We have implemented lazy symbolic execution for the Haskell language in a tool named G2. It is open source, and available at https://github.com/BillHallahan/G2. Additionally, G2 is available on Hackage at http://hackage.haskell.org/package/g2. We use the GHC API to parse Haskell programs, and Z3 [64] and CVC4 [39] as SMT solving backends. G2 supports a large Haskell98-like subset of the code compiled by GHC, which also includes features not detailed in Section 2.3.1 such as polymorphism. G2 uses a custom version of Haskell’s Base library and Prelude [158]. For a range of modules, functions, and datatypes, G2 can use this custom standard library to symbolically execute programs written with the standard Base and Prelude.

2.5 Related Work

Haskell Libraries, Program Analysis, and Testing Catch [138] and Reach [140] are static analyses for Haskell that look for specific kinds of errors as opposed to our general symbolic execution. QuickCheck [58] and SmallCheck [171] and Target [178] test properties by running Haskell code on large numbers of random or SMT-generated, concrete inputs.

Symbolic Execution for Functional Languages CutEr [85] is a symbolic execution engine for Erlang programs. SCV [189, 143] is a static contract verifier for Racket based on symbolic execution. Racket and Erlang are strict languages, and thus neither of the above tools considers lazy evaluation, which requires a different approach as demonstrated in Section 4.2. Further, to our knowledge, neither of the
above scales to check inductive properties (e.g. size, height) of recursive datatypes (e.g. lists, trees).

** Solver-aided Programming ** Solver-aided programming is a paradigm that makes it easier to write code that uses a constraint solver. As opposed to G2, which focuses on finding assertion violations, solver-aided programming allows directly manipulating symbolic values in code.

ROSETTE [190, 191] is a general purpose framework that enables solver-aided programming in Racket. This makes it easy to use Racket to formulate search problems over a symbolic domain. Programs can be written with traditional Racket code, but ROSETTE introduces symbolic integer and Boolean values. Unlike G2, ROSETTE does not attempt to find assertion violations in code. Rather, it gives programmers a higher level interface to constraint solvers, simplifying the writing of tools that manipulate symbolic values.

SmtEn [194] is a plugin for GHC that allows for high level constraint solving. A user can call SmtEn’s API to manipulate symbolic values. Similarly to ROSETTE, SmtEn gives a higher level interface to the capabilities of constraint solvers. It is therefore better suited than G2 for programmers who want to make use of constraint solving in their programs. However, SmtEn does not symbolically execute general purpose Haskell code, which makes G2 more usable as an off-the-shelf debugging aid.

G2Q, an application of G2 which will be discussed in Chapter 3, implements a solver-aided programming system using G2. A discussion of G2Q compared to ROSETTE and SmtEn is presented in Section 3.6.

** Symbolic Execution for Imperative Languages ** There are many symbolic execution engines for imperative languages, including Dart [86] and Cute [179] for C, Symbolic Pathfinder [155] for Java, Pex [188] for .NET, Sage [87] for x86 Windows applications, and EXE [47] and its sequel Klee [46] for LLVM. The execution semantics of imperative programs are quite different from ours. However, certain techniques, such as path explosion mitigation strategies [43, 107, 101, 84, 193, 50] and constraint solving strategies [77, 153, 109, 157, 54], are likely to be applicable to symbolic execution of functional languages.
3.1 Introduction

The advancements in constraint solvers, such as integer linear programming and SMT solvers, have enabled a range of new programming language tools. Such solvers have brought tackling previously intractable NP-hard problems into the realm of practicality. In particular, SMT solvers have been applied to a wide variety of challenges, including tools that strengthen the Haskell type system [68, 196, 197, 195, 149], test Haskell functions [178], verify DSL programs [75], and synthesize code [164].

Unfortunately, using Haskell to interact with a SMT solver requires a significant amount of engineering effort. When applying SMT solvers on some formula, it is frequently the case that variables, predicates and functions appearing in that formula require two representations: one in a traditional programming language, and one in the language of SMT solvers via the SMT-LIB format [40]. Additionally, one also needs to develop a parser which translates from each representation to the other. Furthermore, one of the most ubiquitous means of communication is via textual representation, which offers no type safety. These issues are compounded by the fact that, often, a direct translation of a problem is not enough. SMT solvers are sensitive to the encoding scheme [15], and it often requires several iterations to arrive at the best translation of a problem to a formula or formulas. In the process of this iteration it is – naturally – easy to introduce bugs and mistranslations [146, 17].

To make this more concrete, consider the following scenario: our goal is to write a function, \texttt{sumToN}, which takes as input two variables: \texttt{n}, an \texttt{Int}, and \texttt{xs}, a list of \texttt{Ints}. The function \texttt{sumToN} needs to return a non-empty list of \texttt{Ints} \texttt{ys} such that the sum of all elements of \texttt{ys} is \texttt{n}, and every element in \texttt{ys} also appears in \texttt{xs}.

One way to approach this problem would be to make use of a SMT solver. Figure 3.1 contains an encoding of this problem in the SMT-LIB format. The encoding can be seen as a template describing the above: we first represent a list datatype in
(declare-datatypes (T)
  ((list nil (cons (head T) (tail list)))))

(define-fun-rec sum ((zs (list Int))) Int
  (ite (is-nil zs)
       0
       (+ (head zs) (sum (tail zs)))))

(define-fun-rec length ((zs (list Int))) Int
  (ite (is-nil zs)
       0
       (+ 1 (length (tail zs)))))

(define-fun-rec elem
  ((z Int) (zs (list Int))) Bool
  (ite (is-nil zs) false
       (ite (= z (head zs)) true
            (elem z (tail zs)))))

(declare-const xs (list Int))
(declare-const ys (list Int))
(declare-const n Int)

(assert (= (sum ys) n))
(assert (forall ((y Int))
           (implies (elem y ys) (elem y xs))))
(assert (>= (length ys) 1))
(assert (= xs XS))
(assert (= n N))

(check-sat)
(get-model)
Out: Unknown

Figure 3.1: Finding the solution to \texttt{sumToN} using a direct encoding to a SMT solver.

the SMT-LIB format, next we define functions to sum the elements of the list, we then check if an element is in a list, and finally we calculate the length of the list (to check if the list is non-empty). Finally, a SMT solver is invoked at runtime when the values of \(n\) and \(xs\) are known. We therefore have two more assertions, containing the variables \(XS\) and \(N\), which are instantiated with concrete values at runtime.

Analyzing the code in Figure 3.1 we see that we had to duplicate much of what already exists in Haskell: the list datatype, and three functions to manipulate and examine it. Furthermore, in order to call this code with lists from a Haskell program, we still need to write code to translate a Haskell list to a SMT list. After completing all these tasks, unfortunately our efforts were fruitless: even for very simple values of \(xs\) and \(n\) (for example \(xs = [-5, 5]\) and \(n = 0\)), when the formula is passed to a state-of-the-art SMT solver, Z3 \[64\], \texttt{Unknown} is returned, meaning it could neither find a value for \(ys\) nor determine if such a value exists. Our experimental evaluations found that Z3 cannot find a solution as soon as \(xs\) has two or more elements due to difficulties that SMT solvers have with handling quantified formulas and recursive
To address all these problems, we introduce G2Q, a new library that defines the \texttt{g2} quasiquoter to simultaneously empower and simplify the solving of complex constraints. A quasiquoter \cite{133} is a way of using metaprogramming to embed a domain specific language (DSL) into Haskell. At compile time, the code encapsulated in the quasiquoter is automatically translated into traditional Haskell code using Template Haskell metaprogramming \cite{180}.

The \texttt{g2} quasiquoter, \texttt{[g2]}…\texttt{]}], allows Haskell programmers to write constraints in a flexible, type-safe language: Haskell itself. Programmers do not need to concern themselves at all with the low level details of external constraint solvers. Rather, the library’s quasiquoter allows a programmer to write a predicate using traditional Haskell syntax and Haskell functions while making use of concrete (runtime determined) variables, and \textit{symbolic} (unknown) variables. The quasiquoter generates code that will at runtime accept the concrete arguments and return either (1) \texttt{Nothing} if no values for the symbolic variables that will satisfy the predicate are found or (2) \texttt{Just} values for the symbolic variables.

Figure 3.2 contains the aforementioned \texttt{sumToN} problem, written using our quasiquoter. The quasiquoter takes two concrete arguments, \texttt{xs} and \texttt{n}, and returns an \texttt{IO (Maybe [Int])} – a value for \texttt{ys}, if one exists. The function returns in the \texttt{IO} monad because G2Q’s constraint solving may be non-deterministic.

Obviously, the quasiquoter code is significantly shorter and also allows us to reuse the existing Haskell datatypes and functions. There is another key advantage to using our library: G2Q is not simply bindings to a SMT solver. Rather, under the hood, G2Q makes use of the Haskell symbolic execution engine G2, as described in Chapter 2. By using symbolic execution, we can reduce the Haskell predicates to constraints over the symbolic inputs, and then solve those constraints with a SMT solver. Function unrolling allows G2Q to solve predicates making use of recursive functions and datatypes. As a result, the \texttt{g2} quasiquoter can actually solve many problems that are not solvable with more direct SMT encodings. For instance,
G2Q is capable of handling inputs to `sumToN` that challenge SMT solvers. Running `sumToN 0 [-5, 10, -15, 20, 25]` outputs a valid solution: `Just [20, -15, -5].`

At compile time, the `g2` quasiquoter converts constraints from Haskell code into G2’s intermediate representation. Furthermore, it instruments the code with functions to translate input-output between their actual values and value representations in G2’s intermediate language. As functions to perform these conversions are defined in a derivable typeclass, the details of this translation is hidden from users of the G2Q library.

To evaluate G2Q, we wrote four programs using it. These programs demonstrate a range of use cases ranging from an n-queens problem solver to a program analyzer. They also demonstrate a spectrum of complexity, suggesting possible ways G2 could be improved and optimized in the future.

In short, we make the following contributions:

1. We describe our library, which provides a quasiquoter to give programmers access to the capabilities of constraint solving via writing Haskell predicates. In addition, we describe the quasiquoter’s *strictness* and *fairness* properties, which govern how the quasiquoter handles infinitely large data structures, and searches over infinitely large sets of values.

2. Behind the scenes, the quasiquoter is using a Haskell symbolic execution engine, G2, to reduce the user-written Haskell code to constraints that are solvable by SMT solvers. We describe how we compile a quasiquoter to a form that is runnable in G2.

3. We show code for a number of case studies, demonstrating a variety of use cases of our library. In the next section, we will describe a technique to easily convert a program executor to a program analyzer. In Section 7.5, we will present three additional use cases.

### 3.2 G2Q for Program Analysis

Haskell is frequently used to implement programming languages and DSLs [75, 73, 32]. Here, we consider a simple imperative language with support for basic arithmetics as shown in Figure 3.3.

The language supports *assertion* statements, a common technique for performing sanity checks and error detection during software development. The `evalStmts` function is responsible for running a program. It accepts an `Env` – which maps variables to values, as an input – and also allows the caller to specify initial values. `evalStmts` and its subfunctions either return `Just` some type if they succeed, or `Nothing` if an assertion is violated.

Although this language and its interpreter are rather small, it suffices to represent many large imperative programs. Without specialized tooling and engineering overhead, it can be difficult to tell if and how an assertion will fail.
Consider, for instance, the program in Figure 3.4. This program contains an assertion that claims that upon completing execution, the assertion

\[(\text{Lt} (\text{Mul} (\text{Var} \ "n") (\text{I} 2)) (\text{Var} \ "z"))\]

will hold. From just manually examining the program, it is not immediately clear whether inputs exists that violate this assertion. Of course, one could rely on testing, but such approaches still require picking the correct values to violate an assertion.

G2Q provides an easy way for the language developer to find assertion violations through a *symbolic search* over the space of inputs by leveraging the existing evalStmts function. The developer can simply write the following function:

```haskell
badEnvSearch :: Stmts -> IO (Maybe Env)
badEnvSearch = \[g2 | (stmts :: Stmts) -> ?(env :: Env) |
    evalStmts env stmts == Nothing \]
```

`badEnvSearch` takes a concrete `Stmts` as an arguments, and searches for an `Env` that causes `evalStmts` to evaluate to `Nothing`, thereby indicating an assertion violation.

We can run this on the program, to see if it can find an assertion violation. The call:

```haskell
env <- badEnvSearch prog
putStrLn $ show env
```

returns `Just ["j", -18]`, revealing that an assignment of \(j = -18\) will lead to an assertion violation. Notably, no random testing occurred to land on the value \(-18\). Rather, constraints were generated from `evalStmts` and solved to determine that \(j = -18\) would violate an assertion.
type Ident = String

type Env = [(Ident, Int)]

type Stmts = [Stmt]

data AExpr = I Int | Var Ident
        | Add AExpr AExpr | Mul AExpr AExpr
        deriving (Show, Eq, Data)

$(derivingG2Rep ''AExpr)

data BExpr = Not BExpr
        | And BExpr BExpr | Or BExpr BExpr
        | Lt AExpr AExpr | Eq AExpr AExpr
        deriving (Show, Eq, Data)

$(derivingG2Rep ''BExpr)

data Stmt = Assign Ident AExpr | Assert BExpr
        | If BExpr Stmts Stmts | While BExpr Stmts
        deriving (Show, Eq, Data)

$(derivingG2Rep ''Stmt)

evalA :: Env -> AExpr -> Int
evalA = ...

evalB :: Env -> BExpr -> Bool
evalB = ...

evalStmt :: Env -> Stmts -> Maybe Env
evalStmt e (Assign ident aexpr) =
    Just $ (ident, evalA e aexpr) : e

evalStmt e (If bexpr lhs rhs) =
    if evalB e bexpr
    then evalStmts e lhs
    else evalStmts e rhs

evalStmt e (While bexpr loop) =
    if evalB e bexpr
    then evalStmts e (loop ++ [While bexpr loop])
    else Just e

evalStmt e (Assert bexpr) =
    if evalB e bexpr then Just e else Nothing

evalStmts :: Env -> Stmts -> Maybe Env
evalStmts = foldM evalStmt

Figure 3.3: Simple arithmetics language
prog :: Stmts
prog =
[ Assign "k" (I 1),
  Assign "i" (I 0),
  Assign "n" (I 5),
  While (Or (Lt (Var "i") (Var "n")))
    (Eq (Var "i") (Var "n")))
  [ Assign "i" (Add (Var "i") (I 1))],
  Assign "z" (Add (Var "k")
    (Add (Var "i") (Var "j"))),
  Assert (Lt (Mul (Var "n") (I 2)) (Var "z")) ]

Figure 3.4: A program inspired from [69] written with the simple arithmetic language shown in Figure 3.3. It accepts a variable "j" as input, and checks an assertion at its end.
3.3 Solver-Aided Interface

In this section, we present the exposed API of G2Q, which enables Haskell solver-aided programming. We begin with a description of the core of G2Q: the g2 quasiquoter. Using this quasiquoter, programmers can write Haskell predicates over symbolic (unknown) variables and automatically find concrete values that satisfy the predicate. We then briefly discuss the G2Rep typeclass, which is required to lift values to and from the quasiquoter. Finally, we discuss the strictness and fairness guarantees offered by our library.

3.3.1 The g2 Quasiquoter

The principal feature of G2Q is the g2 quasiquoter which, as shown in the grammar in Figure 3.5, uses a slightly edited version of standard Haskell syntax: concrete arguments $x_1 :: \tau_1 ... x_m :: \tau_m$ are bound by a lambda expression; symbolic variables $s_1 :: \tau^s_1 ... s_n :: \tau^s_n$ are then specified. Finally, a predicate $e$ is written over the full set of variables.

The quasiquoter generates a function of type:

$$\tau_1 \to ... \to \tau_m \to \text{IO} (\text{Maybe} (\tau^s_1 ... \tau^s_n))$$

At runtime, this function sets the concrete arguments in the predicate $e$ to the values passed by the user. Next, the backend attempts to find satisfying instantiations of $s_1 ... s_n$. If it succeeds, Just a tuple of the found values is returned. Otherwise, Nothing is returned. Note that there is no guarantee that the backend is deterministic, and as such, the value is returned in the IO Monad.

3.3.2 The G2Rep Typeclass

The types of all concrete arguments and symbolic inputs in a g2 quasiquoter are required to be instances of the G2Rep typeclass which we further describe in Section 3.4.1. This G2Rep typeclass is defined by G2Q, to allow lifting instances to and from the representation required by the g2 quasiquoter. Defining an instance of G2Rep manually requires knowledge of the internals of G2Q. To allow programmers to easily use their own first-order datatypes with the g2 quasiquoter, we provide derivingG2Rep, a TemplateHaskell function to automatically derive instances of G2Rep with a single line of code. derivingG2Rep requires only an instance of the Data typeclass (which is also derivable, using the DeriveDataTypeable language extension), and for the ScopedTypeVariables language extension [111] to be turned on (for reasons discussed in Section 3.4.1).

3.3.3 Strictness and Fairness

Here, we discuss the strictness and fairness properties of the g2 quasiquoter.
QQ ::= λ x₁ ... xₘ → s₁ ... sₙ | e
x ::= (y :: τ)  \text{concrete argument}
s ::= ?(y :: τ)  \text{symbolic variable}

Figure 3.5: The grammar accepted by G2Q. y and τ represent standard Haskell variables and types, respectively. e represents a standard Haskell expression, which must be of type \texttt{Bool}.  

(a) \[ g2 \mid \forall (xs :: [\text{Int}]) \rightarrow
? (x :: \text{Int}) \mid x == \text{head} \ xs \] [1..]

Out: \textbf{Just 1}

(b) \[ g2 \mid \forall (xs :: [\text{Int}]) (t :: \text{Int}) \rightarrow
? (ys :: [\text{Int}])
\mid ys == \text{take} t \ xs \] [1..] 4

Out: \textbf{Just} [1, 2, 3, 4]

(c) \[ g2 \mid \forall (xs :: [\text{Int}]) \rightarrow ? (y :: \text{Int})
\mid \text{head} \ xs > y \&\& y > 0 \] [0..]

Out: \textbf{Nothing}

Figure 3.6: Here, we show several examples of G2Q’s behavior on infinite lists, for which the quasiquoter will give output. The key requirement is that the predicate require evaluation of only a finite amount of the infinite input.

\textbf{Strictness}

Strictness refers to the reduction order of an expression during program execution. The \texttt{g2} quasiquoter preserves Haskell’s lazy evaluation semantics [161].

\textbf{Infinite Data Structures}  As may be expected, lazy evaluation allows the \texttt{g2} quasiquoter to both consume and produce infinite data structures. When consuming infinite data structures, the quasiquoter must be able to fully evaluate the predicate after evaluating only a finite portion of the structure. Figure 3.6 shows several examples where a quasiquoter can terminate on infinite input, because finding a correct output requires only a finite portion of the input. Figure 3.7 shows two examples that will not terminate because there is no bound on the amount of the input that must be evaluated.

Finally, Figure 3.8 shows a quasiquoter that produces an infinite data structure. Similarly to the input, such quasiquotes return if and only if checking the correctness of the predicate requires evaluating only a finite amount of the output.
Figure 3.7: Here, we show two examples of G2Q’s behavior on infinite lists, that will result in non-termination.

Figure 3.8: Here, we show two examples of \( g2 \) quasiquoter’s, with an output type that is an infinite data structure. When only a finite amount of the output has to be evaluated to check the predicate, the quasiquoter can return such an infinite data structure. However, trying to satisfy a predicate that requires evaluating an infinite amount of the infinite data structure results in non-termination.

**Recursive Functions**  G2Q allows arbitrary Haskell code, including the use of recursive functions. While this can be quite powerful, it also means care must be exercised if the input to a recursive function is symbolic. When executed in a \( g2 \) quasiquoter on symbolic values, recursive unrollings of functions can lead to non-termination even if the function is guaranteed to terminate when normally executed.

To see why, consider the code in Figure 3.9. \( \text{mult} \) is simply an implementation of multiplication based on repeated addition, and will always terminate. However, the \( g2 \) quasiquoter is searching for an integer \( n \) such that \( \text{mult} n 3 = 10 \). Of course no such integer exists, so the predicate is unsatisfiable. However, the recursive search over the symbolic variable will result in a deeper and deeper search to find such an \( n \), resulting in non-termination.

It follows from the halting problem that any automatic approach to prevent this
mult :: Int -> Int -> Int
mult n x
  | n == 0 = 0
  | n >= 0 = x + mult (n - 1) x
  | otherwise = mult (n + 1) x - x

\[ g2 \mid (x :: Int) \rightarrow \]

\(? (n :: Int) \mid \text{mult} n x == 10 \mid 3 \]

Out: \perp

**Figure 3.9:** A quasiquoter that will fail to terminate, because of an unsatisfiable predicate involving a recursive function.

kind of error would, unfortunately, rule out at least some good programs. Given that Haskell itself does not prove termination, we therefore leave it up to programmers to ensure their `g2` quasiquoters terminate. To prevent non-termination, it is sufficient to ensure that either (1) no recursive function call’s termination depends on a symbolic variable, or (2) whenever a recursive function call depends on a symbolic variable, the predicate is satisfiable.

**Fairness**

We offer two *fairness guarantees* to users of G2Q. Here, we present the minimal information needed for users of G2Q. In Section 3.4.3, we will revisit these guarantees, and provide justifications. Both are relative to the completeness of the underlying SMT solver. That is, they are true to the extent that G2’s underlying SMT solver is able to answer every query correctly:

1. In a predicate with no recursive function calls or let bindings, if the predicate is unsatisfiable we will eventually return `Nothing`.

2. If the `g2` quasiquoter’s predicate will evaluate to `True` given some instantiation of the symbolic variables, G2Q will eventually return a solution.

In Section 3.4.3, we will return to and justify these fairness guarantees.

### 3.4 Solver-Aided Backend

The backend of G2Q relies on an existing Haskell symbolic execution engine, as described in Chapter 2. Here, we elaborate on the design of G2Q’s other components. Then, we state some limitations of our approach and how they may be alleviated in the future.
class G2Rep g where
  g2Rep :: g -> G2Expr
  g2UnRep :: G2Expr -> g
  g2Type :: g -> G2Type

Figure 3.10: The G2Rep typeclass, which converts values to and from G2’s representation.

class G2Rep a => G2Rep [a] where
  g2Rep [] = g2Nil (g2Type (undefined :: a))
  g2Rep (x:xs) =
    g2Cons (g2Type (undefined :: a))
    (g2Rep x) (g2Rep xs)
.
Figure 3.11: The g2Rep definition for lists. We denote the standard Haskell list constructors as : and [], and G2’s representation of a list as g2Cons and g2Nil, respectively.

3.4.1 The G2Rep Typeclass

A slightly simplified version of the G2Rep typeclass is shown in Figure 3.10 (some types from G2 that do mundane mapping have been hidden, to simplify the presentation.) It includes three functions: g2Rep, g2UnRep, and g2Type. The g2Rep and g2UnRep functions map from real Haskell values to G2’s representations of those values and back. g2Type is a helper function for g2Rep and g2UnRep: polymorphic type arguments are explicitly represented in G2’s core language, and so we require g2Type to give us access to the G2 representation of the type of polymorphic arguments.

Figure 3.11 shows part of the definition of G2Rep for lists. For the most part, the mapping is very routine – the sole point of interest is the use of g2Type. Sometimes (as in the expression for a nil list constructor) we require calling the instance of g2Type for a type of which we do not have a value. Fortunately, we can insist that undefined is a value of any type and use it to call the appropriate g2Type. This does require the ScopedTypeVariables language extension to be enabled, so that the type variable in the instance declaration and the type variable in the function body are bound to the same type. If a programmer tries to deriving G2Rep without enabling the extension, we show a error message, reminding them to turn it on.

3.4.2 g2 Quasiquoter Compilation

We now describe the translation of a g2 quasiquoter into a Template Haskell expression.

Imported Modules  G2Q allows making use of functions from imported modules, as long as the source code is available for G2 to compile into its internal representation. We use Template Haskell to pull the list of imported modules from the current file.
and use a cabal file [110] to search for them. We also utilize a custom version of the standard Haskell Base and Prelude [158], which supports many of the commonly used types and functions (and which we are working on expanding).

**Parsing**  G2Q accepts a lightly modified version of traditional Haskell syntax. The only differences are:

1. We require type annotations to be given in the concrete variable lambda binding.
2. We introduce a new notation to specify symbolic variables.

Our parser extracts the concrete variables along with their types $x_1 :: t_1 \ldots x_m :: t_m$, the symbolic variables and types $s_1 :: t_{s1} \ldots s_n :: t_{sn}$, and the predicate expression $e$ from the $g2$ quasiquote. Then, it rewrites the user’s query as a Haskell predicate function with both the concrete and symbolic variables bound by lambda expressions, in addition to an explicit type signature:

\[
\text{pred} :: t_1 \to \ldots \to t_m \to \ldots \to t_{s1} \to \ldots \to t_{sn} \to \text{Bool}
\]

\[
\text{pred} \; x_1 \ldots x_m \; s_1 \ldots s_n = e
\]

We then use G2’s existing parser (which itself makes use of GHC’s parser) to translate this traditional Haskell code into G2’s internal representation.

**State Construction**  We construct a state $s = (E, H, P)$. $H$ is a heap containing functions from imported modules, and $P$ is initialized to empty. We initialize $E$ to:

\[
\text{let } r = \text{pred} \; x_1 \ldots x_m \; s_1 \ldots s_n \\
\text{in } \text{assume} \; r \; (s_1, \ldots, s_n)
\]

$x_1$ to $x_m$ are placeholders for the concrete arguments (which will be replaced in the next step), $\text{assume} \; p \; e$ assumes the predicate $p$ holds and then returns $e$. Thus, the code in $E$ will force the quasiquoter’s predicate to hold, and if it does, return a tuple of the values of the symbolic variables.

**Argument Bindings**  For each concrete argument $x_1 \ldots x_m$ we construct a Template Haskell expression that will bind $g2_xi$ to $g2\text{Rep} \; xi$ at runtime. Then, we construct a runtime call to a function named $\text{floodConsts}$, which receives the state $s = (e, h, p)$ and a list of the $g2_xi$ as arguments. $\text{floodConsts}$ lazily (so as to not force too much of the input values) replaces each concrete argument in the states expression $e$ with the $g_xi$ from the list.

**Solving Symbolic Variables**  Given a state with the concrete arguments filled in via $\text{floodConsts}$, G2 is able to symbolically execute the state at runtime. Assuming it terminates (as discussed in Section 3.3.3), G2’s constraint solving finds concrete values for the symbolic variables and returns a tuple of the found solutions. We then use $\text{g2UnRep}$ to lazily translate the tuple from G2’s representation into regular Haskell values. $\text{g2UnRep}$’s laziness allows returning infinite data structures when needed, as discussed in Section 3.3.3.
Type Safety We use Template Haskell to, at compile time, wrap each input to \texttt{g2Rep} and call to \texttt{g2UnRep} with an explicit type annotation. These annotations provide programmers with type errors if they try to mistype an argument or the returned value.

3.4.3 Fairness and Heuristic Search

As discussed in Section 3.3.3, G2Q offers two fairness guarantees. We justify theses fairness guarantees here and then discuss some heuristics we implement as well as how those heuristics preserve the fairness guarantees.

Fairness Guarantees

We begin by presenting and justifying our fairness guarantees. Both guarantees are relative to the completeness of the underlying constraint solver.

**Guarantee 1** First, we guarantee that, if the predicate in the quasiquoter contains no recursive function calls or recursive let bindings, and is unsatisfiable, the quasiquoter will eventually terminate by returning \texttt{Nothing}. This can be trivially seen from an examination of the reduction rules used by G2, as shown in Chapter 2. The only possible source of an infinite loop is a recursive call, since all other reduction rules reduce the size of the expression being evaluated and thus will lead to termination. Therefore, in the absence of a recursive function call or recursive let binding we can fully explore the set of possible states and return \texttt{Nothing} if all are unsatisfiable.

**Guarantee 2** The second guarantee is that, if there is some instantiation of the symbolic variables such that the predicate in the quasiquoter \( q \) will evaluate to true, some solution will eventually be returned.

To ensure this, it is sufficient to show that:

1. Whenever we hit a branch in the code, we create states to explore along each possible branch.

2. If some state exists, and it has not yet fully executed to \texttt{True} or \texttt{False}, either the state will eventually be executed, or some other state will be executed that returns a solution.

(1) can be seen from an examination of the reduction rules, in Chapter 2. The sources of branching are limited, all case expression branches are initialized as separate states.

(2) is trickier, as it requires us to fairly evaluate all states. Otherwise, we might only evaluate some subset of states where the predicate is false and miss some state where the predicate is true. To ensure that (2) holds, we fix a predicate \( p \) that we
know any state we execute will eventually violate (an example of such a guarantee is given below, in Section 3.4.3.) We then store the states in a queue.

During symbolic execution, we pop the state at the head of the queue. We symbolically execute this state either until we discover that it satisfies the quasiquoter’s predicate \( q \) or until it has violated the predicate \( p \). If the state is not yet fully evaluated but \( p \) becomes false, we re-insert the state at the end of the queue. If the state splits at a branch, we arbitrarily choose one of the states to continue executing. The others get inserted at the end of the queue.

With this scheme we can see via a classic argument that all states not yet fully evaluated will eventually be executed (unless another state that satisfies \( q \) is found first). At any point such a state \( s \) is in the queue. Suppose there are \( s\# \) states ahead of the \( s \) in the queue. Since we (at least temporarily) halt execution of every state eventually, either one of those \( s\# \) states will turn out to be a solution or \( s \) will eventually be executed.

**Symbolic Execution Heuristics**

A challenge in making symbolic execution effective for finding solutions are the heuristics employed. In particular, because symbolic execution tracks multiple states at once, yet (outside of parallelization) only one state is executed at a time, the order in which states are chosen for reduction is crucial. A bad ordering causes an increase in the time required to find a satisfying solution to the predicate.

G2Q employs a heuristic that prioritizes states with fewer symbolic variables. The intuition is that such states will (often) lead to fewer new path constraints – resulting in cheaper calls to the SMT solver, and reduced future state splitting.

**Preserving Fairness**  As described in the previous section, our fairness guarantee depends on a queue, with some predicate \( p \) that will eventually be violated. To ensure that we violate the \( p \) with this heuristic, we choose \( p \) to be that the state (1) contains fewer symbolic variables than the state at the head of the queue, and (2) has increased its symbolic variable count in the last \( k \) steps (for some fixed \( k \)).

If condition (2) is consistently met, then eventually condition (1) will be violated, and the state will be sent to the back of the queue. Otherwise, condition (2) will send the state to the back of the queue. Thus our predicate is sufficient to guarantee fairness.

**3.4.4 Limitations**

**Scoping and Module Imports**  A \texttt{g2} quasiquoter requires that all functions and datatypes used in it are defined in an imported module, and \textit{cannot} use functions or datatypes declared in the same module. This is because in order to perform symbolic execution, G2 has to be able to compile the code in the quasiquoter – and the code’s
dependencies – to G2’s internal representation. Trying to compile the module the
quasiquoter is in would result in an infinite loop.

**Argument Types** Within a quasiquoter, G2Q requires that type signatures be
provided (rather than inferred) for the concrete and symbolic arguments to the \texttt{g2}
quasiquoter. In addition, we require that such types be monomorphic and are also
first-order. It should be stressed that these restrictions apply only to the quasiquote
arguments. Within the rest of the predicate’s code, polymorphic and higher order
functions may be used.

The source of each of these limitations is in fact the same. Currently, G2Q relies
on having access to all the code used in \texttt{g2} quasiquoter available at compile time, so
that it can compile the code into G2’s intermediate representation. If a programmer
tries to use some library that G2Q cannot access the code for, an error is given at
compile time. Allowing passing arbitrary higher order functions would require G2Q to
have some way of dynamically converting the passed function to G2’s representation
at runtime. Typeclasses are, in the internals of both GHC and G2, just a dictionary
of higher order functions [93] and thus present the same issue.

To simplify for end users and hopefully prevent confusion, we disallow polymor-
phism entirely since we cannot support typeclasses. It is possible that this decision
is overly conservative, and if that proves to be the case, we could relax the restric-
tion to allow limited polymorphism in the future. However, we want to get further
experience using G2Q before making this decision.

**Base Support** In order to make use of datatypes and functions from Base, G2Q
requires them to be compiled into our intermediate language. As compiling Base is a
complicated process which relies closely on GHC, G2 currently make use of a custom
Base library. As such, G2, and by extension G2Q, currently supports only a subset
of functions and datatypes.

In the future, we plan on expanding this subset. In addition, we plan to investigate
means by which we could compile the whole of Base (such as instrumenting GHC to
write out our intermediate language, for example).

### 3.5 Evaluation and Case Studies

We have made G2Q available on Hackage at \url{http://hackage.haskell.org/package/g2q}. Here, we present case studies and an evaluation, showing how G2Q can be used
to solve a variety of problems.

#### 3.5.1 Programming by Example in Lambda Calculus

Programming by example is a paradigm that allows code to be synthesized from
input-output examples.
Consider a lambda calculus language and evaluator based on De Bruijn indexing [63] as shown in Figure 3.12. De Bruijn indexing eliminates variable names, by writing bound variables as an integer that indicates the number of lambdas between the variable and its binder.

The evaluator function $\text{eval}$ for the lambda calculus is standard: it simplifies a lambda expression as much as possible and then outputs it to the user.

Inspired by the programming by example paradigm [130, 117], one can use G2Q to not only execute lambda calculus expressions but to also synthesize expressions based on input-output examples.

To do this, we may write a function as follows:

$$\text{solveDeBruijn} :: \{[[\text{Expr}], \text{Expr}]\} \rightarrow \text{IO} \ (\text{Maybe Expr})$$

$$\text{solveDeBruijn} = \[g2| \ (\text{es :: }[[\text{Expr}], \text{Expr}]) \rightarrow ?(\text{func :: Expr}) \ |
\qquad \text{all} \ (\text{e} \rightarrow \text{eval} \ (\text{app} \ (\text{func} : \text{fst} \ \text{e}))) \ == \ \text{snd} \ \text{e} \ \text{es}] \]$$

The $\text{solveDeBruijn}$ function takes an input list of pairs of the form (arguments, result). The goal is to then synthesize a new function $\text{func}$ such that when all the arguments are applied to $\text{func}$, the result yielded is result.

As a simple example consider the Haskell $\text{const}$ function, which takes two arguments and returns the first unmodified. By writing the function call:

$$\text{solveDeBruijn} \ [ \ ([\text{num 1}, \text{num 2}], \text{num 1}) 
\qquad , \ ([\text{num 2}, \text{num 3}], \text{num 2}) \]$$

we can synthesize a lambda expression with this effect:

$$\text{Lam} \ (\text{Lam} \ (\text{Var} \ 2))$$

Somewhat less trivially, we can use the Church encoding of Booleans [38] to synthesize Boolean functions. In Church encoding we denote $\text{True}$ as $\text{Lam} \ (\text{Lam} \ (\text{Var} \ 2))$ and $\text{False}$ as $\text{Lam} \ (\text{Lam} \ (\text{Var} \ 1))$.

Using these definitions, we can write examples for Boolean functions, such as or:

$$\text{solveDeBruijn} \ [ \ ([\text{trueLam}, \text{trueLam}], \text{trueLam}) 
\qquad , \ ([\text{falseLam}, \text{falseLam}], \text{falseLam}) 
\qquad , \ ([\text{falseLam}, \text{trueLam}], \text{trueLam}) 
\qquad , \ ([\text{trueLam}, \text{falseLam}], \text{trueLam}) \]$$

and synthesize correct definitions for those functions:

$$\text{Lam} \ (\text{App} \ (\text{Var} \ 1) \ (\text{Var} \ 1))$$

### 3.5.2 $n$-Queens

A mathematical puzzle called the $n$-queens problem asks how $n$ queen pieces may be placed on an $n \times n$ chess board such that no two queens threaten each other [170]. That is, no two queens may be in the same row, column, or diagonal. We demonstrate how this problem may be solved with G2Q via an encoding in Figure 3.13.

Since no two queens may be in the same row and we have $n$ queens to be placed
in $n$ rows, there is clearly a queen in every row. Thus, we represent the set of queens by a list of Int’s, where the Int $j$ in the $i$th position in the list, indicates there is a queen at $(i, j)$. allQueensSafe checks if the given list of Queens is a valid solution to the $n$-queens problem. Specifically, it checks if the list is the correct length, that all the queens on in legal positions, and that none of the queens can attack each other.

A classic version of this problem is for a traditional chessboard with $n = 8$. The solution that G2Q produces for the 8-queens problem is shown in Figure 3.14.

### 3.5.3 Regular Expressions

Borrowing an example from SmtEn [194], suppose a user has written a regular expression implementation as a domain specific language. The implementation includes a `match` function which, given a regular expression and a string, returns whether the string matches the regular expression.

Provided a regular expression written in such a DSL, it may be helpful to search for examples of strings that are accepted by this regular expression. Such a problem is examined in SmtEn, but requires the user to also implement several functions to assist the search. We demonstrate how G2Q and `match` can be used to solve this problem, with very little additional code required from the user. We defer further discussion of SmtEn to Section 3.6.

First, we augment the `RegEx` algebraic data type with a `Data` derivation, as well as a `derivingG2Rep`. Other parts of the code may be left untouched.

```haskell
data RegEx =
  Empty -- The empty language
  | Epsilon -- The empty string
  | Atom Char |
  | Star RegEx |
  | Concat RegEx RegEx |
  | Or RegEx RegEx |
  deriving (Show, Eq, Data)
$(derivingG2Rep '' RegEx)

match :: RegEx -> String -> Bool
match = ...
```

SmtEn’s implementation of regular expressions includes both `Epsilon` to denote the empty string, as well as `Empty` for the empty language. Next, we encode a query into the `g2` quasiquoter.

```haskell
stringSearch :: RegEx -> IO (Maybe String)
stringSearch =
  [g2] \(r :: RegEx) -> ?(str :: String) |
  match r str |
```

Finally, we can call this search function with a regular expression:

```haskell
-- (a + b)* c (d + (e f)*)
stringSearch $(
  Concat (Star (Or (Atom 'a') (Atom 'b'))))
```

37
G2Q is able to successfully find a string – Just "cd" – matched by this regular expression.

3.5.4 Evaluation

While writing the case studies we discovered two key factors that affect performance: predicate order in the quasiquoter and state explosion. Here we discuss both these factors and then address the runtimes of our case studies.

Predicate Order  Simple changes to the predicate can have a dramatic affect on runtime. For example, consider allQueensSafe’ in Figure 3.16. This function is the same as allQueensSafe from Figure 3.13, except that the constraint on the length of the list has been moved from being the first conjoined constraint to the last. However, solving the 8-queens problem with allQueensSafe takes only 2.30 seconds while solving it with allQueensSafe’ takes 36.42 seconds. This is because constraining the length of the list allows quickly filtering out many states where the list is either too long or too short. In future work it would be valuable to explore ways of automatically reordering predicates to optimize performance.

State Explosion  Programs with large amount of branch can cause symbolic execution to suffer from state explosion, in which the number of states the symbolic execution engine generates and must evaluate grows exponentially. In particular, G2’s current handling of symbolic algebraic datatypes can cause it to branch into many states. In future work we hope to implement state merging in order to reduce the number of states that must be individually evaluated.

Evaluation Results  Figure 3.15 shows evaluation results from our case studies and some other associated benchmarks. We ran all tests with a timeout of two minutes. Two of the tests did not terminate in this time. These timeouts are largely due to state explosion from G2’s handling of algebraic datatypes. In future work we hope to improve there performance by implementing state merging.

Despite the timeouts, we view these results as very positive. All our benchmarks involve programs with recursion and yet G2Q manages to solve eight of the benchmarks in under a second. In contrast, unassisted SMT solvers are known to struggle with recursion or loops [100]. Thus, even with the timeouts, our results indicate an improvement over direct encoding of SMT formulas.
type Ident = Int

data Expr = Var Ident
  | Lam Expr
  | App Expr Expr
deriving (Show, Read, Eq, Data)

$(derivingG2Rep ''Expr)$

type Stack = [Expr]

eval :: Expr -> Expr
eval = eval' []

eval' :: Stack -> Expr -> Expr
eval' (e:stck) (Lam e’) =
  eval’ stck (rep 1 e e’)
eval’ stck (App e1 e2) = eval’ (e2:stck) e1
eval’ stck e = app $ e:stck

rep :: Int -> Expr -> Expr -> Expr
rep i e v@(Var j)
  | i == j = e
  | otherwise = v
rep i e (Lam e’) = Lam (rep (i + 1) e e’)
rep i e (App e1 e2) =
  App (rep i e e1) (rep i e e2)

app :: [Expr] -> Expr
app = foldl1 App

num :: Int -> Expr
num n = Lam $ Lam $
  foldr1 App (replicate n (Var 2) ++ [Var 1])

Figure 3.12: De Bruijn index based lambda calculus and evaluator.
type Queen = Int

indexPairs :: Int -> [(Int, Int)]
indexPairs n = [(i, j) | i <- [0..n-1], j <- [i+1..n-1]]

legal :: Int -> Queen -> Bool
legal n qs = 1 <= qs && qs <= n

queenPairSafe :: Int -> [Queen] -> (Int, Int) -> Bool
queenPairSafe n qs (i, j) = let qs_i = qs !! i
                                qs_j = qs !! j
                                in (qs_i /= qs_j)
                                && qs_j - qs_i /= j - i
                                && qs_j - qs_i /= i - j

allQueensSafe :: Int -> [Queen] -> Bool
allQueensSafe n qs = (n == length qs)
                    && all (legal n) qs
                    && (all (queenPairSafe n qs) (indexPairs n))

Figure 3.13: N-Queens

8
7
6
5
4
3
2
1

a b c d e f g h

Figure 3.14: A solution to 8-queens produced by G2Q
<table>
<thead>
<tr>
<th>Task</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>badEnvSearch prog</td>
<td>59.32</td>
</tr>
<tr>
<td>Search for non-zero Mul x y == Add x y</td>
<td>0.56</td>
</tr>
<tr>
<td>solveDeBruijn for id</td>
<td>0.04</td>
</tr>
<tr>
<td>solveDeBruijn for const</td>
<td>1.06</td>
</tr>
<tr>
<td>solveDeBruijn for NOT</td>
<td>Timeout</td>
</tr>
<tr>
<td>solveDeBruijn for OR</td>
<td>86.22</td>
</tr>
<tr>
<td>solveDeBruijn for AND</td>
<td>Timeout</td>
</tr>
<tr>
<td>solveQueens 4</td>
<td>0.36</td>
</tr>
<tr>
<td>solveQueens 5</td>
<td>0.55</td>
</tr>
<tr>
<td>solveQueens 6</td>
<td>0.90</td>
</tr>
<tr>
<td>solveQueens 7</td>
<td>1.47</td>
</tr>
<tr>
<td>solveQueens 8</td>
<td>2.30</td>
</tr>
<tr>
<td>stringSearch for (a + b)<em>c(d + (ef)</em>)</td>
<td>0.26</td>
</tr>
<tr>
<td>stringSearch for abedef</td>
<td>4.23</td>
</tr>
<tr>
<td>stringSearch for a + b + c + d + e + f</td>
<td>0.05</td>
</tr>
<tr>
<td>stringSearch for a<em>b</em>c<em>d</em>e<em>f</em></td>
<td>0.02</td>
</tr>
</tbody>
</table>

**Figure 3.15:** Case study running times.

```haskell
allQueensSafe' :: Int -> [Queen] -> Bool
allQueensSafe' n qs =
    all (legal n) qs
    && (all (queenPairSafe n qs) (indexPairs n))
    && (n == length qs)
```

**Figure 3.16:** `allQueensSafe’` is the same as `allQueensSafe`, from Figure 3.13, except the constraint on the list length is moved to the end of the function. This change has a dramatic affect on running time. Solving 8-queens with `allQueensSafe` takes only 2.30 seconds, while using `allQueensSafe’` requires 36.42 seconds.
3.6 Related Work

Here we give an overview of related work, with a particular focus on work that aims to simplify the use of SMT solving in high level languages.

Solver-Aided Languages  
Like G2Q, SmtEn [194] is designed to ease using SMT solvers in Haskell. However, the interfaces provided by the two tools are quite different. SmtEn provides users with functions to build up a Space (that is, a set) and then use a SMT solver to search through the Space, for a value that satisfies some condition. In contrast, with our tool, the quasiquoter can be called directly, with no need to provide a set of possible instantiations.

In fact, both SmtEn and G2Q’s APIs have advantages. SmtEn’s treatment of Spaces as first class values allows them to be passed around and manipulated in code before being queried. On the other hand, as treated in SmtEn, a Space either has to be built from other Spaces, or constructed from scratch as a singleton. As such, constructing a Space results in a great deal of often tedious code, which can be avoided by our approach.

We see great potential in combining the approach of Spaces, (or some close equivalent) and our quasiquoter approach. For example, one could imagine a hybrid approach that uses a g2-like quasiquoter to construct Space-like values. We leave such considerations to future work.

Curry [99] is a logic driven functional programming language. Curry may be seen as an approach to design a functional language around logic programming. In contrast, G2Q is an attempt to fit constraint solving into an existing functional language. As such, G2Q’s semantics (that is, really Haskell’s semantics) are likely more comfortable for existing Haskell programmers.

Rosette [190, 191] is an extension of Racket, which allows for constraint based programming. Unlike G2Q, Rosette requires that all constraint generation be self-finitizing; that is, all constraint generation must terminating. The trade-off here is that, Rosette, unlike G2Q, offers guaranteed termination. However, this also means that Rosette rules out some valid programs. Rosette supports only symbolic integers and booleans, while G2Q supports lifting any first-order value to a symbolic value (given an instance of G2Rep).

Like Rosette, Kaplan [122] allows for constraint based programming, although in Scala rather than in Racket. Like G2Q, Kaplan supports a variety of types, including algebraic datatypes. Due to Racket and Scala both being strict languages, however, neither Rosette nor Kaplan account for non-strict execution.

Compile-time Theorem Proving  
HALO [198] and [204] aim to translate Haskell programs into first-order logic in order to apply contract verification to Haskell programs. G2Q, on the other hand, is aimed at supporting runtime constraint solving in Haskell while these tools instead focus on ensuring that Haskell programs satisfy contract specifications.
**Constraint-Based Synthesis** Complete Functional Synthesis [123] also describes a technique to write programs by writing constraints. Unlike the other discussed work, it relies on synthesis of code at compile time, rather than constraint solving at runtime. This presents a trade-off: the code it synthesizes is more efficient, but the logic it can reason about is more restricted. For example, Complete Functional Synthesis does not support recursive functions or algebraic datatypes.

**SMT APIs** A number of SMT solvers, including Z3 [64], CVC4 [39], and Yices [71], have API interface for a variety of languages. Depending on the language the API is intended for, some of these offer strong type guarantees. Relatedly, there are Haskell packages [78, 18] and packages for other languages [20, 121] that expose strongly typed bindings to SMT solvers. However, these sorts of API interfaces are all very close to the abstraction level of the SMT solver, as they directly expose SMT constructs. In addition, these strongly typed API interfaces still require a great deal of manual work related to duplicating and copying data.

### 3.7 Conclusion

We present G2Q, a quasiquoter for Haskell to ease access to constraint solving. By leveraging the G2 symbolic execution engine, G2Q allows users to easily encode constraints with minimal engineering overhead, and a higher level of abstraction than with tools like SMT solvers.
Chapter 4

Counterfactual Symbolic Execution

This chapter describes work completed in collaboration with Anton Xue, Maxwell Bland, Ranjit Jhala, and Ruzica Piskac. This work includes material originally published in [94].

4.1 Introduction

Modular verifiers allow programmers to specify correctness properties of their code using function contracts, such as pre- and post-conditions (e.g. ESC/Java [81], Dafny [128]), or refinement types (e.g. DML [202], F* [186]). Unfortunately, modular verifiers can be very difficult to use: when verification fails, the hapless programmer is given no feedback about why their code was rejected, let alone how they can fix it.

There are two ways in which modular verification can fail when checking if a function $f$ satisfies a contract given by a pre-condition $P$ and a post-condition $Q$.

First, the code may be wrong. That is, the precondition $P$ may be too weak and the postcondition may only hold on a smaller set of inputs than those described by the precondition. Alternatively, the postcondition $Q$ may be too strong i.e. the function’s code is incorrect and establishes a weaker property than stipulated by the postcondition.

Second, more perniciously, the code of $f$ may be correct, but verification may still fail as the library functions’ contracts may be wrong: the post-condition for some callee (library) function $g$ may not capture enough information about the values returned by that function in order to allow the desired property to be established at the caller (client) $f$. For example, consider the Dafny [128] code shown in Figure 5.1. The Dafny verifier rejects this code, complaining that it cannot prove the postcondition for $\text{main}$. The problem here is not the code, which is clearly correct, but that the contract for $\text{incr}$ is too weak: the post-condition that it returns a non-negative value is not enough to prove the post-condition that $\text{main}$ returns $x + 2$.

One might be tempted to use bounded model checking [41] or symbolic execution [119] to enumerate paths through the code in order to find execution traces that witnesses the failure i.e. to find set of inputs that satisfy the precondition but which
Figure 4.1: A Dafny program where `main` fails to verify due to a weak specification for `incr`.

```
method incr(x: int) returns (r: int)
requires 0 ≤ x
ensures 0 ≤ r
{ r := x + 1; }

method main(x: int) returns (r: int)
requires 0 ≤ x
ensures r = x + 2
{ var tmp := incr(x);
  r := incr(tmp); }
```

produce an output which violates the postcondition [48]. However, this approach will be fruitless in the case where the code actually satisfies the contract but verification fails due to imprecise specifications for callee functions.

We introduce the novel concept of abstract counterexamples to help programmers debug errors due to imprecise specifications. An abstract counterexample for a function `f` and its callee `g` is a partial definition of `g` that satisfies `g`'s contract, but creates a violation of `f`'s contract. For the code in Fig. 5.1 we aim to find an abstract counterexample:

```
main(0) = 0
violating the contract of 'main' if
incr(0) = 0
Strengthen contract of 'incr'
to eliminate this possibility
```

The counterexample is a partial definition of the callee `incr` where `incr(0) = 0`. This definition satisfies `incr`'s contract but causes a violation of the caller `main`'s contract. The user can use the above to strengthen `incr`'s contract to `r == x + 1` to verify `main`.

In this chapter, we develop and evaluate lazy counterfactual symbolic execution, a new technique to generate concrete and abstract counterexamples that localize the causes of failure of static modular verification for non-strict languages like Haskell. We do so via the following concrete contributions.

1. **Counterfactual Branching** Our first contribution is the notion of counterfactual branching that allows us to simultaneously conduct a symbolic search for both concrete and abstract counterexamples (Section 4.3). A counterfactual branch denotes a choice between two alternative implementations of some function, e.g. the function’s concrete implementation or an abstract one derived from the function’s specification. Our key insight is that we can find abstract counterexamples by finding a counterfactual branch from which all concrete executions are safe, but from which some abstract execution leads to an error.

2. **Refinement Types as Contracts** Our second contribution is to show how to use counterfactual symbolic execution to localize the cause of refinement type errors
We show how to translate refinement types into value-level assertions and where refinement type specifications for functions are translated into the abstract implementations to be used at counterfactual branches.

3. Implementation and Evaluation Our last contribution is an implementation of our approach as a tool, G2. We evaluate G2 on a corpus of 7550 refinement type errors from users of LiquidHaskell, a verification tool that has been used to verify various properties of the Haskell standard libraries [196] (Section 4.5). G2 is able to quickly find counterexamples 97.7% of the time. 57.6% of the time, G2 finds concrete counterexamples showing how the code fails the specification, and 40.1% of the time it finds abstract counterexamples caused by an imprecise specification. By comparing the “error”-ing programs with their “fixed” versions we find that the abstract counterexamples correctly pinpoint the library function whose specification was too weak in 96.1% of the cases, demonstrating the importance, effectiveness and practicality of counterfactual symbolic execution in making modular verification more usable.

4.2 Overview

4.2.1 Refinement Type Counterexamples

A refinement type constrains classical types with predicates in decidable first-order logics. For example, we can specify that the function die should never be called at run-time by assigning it the type:

\[
\text{die} :: \{ x : \text{String} \mid \text{false} \} \to a
\]
\[
\text{die} \ x = \text{error} \ x
\]

The refinement type checker will verify that at each call-site, the function die is called with values satisfying the condition false. As no such value exists, the code will only typecheck if all calls to die are, in fact, provably unreachable.

A restricted class of functions may be lifted into refinement types to specify properties of algebraic data types. For example, the following function computes the size of a list:

\[
\text{size} :: [a] \to \text{Int}
\]
\[
\text{size} \ [] = 0
\]
\[
\text{size} \ (x:xs) = 1 + \text{size} \ xs
\]

Using size, one can write a safe head function as:

\[
\text{head} :: \{xs:[a] \mid \text{size} \ xs > 0\} \to a
\]
\[
\text{head} \ (x:xs) = x
\]
\[
\text{head} \ [] = \text{die} \ "\text{Bad call to head}"
\]

The input refinement type of head states that it is only called with positively-sized lists. As in the second equation the size is equal to 0, the second pattern is inconsistent with the input refinement, and hence, provably never reachable.
Concrete Counterexamples

It is often not obvious why a refinement type fails. Consider `zip`, defined below:

```haskell
zip :: xs: [a] -> {ys: [b] | size xs > 0 => size ys > 0} -> [(a, b)]
zip [] [] = []
zip ( x:xs ) ( y:ys ) = (x, y) : zip xs ys
zip _ _ = die "Bad call to zip"
```

The function iterates over two lists and produces a new list of corresponding pairs. It is rejected by the refinement type checker LiquidHaskell [197] with the vexing error:

```
zip (x:xs) (y:ys) = (x, y): zip xs ys
```

```
^^^^^^^^^
```

Inferred type

```haskell
VV : {v : [a] | size v >= 0 && len v >= 0 && v == ys}
```

not a subtype of Required type

```haskell
VV : {VV : [a] | size xs > 0 => size VV > 0}
```

This error can be more confusing than helpful. Instead, a counterexample that illustrates an instance where program execution violates the refinement types may provide better insight. Running our tool yields the following:

```
zip [] [0] = error
```

makes a call to

```
die "Bad call to zip" = error
```

violating die’s refinement type

The counterexample ([] [0]) illustrates an input that satisfies `zip`’s precondition, but causes `zip` to invoke the `die` function. With this information in hand, the user can see how to improve the refinement type (namely it is not enough that the second list be non-empty when the first is - we require that the lists have the same size.)

4.2.2 Localizing Imprecise Refinement Types

Next, consider `concat`, which concatenates a list of lists into a single list, with the goal of verifying that the size of the returned list is the sum of the sizes of the lists in the input:

```haskell
sumsize :: [[a]] -> Int
sumsize [] = 0
sumsize ( x:xs ) = size x + sumsize xs
```

```haskell
concat :: x: [[a]] -> {v:[a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((append xs ys):xss)
```
append :: [a] -> [a] -> [a]
append xs [] = xs
append [] ys = ys
append (x:xs) ys = x:append xs ys

This `concat` implementation is correct, but is rejected by LiquidHaskell. To make verification modular, and hence, scalable, at each function call, LiquidHaskell is only aware of the refinement type of the callee, and not the actual definition. Thus, when trying to verify `concat`, LiquidHaskell knows nothing about the value returned by `append`.

Thus, the above example illustrates a common, and confusing, situation where the verifier rejects a program, not because the property being checked does not hold (as in `zip`), but because the specifications for called functions are too weak. Worse, as the code is correct, we cannot report counterexamples, since they do not exist.

**Abstract Counterexamples** In this situation, ideally we would point the user to the function whose type needs to be tightened. We do so by introducing the notion of an abstract counterexample, where we show how the overall property can be violated by using an abstract implementation of the callee that is derived solely from the (refinement type) specification for the callee.

For example, an abstract counterexample for `concat` is:

```
concat [[0], []] = [0, 0]
```

violating its refinement type, if

```
append [0] [] = [0, 0]
```

Strengthen the refinement type of `append` to eliminate this possibility

The abstract counterexample tells the user that the existing specification for `append` permits the call `append [0] []` to return `[0, 0]`, which causes the evaluation of `concat [[0], []]` to return a value that violates its specification.

Crucially, the abstract counterexample points the user to the fact that the error only arises due to the (trivial) refinement type specification for `append` and not due to the actual implementation of the function. Inspired by this message, a user could improve the type refinement on `append` to:

```
append :: x:[a] -> y:[a]
- > {z:[a] | size x + size y = size z}
```

which then lets LiquidHaskell verify `concat`.

**Counterfactual Symbolic Execution** We can find both concrete and abstract counterexamples with a new technique called counterfactual symbolic execution. We introduce a counterfactual branching operator, essentially a non-deterministic choice operator that can evaluate either of its two arguments. Each function definition is replaced with a counterfactual branch that non-deterministically chooses either the concrete implementation, or an abstract version derived solely from the function’s refinement type.

We can then run symbolic execution as before, and report an abstract coun-
terexample at those counterfactual branches where the concrete choice produces no counterexamples, but the abstract one does. In this case, as illustrated above, we can also report exactly how the abstract implementation leads to a property violation.

### 4.3 Counterfactual Symbolic Execution

Modular verifiers allow users to write and automatically check specifications describing preconditions or postconditions for functions. Unfortunately, verification errors can be difficult for users, as error messages typically involve logical formulas, which may not be obviously linked to the written specifications.

As discussed in Section 4.2.1 and Section 4.2.2 we use symbolic execution to find two types of counterexamples. Ideally, we find concrete counterexamples, i.e. actual function inputs that lead to a specification violation. However, we also introduce abstract counterexamples, found via counterfactual symbolic execution, to help debug spurious errors. As shown in Section 4.2.2, counterfactual symbolic execution finds partial function definitions for directly called functions that obey their function specifications, but demonstrate why the caller’s specification is not verified.

Our goal, then, is to find a minimally abstract or least abstracted counterexample—either a concrete counterexample, or a counterexample with a minimal number of abstracted functions. Such states are likely to be the most understandable to a user, as they most closely resemble an actual execution of the program, and (in the case of an abstract counterexample) most precisely identify which functions might need stronger specifications.

**Specifications** To this end, we introduce three functions that we require on the original specifications: `pre` returns just the preconditions, `post` returns just the postconditions, and `toExp` converts a specification to a $\lambda_G$ expression. Here, we assume these functions can be implemented for some arbitrary set of specifications. In Section 4.4, we show these functions over LiquidHaskell refinement types.

**Counterfactual Function Definitions** To find abstract counterexamples, we create assertion functions and counterfactual functions. Given a function $f \equiv \lambda \overline{x}. e$ with a specification $c$, we define its assertion function as:

$$f^a \equiv \lambda \overline{x}. \text{let } r = f \overline{x} \text{ in assert (toExp}(c) \overline{x} r) \text{ in } r$$

We define the counterfactual function of $f$ as:

$$\hat{f} \equiv \lambda \overline{x}. f^a \overline{x} (\text{let } s = ? : \tau \text{ in assume (toExp}(post(c)) \overline{x} s) \text{ in } s)$$

When symbolic execution reduces $\hat{f}$, it binds the arguments to lambdas as usual. Then, due to the counterfactual branch, it splits into two symbolic states. We will refer to these as the left and right states, corresponding to the left and right of the counterfactual choice. The left state corresponds to normal execution, with an assertion that both ensures that the function’s preconditions are met, and that the
function returns values that satisfy its postcondition. In the right state, we introduce a new symbolic variable, $s$, that is assumed to satisfy the function’s postcondition (as defined by the specification $c$), but which otherwise makes no use of $f$’s definition. Therefore, $s$ can take on any value that $f$ would be allowed to return by its postcondition. This allows us to find abstract counterexamples when $f$’s implementation is correct, but its specification does not describe its behavior precisely enough to verify a caller. The right state does not check that its arguments satisfy its preconditions, because if there is a violation of a precondition, it will also occur in the left case.

We can find (abstract) counterexamples for a function $f$ of arity $n$, with specification $c$. To do so, we define another special copy of $f$, called $f_{det}$. The function $f_{det}$ is $f$, but with each occurrence of a callee function $g$ replaced by $\hat{g}$. This matches how modular verifiers use the implementation of their client functions, by using the definition of $f$, but only the specifications for library functions when verifying that $f$ meets its specification.

Then we perform symbolic execution starting from an initial state defined as follows:

$$\text{assume} \left(tocExp\left(\text{pre}(c) \ s\right)\right) \ \text{in} \ \left((f_{det})^a \ s\right)$$

$s$ are symbolic inputs that ensure that any counterexample we find use inputs satisfying $f$’s precondition.

In order to find minimally abstract counterexamples, we maintain a counter of the number of right paths selected for each states. Then, we filter the found states, and present only those which require the fewest abstracted functions.

### 4.3.1 Search Strategy

Symbolic execution, as used for counterfactual symbolic execution, is an unbounded and therefore incomplete search technique. When searching for counterexamples, we aim to minimize the number of abstracted functions, but we can almost never actually prove we succeeded (and an incompletely minimized counterexample may still be useful to a user.) Here, we describe two strategies we employ to try to minimize time spent searching, while still finding useful, and close to minimal, counterexamples.

**Abstract Counterexample Filtering** Presenting only minimally abstract counterexamples allows us to prune during symbolic execution. If we find an assertion violation with $n$ abstracted functions, we can drop any state— including states which have not finished execution— in which we abstracted $n + 1$ or more functions.

**Search Deepening** The reductions rules in Section 2.3.3 implicitly create a (often infinite) tree of states. The order we search the branches of this tree, and how deep we search, is an important consideration to find counterexamples efficiently.

We search in a depth first manner, to some maximal depth. If we have explored all branches, and not found a counterexample, we increase the maximal depth and continue searching. Every time we find a counterexample that is better (has less
abstracted functions) than our current best counterexample, but that is deeper in the tree, we also increase the maximal depth.

Thus, this strategy allows searching to continue if better counterexamples are being found by searching deeper in the tree. However, we avoid fruitlessly searching too many states, if they are not producing promising results.

The gradual increase in the maximal depth ensures we are evaluating a variety of states and branches, preventing us from spending too much time on branches that will not yield a counterexample. It often enables us to find a close-to-minimal counterexample fairly quickly, allowing us to prune all states with a greater number of abstracted functions.

### 4.4 Refinement Type Counterexamples

Now that we have described a general technique for counterfactual symbolic execution, we turn our attention to leveraging it to generate counterexamples to refinement types, as shown in Section 4.2.1 and Section 4.2.2.

**Refinement Types** We support the language of refinement types shown in Figure 4.2. This subset includes operations on numeric types, measures (e.g. size from Section 4.2.2), and refinements on polymorphic arguments. In the refinement language, \( \{ v : b [\tau_1 \ldots \tau_k] | r \} \) represents the base type \( b \) refined by the predicate \( r \). The \( [\tau_1 \ldots \tau_k] \) are type arguments to the base type, which may themselves be further refined. The \( v \) is an inner bound name, allowing reference to the value of the type in \( r \) and \( \tau_1 \ldots \tau_k \). The \( x : \tau_1 \rightarrow \tau_2 \) is a function of type \( \tau_1 \) to \( \tau_2 \). The \( x \) is a outer bound name to refer to the value of \( \tau_1 \), allowing it to be referenced in refinements in \( \tau_2 \).

To use counterfactual symbolic execution for refinement types, we need only convert refinement type specifications to assume and assert expressions. That is, we need only implement the three functions, \( \text{pre} \), \( \text{post} \), and \( \text{toExp} \), described in Section 4.3, that describe the contracts of each function.

**Pre and Post** Figure 4.3 shows \( \text{pre} \) and \( \text{post} \). \( \text{pre} \) walks over the function and drops the return type. \( \text{post} \) keeps the argument bindings, but sets all refinements, except the return type’s refinement, to True. Keeping the bindings is important, as they may be used in the return type’s refinement.

**Converting Refinements to Contracts** Refinement types are converted to contracts, i.e. asserts and assumes, on the inputs and output of a function. \( \text{toExp} \), shown in Figure 4.4, translates LiquidHaskell refinement types into predicates in \( \lambda_G \). This function has many subparts:

- \( \text{toExp}_\lambda \) creates lambda bindings, giving us names to refer to both the inputs and outputs of the function.

- \( \text{toExp}_b \) and \( \text{toExp}_r \) translate each individual refinement on a type into a \( \lambda_G \) predicate on a value.
\[
\tau ::= \begin{align*}
\{ v : b[\tau] \mid r \} & \quad \text{refinement} \\
x : \tau \rightarrow \tau & \quad \text{function}
\end{align*}
\]

\[
b ::= \begin{align*}
\text{Int} & \quad \text{integer} \\
\text{Bool} & \quad \text{boolean} \\
A & \quad \text{algebraic data type}
\end{align*}
\]

\[
r ::= \begin{align*}
r == r & \quad \text{equality} \\
r < r & \quad \text{inequality} \\
r \land r & \quad \text{conjunction} \\
\neg r & \quad \text{negation} \\
x & \quad \text{variable} \\
m r & \quad \text{measure application} \\
n & \quad \text{integer value} \\
r \oplus r & \quad \text{integer operation} \\
\text{true} & \quad \text{true} \\
\text{false} & \quad \text{false}
\end{align*}
\]

\[
m ::= m
\]

Figure 4.2: \(\lambda_D\) types

- \text{toExp}_\tau\) walks over the spine of a LiquidHaskell function type, to apply \text{toExp}_b\) to each argument.

\textit{Polymorphic Data Types} LiquidHaskell allows checking refinements on polymorphic type variables. For example, we may refine a polymorphic list \([a]\) to contain only positive integers, by writing \([\{ x : \text{Int} \mid 0 < x \}]\). Thus, we require a way to translate LiquidHaskell polymorphic type refinements, into predicates on expressions in \(\lambda_G\). To do this for a type constructor \(\tau\), with type variable \(a\), a higher order function \(p_r\) is automatically created. The function takes an expression of type \(\tau\ a\), and a predicate of function type \(a \rightarrow \text{Bool}\). It walks over the structure of the type, conjoining the application of the predicate to each occurrence of \(a\). We can then apply \(p_r\) to a predicate expression and an expression of type \(\tau\), to assume or assert that those predicate expressions hold on all type variables in \(p\). For example, on a list, we have:

\[
p_{\text{List}} \quad p \quad [] \quad = \quad \text{True}
\]
\[
p_{\text{List}} \quad p \quad (x : xs) \quad = \quad (p\ x) \land (p_{\text{List}}\ p\ x\ s)
\]

and we translate \([\{ x : \text{Int} \mid 0 < x \}]\) to \(p_{\text{List}} (\lambda x . 0 < x)\).
\[
\begin{align*}
pre(\tau) &= \begin{cases}
x_1: \tau_1 \rightarrow \tau = x_1: \tau_1 \rightarrow (x_2: \tau_2 \rightarrow \tau_3) \\
pre(x_2: \tau_2 \rightarrow \tau_3) \\
\tau_1 & \quad \tau = x: \tau_1 \rightarrow \tau_2
\end{cases}
\]

\[
post(\tau) = \begin{cases}
x_1: \text{bt}(\tau_1) \rightarrow \text{post}(\tau_2) & \quad \tau = x: \tau_1 \rightarrow \tau_2 \\
\{v: b[\tau_1 \ldots \tau_k] | r\} & \quad \tau = \{v: b[\tau_1 \ldots \tau_k] | r\}
\end{cases}
\]

\[
\text{bt}(\tau) = \begin{cases}
\{v: b[\text{bt}(\tau_1) \ldots \text{bt}(\tau_k)] | true\} \\
\tau & \quad \text{otherwise}
\end{cases}
\]

**Figure 4.3:** $\lambda_D$ precondition and postcondition

\[
\text{toExp}(\tau) = \text{toExp}_\lambda(\tau, \tau)
\]

\[
\text{toExp}_\lambda(\tau, \tau_a) = \begin{cases}
\lambda x . \text{toExp}_\lambda(\tau_2, \tau_a) & \quad \tau = x: \tau_1 \rightarrow \tau_2 \\
\lambda x^F . \text{toExp}_r(x^F, \tau_a) & \quad \text{for fresh } x^F \\
\lambda x . \text{toExp}_\lambda(\tau_2, \tau_a) & \quad \tau = \{v_1: b[\ldots] | r\}
\end{cases}
\]

\[
\text{toExp}_r(x^F, \tau) = \begin{cases}
\text{toExp}_b(x, \tau_1) \land \text{toExp}_r(x^F, \tau_2) & \quad \tau = x: \tau_1 \rightarrow \tau_2 \\
\text{toExp}_b(x^F, \tau) & \quad \tau = \{v: b[\tau_1 \ldots \tau_k] | r\}
\end{cases}
\]

\[
\text{toExp}_b(x, \tau) = \begin{cases}
(\lambda v . p_b(v, \text{toExp}_r(x_1, \tau_1), \ldots, \text{toExp}_r(x_k, \tau_k))) \\
\land \text{toExp}_r(r) & \quad \text{for fresh } x_1 \ldots x_k \\
\text{True} & \quad \tau = x: \tau_1 \rightarrow \tau_2
\end{cases}
\]

\[
\text{toExp}_r(r) = \begin{cases}
\text{toExp}_r(r_1) \equiv \text{toExp}_r(r_2) & \quad r = r_1 \equiv r_2 \\
\text{toExp}_r(r_1) < \text{toExp}_r(r_2) & \quad r = r_1 < r_2 \\
\text{toExp}_r(r_1) \land \text{toExp}_r(r_2) & \quad r = r_1 \land r_2 \\
\ldots & \quad \ldots
\end{cases}
\]

**Figure 4.4:** $\lambda_D$ to $\lambda_G$ translation
4.5 Implementation and Evaluation

We next present an evaluation that demonstrates the effectiveness of our method for localizing refinement type errors.

4.5.1 Quantitative Evaluation

The goal of our evaluation is twofold. Q1 Does symbolic execution find counterexamples that explain refinement type errors? Q2 Do the abstract counterexamples accurately pinpoint the functions whose specifications are too weak to permit type checking?

Our empirical evaluation answers these questions positively. We use G2 to generate counterexamples for refinement type errors on a corpus of programs written by students using LiquidHaskell for a homework assignment in CSE 230, a graduate level programming languages class, at the University of California, San Diego (IRB #140608). The assignment contained a variety of exercises. For some, the students had to write code that implemented a function, and matched a given refinement type. For others, the students were asked to write refinement types for prewritten functions. In total, each students assignment was roughly 150 to 200 lines of code.

Corpus The corpus contains, in total, 10,349 incorrect refinement types. The data was collected by logging the student’s work every time a student typechecked their code with LiquidHaskell. Consequently, the data set comprises traces of files, giving us access to the code at different stages of progression — both the incorrect programs and the correct one that finally type checked.

Preprocessing The corpus was collected from a class run in 2015. LiquidHaskell’s syntax has changed since then, rendering some of the files non-parsable. Altogether, on the student written data set G2 can be applied to 93.6% of the files. From those, we excluded 2136 functions because they were only stubs, which immediately called error. Finding counterexamples for these functions is trivial, because any input would be a counterexample. This left us with a total of 7550 functions to evaluate G2 on.

Search Strategy Our search deepening strategy (Section 4.3.1) takes two parameters: an amount $s$ to increase the search depth, if no counterexample is found, and an amount $c$ to increase the search depth, when a better counterexample is found. Based on our experience with G2 we selected $s = 300$ and $c = 500$ as values that appeared to give reasonable results. G2 was given a maximum of 2 minutes to find counterexamples for each function.

Results Figure 5.5 summarizes the results of our evaluation on the 7550 functions, drawn from actual code written by students. It demonstrates that G2 finds counterexamples for the vast majority of the LiquidHaskell errors. In total, we found counterexamples for 7379, or 97.7%, of the errors. We found concrete counterexamples for 4354, or 57.6%, of the errors, and found abstract counterexamples for 3025, or 40.1%, of the errors. While G2 has an average runtime of only 17.6 seconds, the
median running time is even lower – 7.9 seconds. This shows that G2 is a practical and efficient tool to help debug LiquidHaskell refinement type errors, giving a very positive answer to \textit{Q1}.

G2 failed to find a counterexample only 2.3% of the time. 1.5% of our failures come from timeouts, while the remaining 0.7% is accounted for by errors in G2, which mostly relate to unimplemented edge cases in LiquidHaskell specifications.

\textbf{Correctness of Abstract Counterexamples} Our benchmarks come from traces of programmers iteratively invoking LiquidHaskell to verify some properties. Thus, we determine whether G2’s abstract counterexamples correctly localize the imprecise specification by comparing each “bad” file – that was rejected by LiquidHaskell, for which G2 found an abstract counterexample – with the first “fixed” file along the user’s trace that was accepted by LiquidHaskell. We say that an abstract counterexample correctly localizes the error if the counterexample blames a call to some function \( f \) such that in the “fixed” version (a) the user specifies a different type for \( f \), or (b) the user replaces \( f \) with a different function with a stronger type, or (c) LiquidHaskell infers a different type for \( f \) e.g. because it is used differently in the code. We say an abstract counterexample is spurious otherwise.

\textbf{Evaluating Correctness} Of the 3025 counterexamples, after discarding 1041 “bad” files that had no “fixed” version (as some students did not finish the assignments) we were left with 1984 abstract counterexamples. We categorized these counterexamples via a combination of scripts and manual inspection as one of (a), (b), (c) or spurious. We find that in 1747 (88.1%) cases the user ends up specifying a different type (a), in 9 (0.4%) cases the user ends up replacing the function (b), and in 151 (7.6%) cases the user ends up changing other code to allow LiquidHaskell to infer the right type needed for verification (c). Thus, we conclude that in 96.1% of the cases, G2’s abstract counterexamples correctly identified the function whose specification was too weak.

\textbf{Replicate Function} One particularly interesting abstract counterexample stood out to us. This counterexample was actually counted as spurious, as it does not fit any of our classifiers for correct abstract counterexamples, but nonetheless shows something interesting about the code. Consider:

\begin{verbatim}
replicate :: n:Int -> a -> { xs: [a] | size xs == n }
replicate 0 x = []
replicate n x = x:replicate n x
\end{verbatim}

\texttt{replicate} is supposed to return a list of the given length, but due to a mistake in the implementation (the counter is never decreased) instead returns an infinite list. However, classical symbolic execution would fail to find a concrete counterexample, because the computation of \texttt{size xs == n} would never terminate. However, G2 finds an abstract counterexample for \texttt{replicate}:

\begin{verbatim}
replicate 1 0 = [0, 0]
\end{verbatim}

violating its refinement type, if

\begin{verbatim}
replicate 1 0 = [0]
\end{verbatim}
If the first recursive call to \texttt{replicate 1 0} returns \([0]\), the outer call to \texttt{replicate 1 0} returns \([0, 0]\), violating the refinement type.

Our primary motivation to develop abstract counterexamples was to aid in cases where the specification was insufficient. Therefore, it was a surprising discovery that it can also provide output in cases of non-termination.
<table>
<thead>
<tr>
<th>Function</th>
<th>Con.</th>
<th>Abs.</th>
<th>Time out</th>
<th>Error</th>
<th>Avg. Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>prop_map</td>
<td>13</td>
<td>523</td>
<td>2</td>
<td>2</td>
<td>8.9</td>
</tr>
<tr>
<td>foldr1</td>
<td>607</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>10.1</td>
</tr>
<tr>
<td>kmeans1</td>
<td>0</td>
<td>875</td>
<td>22</td>
<td>0</td>
<td>120.5</td>
</tr>
<tr>
<td>prop_concat</td>
<td>877</td>
<td>217</td>
<td>0</td>
<td>5</td>
<td>5.7</td>
</tr>
<tr>
<td>replicate</td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>17.9</td>
</tr>
<tr>
<td>mergeCluster</td>
<td>329</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>7.2</td>
</tr>
<tr>
<td>collapse</td>
<td>250</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>8.3</td>
</tr>
<tr>
<td>prop_zipWith</td>
<td>18</td>
<td>708</td>
<td>13</td>
<td>2</td>
<td>10.5</td>
</tr>
<tr>
<td>concat</td>
<td>181</td>
<td>80</td>
<td>53</td>
<td>8</td>
<td>58.3</td>
</tr>
<tr>
<td>nearest</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>10.5</td>
</tr>
<tr>
<td>prop_replicate</td>
<td>335</td>
<td>125</td>
<td>0</td>
<td>2</td>
<td>5.7</td>
</tr>
<tr>
<td>expand</td>
<td>49</td>
<td>74</td>
<td>0</td>
<td>0</td>
<td>8.3</td>
</tr>
<tr>
<td>length</td>
<td>16</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>40.8</td>
</tr>
<tr>
<td>zipWith</td>
<td>760</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>9.2</td>
</tr>
<tr>
<td>kmeans</td>
<td>0</td>
<td>18</td>
<td>2</td>
<td>0</td>
<td>120.5</td>
</tr>
<tr>
<td>centroid</td>
<td>594</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6.4</td>
</tr>
<tr>
<td>prop_size</td>
<td>0</td>
<td>224</td>
<td>2</td>
<td>3</td>
<td>6.7</td>
</tr>
<tr>
<td>mapReduce</td>
<td>93</td>
<td>137</td>
<td>8</td>
<td>8</td>
<td>12.2</td>
</tr>
<tr>
<td>add</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4.4</td>
</tr>
<tr>
<td>concat2</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>10.0</td>
</tr>
<tr>
<td>prop_concat2</td>
<td>36</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5.9</td>
</tr>
<tr>
<td>distance</td>
<td>84</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>9.5</td>
</tr>
<tr>
<td>prop_concat_1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6.0</td>
</tr>
<tr>
<td>prop_join</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>5.6</td>
</tr>
<tr>
<td>Other</td>
<td>16</td>
<td>14</td>
<td>0</td>
<td>7</td>
<td>14.5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>4354</strong></td>
<td><strong>3025</strong></td>
<td><strong>115</strong></td>
<td><strong>56</strong></td>
<td><strong>17.6</strong></td>
</tr>
</tbody>
</table>

**Figure 4.5:** Evaluation results for errors reported by LiquidHaskell on student homeworks. Con. is the number of reported concrete counterexamples. Abs. is the number of abstract counterexamples reported by G2. Timeout is the number of times G2 timed out before returning counterexamples. Error is the errors encountered in G2 when generating counterexamples. Avg. Time is the average amount of time taken by all runs of G2 reported in the table.
4.6 Related Work

Verification Techniques and Debugging An IDE for Dafny that helps debug genuine and spurious failed verification conditions is described in [55]. Like our work, it uses a symbolic execution based approach to find concrete counterexamples. However, for spurious errors, it simply displays the SMT model, which, unlike abstract counterexamples, does not pinpoint any specific function whose specification needs strengthening for verification to succeed.

CORRAL [126, 124, 125] is a reachability solver based on generating verification conditions. CORRAL introduces the stratified inlining technique, which inlines functions on demand if verification fails when just using the function contracts. As opposed to counterfactual counterexamples, stratified inlining aims to improve the underlying verification, rather than improve explainability of verification errors. As such, stratified inlining can be seen as an orthogonal technique to G2’s counterfactual counterexamples. Stratified inlining aims to minimize the number of inlined functions, whereas counterfactual counterexample generation aims to minimize the amount of abstraction.

Haskell Verification Xu’s work on static contract checking [203, 204], relies on a symbolic simplifier, parts of which resemble our reduction rules (Section 2.3.3.) Similarly, Halo [198] and LiquidHaskell [197] aim to verify properties of Haskell programs. However, in contrast to G2, these tools aim for verification, as opposed to refutation which is the goal of our lazy reduction-based symbolic execution. None of them produce abstract counterexamples when verification fails.

4.7 Conclusion

We presented counterfactual symbolic execution for non-strict languages, and used it to find counterexamples that illustrate concretely or abstractly why a modular checker fails to verify a program. Our evaluation on a large corpus of 7550 verification errors from users of LiquidHaskell demonstrates that we can find counterexamples to 97.7% of errors. For 57.6% of the errors we find concrete counterexamples, and for an additional 40.1% of the errors we find abstract counterexamples, which 96.1% of the time correctly pinpoint the imprecision that precludes verification. Thus, our results show that by generalizing the notion of counterexamples via counterfactual execution, we can quickly, automatically, and accurately guide the puzzled developer to the part of their code or specification that they need to fix.
Chapter 5

Counterexample-Guided Inference of Modular Specifications

This chapter describes work completed in collaboration with Ranjit Jhala and Ruzica Piskac.

5.1 Introduction

Modular verifiers like Dafny [128], ESCJava [81], and LiquidHaskell [196]) let programmers specify and verify properties of their code by writing contracts: pre- and postconditions specifying each function’s behavior. Unfortunately, modular verifiers are often difficult to use when a function invokes one or more different functions. The problem is caused by modularity: when verifying a function \( f \), modular verifiers use only the specifications of called functions and not their implementations. Thus, a modular verifier may fail to verify \( f \) not because its specification is incorrect, but because the specification of some of its callees is too weak, i.e. does not capture enough of the callee’s behavior to verify \( f \).

For example, Dafny fails to verify the program in Figure 5.1 reporting, instead, that it cannot prove the postcondition of `main`. While the postcondition of `main` always holds at runtime, modular verification fails as the callee `double` has no specification, and Dafny has no information about the value returned by calling `double` within `main`. Some verifiers try to overcome this issue by inlining the callee [98], but this approach is fruitless for recursive code.

We introduce a new algorithm that automatically synthesizes the specifications needed for modular verification. Our algorithm works by traversing the call-graph of the program in a top-down fashion as summarized in Figure 5.2. In addition to our theoretical algorithm, we develop a practical tool, Lynx, capable of finding specifications for the LiquidHaskell modular verifier.

1. Analyzing Single Calls Our first contribution is a method to synthesize a specification for a single call-edge. Suppose the program has only two functions \( f \)
method double(x: int) returns (r: int)
{ r := x + x; }

method main(x: int) returns (r: int)
ensures r = 2*x
{ r := double(x); }

Figure 5.1: A Dafny program, where verification of main fails due to double’s weak post-condition.

Figure 5.2: An overview of Lynx’s inference algorithm.

and g, where f calls g. Our algorithm uses the two phases of a CEGIS loop [185] — verification and synthesis — to find a specification for g that verifies f. The algorithm starts in the verification phase, where it tries to verify the implementation of f against its specification, modularly, using just the specification of g. If the verifier succeeds, the process stops. However, if the verifier fails, we apply counterfactual symbolic execution (as described in Chapter 4) to find a counterexample that explains the failure. We use this counterexample to produce a constraint that the specification for g must satisfy to successfully verify f. We add this constraint to the set of constraints generated in the previous runs of the CEGIS loop. Next, the algorithm moves into the synthesis phase where it invokes a template-based synthesizer [172, 91] which uses the constraints to generate a candidate specification for g. We take this new candidate specification and repeat the CEGIS loop, until we can verify f.

2. Analyzing Nested Calls Our second contribution is an interpolant based technique to generalize the above to nested calls. Returning to our example, after we derive a specification for g that verifies f, g itself must be verified against the candidate specification. To do so, g now plays the same role as f before, while g’s callees play the role previously played by g. The process continues until we reach the leaves of the call graph. If at some point, some previously guessed specification is found to be incorrect via a concrete counterexample, the algorithm backtracks and synthesizes a different specification for the previous function in the call graph. When reverting, we also pass back an interpolant which blocks the incorrect candidate specification, thus ensuring we do not repeatedly traverse the same ineffectual parts of the search space.
3. Completeness We say an inference algorithm is complete if it (a) always returns a set of specifications sufficient for verification to succeed, when such a set exists, and (b) always find a counterexample showing that an input specification does not hold, if such a counterexample exists. Our algorithm is parameterized by three components: a verifier, counterexample generator and specification synthesizer. We prove that if these components are sound and complete, then our inference algorithm is also sound and complete over a finite set of possible specifications.

4. Bounded LIA Specification Synthesizer Of course, real-world programs do not have a finite set of possible specifications. We extend our approach to handle infinite sets of possible specifications, via our fourth contribution: a notion of size-bounded synthesizers which support sound and complete synthesis of linear integer arithmetic (LIA) specifications. We describe a translation of the synthesis constraints that arise from our inference algorithm into a formula in decidable linear integer arithmetic logic. In the case that the synthesis problem is unrealizable, we further describe how to use the unsatisfiable core of the formula to construct an interpolant.

5. Evaluation Finally, we implemented a tool called Lynx, which automatically finds specifications for the LiquidHaskell modular verifier. We evaluate Lynx on two sets of benchmarks. The first is a variety of functions and function specifications (totaling 529 lines of code) over list-like data structures. We show that Lynx is automatically finding specifications sufficient to prove correctness in several challenging benchmarks. The second set of benchmarks is a collection of 46 programs over integers, originally collected in [70]. This set of benchmarks allows us to compare to existing work. We find Lynx can verify 30 of the benchmarks, including some benchmarks which none of the existing tools can verify.

5.2 Overview

We start with an example that illustrates the problem of inferring modular specifications. At the same time, this example also outlines the basic ideas behind Lynx’s algorithm.

Example 1. Consider the following function which concatenates a list of lists into a single list. The function is defined recursively: it first takes the first two elements (lists) and merges them into one list by appending the second list after the first list. The result of this operation is again a list of the lists that has one element less. The `concat` function proceeds recursively to repeat the process on the list until only zero or one list remains:

```haskell
concat :: [[a]] -> [a]
concat [] = []
concat [xs] = xs
concat (x1s:x2s:xss) = concat ((append x1s x2s):xss)
```

```haskell
append :: [a] -> [a] -> [a]
```
append [] ys = ys
append (x:xs) ys = x:append xs ys

Note that initially both functions do not have any specification.

**Specification** To specify the intended program behavior, the code also contains two functions, `size` and `sumSize`, that respectively specify the number of elements in a list and in a list of lists:

```haskell
size :: [a] -> Int
size [] = 0
size (h:t) = 1 + size t

sumSize :: [[a]] -> Int
sumSize [] = 0
sumSize (h:t) = size h + sumSize t
```

To illustrate Lynx, we will try to verify the simple property that after applying `concat` to a list contains two lists `xs` and `ys`, the size of the resulting list is the sum of the sizes of `xs` and `ys`:

```haskell
prop_concat :: [a] -> [a] -> {b:Bool | b}
prop_concat xs ys =
    size (concat [xs, ys]) == size xs + size ys
```

The function `prop_concat` has a postcondition stating that the returned output must be `True`.

**Verification fails** Unfortunately, modular verifiers like LiquidHaskell [197] or Dafny [128] cannot prove this assertion as is. This is because such verifiers abstract the semantics of each function call with their pre and postconditions. In this case, `concat` is not annotated with any pre- or postconditions, which means the verifiers would use the default postcondition, `true`, which is too weak to prove that `size (concat [xs, ys])` is in fact equal to `size xs + size ys`.

### 5.2.1 Concrete and Abstract Counterexamples

Our algorithm for inferring suitable modular specifications uses a counterexample guided inductive synthesis (CEGIS) loop [185]. A CEGIS loop generates code that fits a set of requirements by switching between two phases: *synthesis* and *verification*. In the synthesis stage of the CEGIS loop, Lynx uses concrete and abstract counterexamples as an input to the synthesizer. In the verification stage we invoke LiquidHaskell as an off-the-shelf verifier. Here, we briefly recall the descriptions of concrete and abstract counterexamples from Chapter 4.

**Concrete Counterexamples** A *concrete* counterexample for a function consists of an input and output pair that violates that function’s specification. For instance, if `concat` from Example 1 is assigned the specification stating the output list has a strictly larger size than the input list

```haskell
concat :: xs:[[a]] -> { ys:[a] | sumSize xs < size ys}
```
Then a concrete counterexample would be the input-output pair:

\[ \text{concat} \begin{bmatrix} [1, 1], [] \end{bmatrix} = [1, 1] \]

since \( \text{sumSize} \begin{bmatrix} [1, 1], [] \end{bmatrix} == 2 \) which is not less than \( \text{size} \begin{bmatrix} 1, 1 \end{bmatrix} == 2 \).

**Abstract Counterexamples** An abstract counterexample explains why a modular verifier failed to verify a specification of a caller function, even if the specification is actually true. In essence, the abstract counterexamples describe input-output pairs that yield a violation of the specification of the caller function that (1) are not produced by the actual implementation, but (2) are allowed by the specification, thereby indicating that the specification of a callee function is inadequate for verification. As an illustration, suppose \( \text{concat} \) is annotated with a different specification:

\[ \text{concat} :: \text{xs} : \begin{bmatrix} [a] \end{bmatrix} \rightarrow \{ \text{ys} : [a] \mid \text{sumSize} \text{ xs} \leq \text{size} \text{ ys} \} \]

This specification is correct — the output \( \text{ys} \) returned by \( \text{concat} \) always has at least as many elements as the input \( \text{xs} \) — but is too weak to verify \( \text{prop_concat} \). The following abstract counterexample illustrates why \( \text{prop_concat} \) cannot be verified using the correct-but-weak specification:

\[ \text{prop_concat} \begin{bmatrix} [] \end{bmatrix} \begin{bmatrix} [] \end{bmatrix} = \text{False} \text{ if } \text{concat} \begin{bmatrix} [], [1] \end{bmatrix} = [1, 1] \]

The input-output pair \( \begin{bmatrix} [], [1] \end{bmatrix} \) satisfies the given specification for \( \text{concat} \), because \( \text{sumSize} \begin{bmatrix} [], [1] \end{bmatrix} = 1 \leq 2 = \text{size} \begin{bmatrix} 1, 1 \end{bmatrix} \), but if \( \text{concat} \begin{bmatrix} [], [1] \end{bmatrix} \) actually returned \( [1, 1] \) then \( \text{prop_concat} \begin{bmatrix} [] \end{bmatrix} \begin{bmatrix} [1] \end{bmatrix} \) would evaluate to \text{False}, violating the assertion that we aim to verify. Thus, these two pairs form an abstract counterexample demonstrating that \( \text{concat} \)'s specification is inadequate to enable modular verification of \( \text{prop_concat} \).

### 5.2.2 Automating Modular Verification

We return to Example 1 to show how Lynx automatically derives the specifications for \( \text{concat} \) and \( \text{append} \) needed to verify \( \text{prop_concat} \).

**Step 1: Verifying \( \text{prop_concat} \)** We call LiquidHaskell to verify \( \text{prop_concat} \), but the verification process fails, due to \( \text{concat} \)'s inadequate (trivial) specification. We next run counterfactual symbolic execution and it returns several abstract counterexamples pinpointing the inadequate specification:

\[ \text{prop_concat} \begin{bmatrix} [] \end{bmatrix} \begin{bmatrix} [] \end{bmatrix} = \text{False} \text{ if } \text{concat} \begin{bmatrix} [], [1] \end{bmatrix} = [1, 1] \]

\[ \text{prop_concat} \begin{bmatrix} [] \end{bmatrix} \begin{bmatrix} [1] \end{bmatrix} = \text{False} \text{ if } \text{concat} \begin{bmatrix} [], [1] \end{bmatrix} = [] \]

Based on those counterexamples, our next goal is to find an adequate specification for \( \text{concat} \).

**Step 2: Synthesizing \( \text{concat} \)'s specification** We use a template based synthesizer to find a specification for \( \text{concat} \). We introduce two uninterpreted predicates \( \text{pre} \) and \( \text{post} \) to describe a specification of \( \text{concat} \):

\[ \text{concat} :: \text{xs} : \{ [[a]] \mid \text{pre}(\text{xs}) \} \rightarrow \{ \text{ys} : [a] \mid \text{post}(\text{xs}, \text{ys}) \} \]
We use the previous counterexamples to define constraints on the specification. The constraints state that if the input values satisfy the preconditions of `concat`, then the input/output pairs should not satisfy the postconditions (because when they do, `prop_concat` cannot be verified):

\[
\begin{align*}
pre (\[\], []) & \implies \neg post (\[\], [], [1]) \\
pre (\[\], [1]) & \implies \neg post (\[\], [1], [])
\end{align*}
\]

Running a synthesizer on these constraints produces the following specification:

```
concat :: xs:{ [[a]] | True} -> ys:{ [a] | size xs \leq size ys}
```

**Step 3: Verifying `prop_concat`:** Having a non-trivial specification for `concat`, we repeat the verification step. Again, verification fails, because the function `concat` could not be verified. We find the following concrete counterexample that does not adhere to its specification:

```
concat (\[\], []) = []
```

If `xs = \[\]` and `ys = []` then `size xs = 2`, but `size ys = 0`. Thus, this counterexample concretely violates the synthesized postcondition that `size xs \leq size ys`.

**Step 4: Synthesizing a new `concat`’s specification:** Using this concrete counterexample, we create a constraint on `concat`’s specification. If the input satisfies the precondition, then the input/output pair must satisfy the postcondition:

\[
\begin{align*}
pre (\[\], []) & \implies post (\[\], [], []),
\end{align*}
\]

With this added to the existing constraints, the synthesizer produces a new `concat` specification:

```
concat :: xs:{ [[a]] | True} \\
-> ys:{ [a] | sumSize xs = size ys}
```

**Step 5: Verifying `prop_concat`:** We again check if we can verify `prop_concat`, using the newly synthesized specification for `concat`. This time, the verifier succeeds. Note that the verification process is not finished. We automatically derived a specification for `concat` that was strong enough to verify `prop_concat`, but we did not verify `concat`. Thus, when we can verify that `concat` adheres to the derived specification, only then we have verified `prop_concat`.

**Step 6: Verification of `concat`:** The function `concat` calls the `append` function. We repeat the process described before and after a couple of steps, we derive the following specification for `append`:

```
append :: xs:[a] -> ys:[a] \\
-> {zs:[a] | size xs + size ys = size zs}
```

This specification is sufficient for the verifier to prove `concat`. Moreover, the specification for `append` is sufficient for an inductive proof of its own correctness, so the verifier just applies its standard verification techniques and proves it correct. At this
point, Lynx terminates.

5.2.3 Reverting Incorrect Choices

In general, the describe technique can be applied to an arbitrary number of callee functions. In our running example there was no need to backtrack, but it is also possible that there is a need to in practice. Given a function $f$ and its callee $g$, it can happen that we derive a specification for $g$ that is sufficient to verify $f$, but that is not correct for $g$. In this case we find a concrete counterexample for $g$. We backtrack one level, add this additional constraint derived from the counterexample to input of the synthesizer, and we repeat the process of finding a specification for $g$ that verifies $f$, but which additionally precludes the found counterexample.

To illustrate that scenario, suppose that, starting from the original Example 1 (with a trivial specification for `append`) we need to prove the following property:

```haskell
prop_append :: [a] -> [a] -> { v:Bool | v }
```

```haskell
prop_append xs ys = size (append xs ys) >= size xs
```

Lynx starts by searching for a candidate specification for `append` that, if true, would allow verifying `prop_append`. Suppose that we found the specification:

```haskell
append :: xs:[a] -> ys:[a] -> { zs:[a] | size zs = size xs}
```

This specification is sufficient to verify `prop_append`. Thus, we proceed to the next level to attempt to verify `append`. There, we find a concrete counterexample:

```haskell
append [1] [1] = [1, 1]
```

If we change the specification of `append` on this level, we risk picking a new specification that is insufficient to show the correctness of `prop_append`. Thus, we instead abandon this level and backtrack. Then, we look for a new specification that both allows for verification of `prop_append`, and accounts for this counterexample. Lynx then finds a specification that satisfies both conditions:

```haskell
append :: xs:[a] -> ys:[a]
-> { zs:[a] | size xs + size ys = size zs }
```

5.3 Formal Foundations

In this section we describe formal foundations of Lynx’s algorithms. Given a program $P$, we formally define the terms described intuitively in the previous sections. Many of those definitions are annotated by a program $P$, as a subscript. However, when it is clear from context, for readability we leave out the subscript. In addition, we describe a desired functionality of external functions used in Lynx’s inference algorithm (verifier, synthesizer, symbolic execution).
5.3.1 Functions and Counterexamples

**Functions** A program, $P$, is a set of functions \( \{f, g, h, \ldots\} \) in a pure language. We assume (without loss of generality) that functions take a single argument. We define \( \text{calls}_P(f) \) as the set of functions directly invoked in $f$’s implementation.

**Specifications** A specification $s$ is a predicate defined over the input and output of a function. In general, a specification $s$ is written in the form $s(i, o) \equiv s^\text{pre}(i) \Rightarrow s^\text{post}(i, o)$. The requirement on the inputs, $s^\text{pre}$, is called the precondition. The postcondition of the function, $s^\text{post}$, is a guarantee about the output.

An environment $E = \{(f_1, s_1), \ldots\}$ maps functions to specifications (but does not indicate that the specifications are correct.) We write $f \in E$ as a shorthand for $\exists s. (f, s) \in E$. We assume that $E$ is a total mapping, and if a function and its specification are not explicitly listed in $E$, we assume that the function has the trivial specification True, i.e. $\text{True} \Rightarrow \text{True}$. To find $f$’s specification in $E$, we use the $\text{lookup}(f, E)$ function. We define the union of two environments, $E_1 \cup E_2$, as follows: for a function $g$, let $\text{lookup}(g, E_1) = s_1$ and $\text{lookup}(g, E_2) = s_2$. Then, $\text{lookup}(g, E_1 \cup E_2) = s$, where $s(i, o) \equiv s^\text{pre}_1(i) \land s^\text{pre}_2(i) \Rightarrow s^\text{post}_1(i, o) \land s^\text{post}_2(i, o)$. Note that if $g$ has a non-trivial specification $s$ in only one of $E_1$ or $E_2$, then $(g, s) \in E_1 \cup E_2$.

**Concrete Counterexamples** Given a program $P$ and environment $E$, modular verification failures are explained by a counterexample $cex$. In general, we denote that $cex$ is a counterexample to the environment $E$ by $cex \not\models_P E$ (we provide more detailed definitions below). Counterexamples can be either concrete or abstract, denoted $cex \not\models_P E$ or $cex \not\models_A E$, respectively.

**Concrete Counterexamples** Concrete counterexamples are concrete demonstrations that a specification is violated. Given a function $f$ and specification $s$ there are two varieties of concrete counterexamples: either the postcondition of $f$ is violated, or the precondition of some callee function of $f$ does not hold.

The first case, a concrete postcondition counterexample, is probably the most standard definition of a counterexample. There is a concrete value $i$ and the output value $f(i)$ that shows that although $i$ satisfies the precondition of $f$, $i$ and $f(i)$ do not satisfy the postcondition of $f$. In other words, $f$’s precondition is insufficient for $f$’s postcondition to hold. Formally, this is stated as $s^\text{pre}(i) \land \neg s^\text{post}(i, f(i))$. We denote this as $(f, i)_C^\text{post} \not\models E$.

The second case is a concrete precondition counterexample. This case involves two functions, $f$ and $g$, such that $f$ calls $g$. This counterexample happens when there is a concrete value $i_f$ that satisfies $f$’s precondition, and to compute $f(i_f)$ we need to invoke $g(i_g)$, but the value $i_g$ violates $g$’s preconditions. Thus, $f$’s preconditions are too weak to ensure that the preconditions of $g$ can hold. Formally, this is stated $s^\text{pre}_f(i_f) \land \neg s^\text{pre}_g(i_g)$. We denote this as $(f, i_f, g, i_g)_C^\text{pre} \not\models E$.

**Abstract counterexamples** We have seen that concrete counterexamples are often insufficient to explain modular verification errors. Thus, we use abstract counterexamples to establish a connection between a caller function and a too weak specification
of a callee function.

The definition of an abstract counterexample has an additional parameter, which we call the abstracted set. Formally, the abstracted set, $F_{\text{abs}}$, is a set of triples $(h, i_h, o_h)$, such that $h(i_h) \neq o_h$, and the pair $(i_h, o_h)$ satisfies the specification of $h$ in $E$. That is, the abstracted set is a set of triples where input/output pairs satisfy the specification of $h$ in $E$, but the output value is not the result of applying $h$ to the input value.

**Example 2.** In Section 5.2.2, in Step 1 there two abstracted sets

$F_1^{\text{abs}} = \{ (\text{concat}, [], [], [1]) \}$

and

$F_2^{\text{abs}} = \{ (\text{concat}, [\text[], [1]], []\} \}$,

each a parameter for one of the two found abstract counterexamples.

Next, we define a modified version of the given program and its environment so that the resulting new environment produces concrete counterexamples, instead of abstract ones. The modified version will be annotated with a superscript, $k$. Consider $F^{\text{abs}} = \{(h_1, i_1, o_1), \ldots (h_n, i_n, o_n)\}$. For each tuple $(h_j, i_j, o_j) \in F^{\text{abs}}$, we define:

$$h^k_j(x) = \text{if } x = i_j \text{ then } o_j \text{ else } h_j(x)$$

Note that for simplicity of presentation we assume that each function $h_j$ appears only once in $F^{\text{abs}}$. If there are more instances of $h_j$ in $F^{\text{abs}}$, the definition can trivially be adapted.

We define the function $f^k = f_{h_1^k/h_1, \ldots, h_n^k/h_n}$, which substitutes each call to $h_j$ in $f$ for the corresponding $h^k_j$. Finally, we define a new environment that overwrites the existing environment and maps each newly defined function to the specification of the function that it is based on:

$$E^k = E[(f^k, \text{lookup}(f, E))/(f, \text{lookup}(f, E)),\hspace{1em}(h^k_j, \text{lookup}(h_j, E))/(h_j, \text{lookup}(h_j, E))],\forall j, 1, \ldots, n.$$  

Similarly as for concrete counterexamples, there are two types of abstract counterexamples: **precondition abstract counterexamples** and **postcondition abstract counterexamples**. For a function $f$, environment $E$ and abstracted set $F^{\text{abs}}$:

- A tuple $(f, i, g, i_g, F^{\text{abs}})^{\text{pre}} \not\models E$ is a precondition abstract counterexample, if its modified version is a precondition concrete counterexample, $(f^k, i, g^k, i_g)^{\text{pre}} \not\models E^k$. The value $i^k$ is a (potentially new) value that function $f^k$ computes that violates the precondition of $g^k$.
A tuple \((f, i, o, F_{\text{abs}})_{A}^{\text{post}} \not\subseteq E\) is a postcondition abstract counterexample, if its modified version is a postcondition concrete counterexample, \((f^{k}, i)_{C}^{\text{post}} \not\subseteq E^{k}\). The output value \(o\) is not needed for the definition, but we pass it as a parameter nevertheless. It will be useful later in the inference algorithm, when we convert each counterexample into a constraint.

Example 3. Consider Example 1 described in Section 5.2 and \(F_{1}^{\text{abs}}\) defined in Example 2. The tuple \((\text{prop_concat}, [], \text{False}, F_{1}^{\text{abs}})_{A}^{\text{post}} \not\subseteq E\) is a postcondition abstract counterexample, because \((\text{prop_concat}^{k}, [])_{C}^{\text{post}} \not\subseteq E^{k}\) is a postcondition concrete counterexample. We show now why is this the case. The original environment contained the following specification for \(\text{prop_concat}\): \(E \supseteq \{(\text{prop_concat}, \text{True} \Rightarrow (\text{output} \leftrightarrow \text{True})\}\). The modified function \(\text{prop_concat}^{k}\) has the same specification. To evaluate \(\text{prop_concat}^{k}\) on \([],\), we first need to compute the output of \(\text{concat}^{k}([], [], \) which is \([1]\). This implies that \(\text{prop_concat}^{k}\) on \([],\) returns \(\text{False}\), which shows that the postcondition of \(\text{prop_concat}^{k}\) is violated. Thus \((\text{prop_concat}^{k}, [])_{C}^{\text{post}} \not\subseteq E^{k}\) is a postcondition concrete counterexample, and \((\text{prop_concat}, [], \text{False}, F_{1}^{\text{abs}})_{A}^{\text{post}} \not\subseteq E\) is a postcondition abstract counterexample.

Other Definitions We use the following functions and predicates. We define the predicate \(\text{evals}_{P}(f, i, f, i, g, i)\) to be true iff evaluating \(f(i)\) requires also evaluating \(g(i)\). The predicates \(\text{isConcrete}(cex)\) and \(\text{isAbstract}(cex)\) are true iff \(cex\) is concrete or abstract, respectively. For each of the counterexamples \((f, i)_{C}^{\text{post}}, (f, i, f, g, i)_{C}^{\text{pre}}, (f, i, o, F_{\text{abs}})_{A}^{\text{post}}, (f, i, f, g, i, F_{\text{abs}})_{A}^{\text{pre}}\) we say the caller is \(f\). We write \(\text{caller}(cex)\) to get the caller of a counterexample.

5.3.2 Verification, Symbolic Execution, and Synthesis

We now describe the required functionality of the tools that our inference algorithm is using.

Verification The verification function, \(\text{verify}(P, E)\), checks if the functions in \(P\) satisfy the specifications in \(E\). If the verifier proves that no specifications is violated, it returns \(\text{Verified}\). If it does not, it returns \(\text{Error}\ err\), where \(err\) is a set of function names that the verifier could not prove correct. A verifier is sound if, whenever there is any type of counterexample, the verifier returns an error set. A verifier is complete if it always either succeeds or returns a non-empty error set.

In this paper, we consider specifically \(\text{modular}\) verifiers. Due to modularity, whether \(f\) is in an error set \(err\) is determined entirely by the specification of \(f\), the definition of \(f\), and the specification of functions called by \(f\).

Counterfactual symbolic execution The goal of the counterfactual symbolic execution function, \(\text{symex}(P, d, E, f)\), is to find concrete or abstract counterexamples with the caller \(f\). The output of the \(\text{symex}\) function is a set of counterexamples. The \(\text{depth}\) argument, \(d\), specifies how many times \(\text{symex}\) should unroll recursive function definitions. The \(\text{symex}\) function is sound if it only returns correct counterexamples.
We call symex complete if, when \( \text{verify}(P,E) = \text{Error} \) and \( f \in \text{err} \), there is some depth \( d \) such that symex returns a counterexample to \( f \), and if whenever the specifications admit a concrete counterexample, there is some depth such that one is found.

**Constraints and Theories** We use counterexamples to determine the constraints that the specifications should satisfy. Independently of their type, a counterexample contains an arbitrary number of functions and some concrete values, meaning that those constraints are always ground formulas. The process of deriving the constraints will be described in detail in Section 5.4. In general, those constraints are ground first order logic formulas defined over over uninterpreted predicates \( s_{\text{pre}} \) and \( s_{\text{post}} \) for functions \( f \) appearing in a counterexample. An example of such a constraint is: from a concrete counterexample \( (\text{append}, [1][1]) \), we construct a constraint \( c_1: \)

\[
\text{append}([1], [1]) \Rightarrow \text{append}([1], [1], [1])
\]

An environment \( E \) is a model for a constraint \( c \), if instantiating each uninterpreted predicate \( s_f \) in \( c \) with its corresponding formula in \( E \) results in the formula that evaluates to true. We write this as \( E \models c \). If the environment \( E \) contains the pair \( (\text{append}, \text{True} \Rightarrow \text{size } xs + \text{size } ys = \text{size } zs) \), then \( E \models c_1 \). We lift this definition to sets of constraint, \( C \), in the obvious way, i.e. \( E \models C \Leftrightarrow \forall c \in C. E \models c \).

Functions and predicates used to define specifications belong to a theory, or specification language, \( T \). We say the environment \( E \) draws from the theory \( T \), and write \( E \subset T \), if all specifications in \( E \) are also in \( T \). We will particularly use a predicate \( \text{isSat}_T(E_U, E_S, C) \) which check if there exists some function specifications in the theory \( T \) that could extend \( E_U \cup E_S \) to satisfy the constraints \( C \), without adding to or modifying the constraints of functions already in \( E_S \):

\[
\text{isSat}_T(E_U, E_S, C) \Leftrightarrow \exists E_N \subset T. \forall f. (f \in E_N \Rightarrow f \notin E_S) \land E_U \cup E_S \cup E_N \models C
\]

One can think about this predicate as follows – there are two environments: one containing the user defined specifications, \( E_U \), and the other containing the synthesized functions and their specifications, \( E_S \). Given a set of constraints \( C \), our goal is to find a new environment \( E_N \) such that the union of all three environments is a model for \( C \). We differentiate the environments \( E_U \) and \( E_S \), because we cannot modify already synthesized specifications, and thus the requirement \( \forall f. (f \in E_N \Rightarrow f \notin E_S) \). We do not impose the same restriction on the user defined specification: a specification of \( f \), if \( f \in E_U \), can also appear in \( E_N \). This way, we do not overwrite, but strengthen the user given specification. This predicate can be seen also as an indicator whether we are on the right path towards fully verifying the program.

**Synthesis** A synthesis function, \( \text{synth}_T(s_f, E_U, E_S, C) \), takes as the input a set of functions \( s_f \), environments \( E_U \) and \( E_S \), and a set of constraints \( C \). The goal of the synthesis function is to find a specification for every function in \( s_f \), such that \( \text{isSat}_{\text{UF}}(E_U, E_S \cup E_{s_f}, C) \) (where UF denotes the theory of uninterpreted functions).
holds for the resulting environment $E_{sf} \subset \mathcal{T}$. If the synthesis process succeeds, it returns $\text{SynthEnv } E_{sf}$. When the synthesis process fails, it returns $\text{SynthFail } I$, where $I$ is an interpolant of $E_U$, $C$ and $E_S$. Note that $C$ is a formula, while $E_S$ and $E_U$ are environments, so we need to adopt the standard definition of an interpolant.

If the synthesis failed, we know that it holds that $\neg \text{isSat}_T(E_U, E_S, C)$: the environment $E_U \cup E_S$ cannot be extend to be a model for $C$. A typical view on an interpolant, as an unsatisfiable core of a formula, can also be mimicked in these settings, too. We can picture an interpolant as a subset of constraints in $C$ that are the root cause why the environments cannot be extended to be a model for $C$. In addition to disjointness to $E_U \cup E_S$, we need to specify that an interpolant also overapproximates $C$. If there would be no theory constraints and if $C$ and $I$ would belong to the same theory, the overapproximation requirement could be simply stated as $C \Rightarrow I$. In our setting, we express this requirement as follows: every environment that can be extended to be a model for $C$ can also be extended to be a model for $I$.

Formally, an interpolant of $E_U$, $C$ and $E_S$ is a constraint $I$ that satisfies the following formula:

$$\forall E_G. \text{isSat}_T(E_U, E_G, C) \Rightarrow \text{isSat}_T(E_U, E_G, I) \land \neg \text{isSat}_T(E_U, E_S, I)$$

We say a synthesis function $\text{synth}_T(sf, E_U, E_S, C)$ is sound if whenever it returns an environment $E_{sf}$, all specifications in $E_{sf}$ are in $\mathcal{T}$, and $\text{isSat}_T(E_U, E_S \cup E_{sf}, C)$. We say a synthesis function is complete if it always returns such an environment when one exists.

**Size-bounded theory** For a theory $\mathcal{T}$, we define a function $\text{size}$, mapping the specifications in $\mathcal{T}$ to $\mathbb{N}$. We require that, for all $n \in \mathbb{N}$, there are only a finite number of $s \in \mathcal{T}$ such that $\text{size}(s) < n$. Given some $\text{size}$, we overload $<$, so that for $s_1, s_2 \in \mathcal{T}$, $s_1 < s_2$ iff $\text{size}(s_1) < \text{size}(s_2)$. We define a bounded theory $\mathcal{T}^k$ to consist of the specifications in $\mathcal{T}$ with size at most $k$.

This allows us to consider size-bounded synthesizers $\text{synth}_{\mathcal{T}^k}(sf, E_U, E_S, C)$ which synthesize specifications in a bounded theory. That is, a size-bounded synthesizer $\text{synth}_{\mathcal{T}^k}$ will only synthesize an environment in which every specification has size at most $k$. Formally, we can state this property as: $\text{synth}_{\mathcal{T}^k}(sf, E_U, E_S, C) = \text{SynthEnv } E_{sf} \implies \forall (f, s) \in E_{sf}. \text{size}(s) \leq k$.

**External and internal functions** We split the functions into external and internal functions. External functions represent interfaces provided by an API or module. Internal functions are used to implement the external functions. We write $\text{external}(P)$ and $\text{internal}(P)$ to get the disjoint sets of external and internal functions in $P$.

During the inference algorithm, the specifications of external functions is fixed. If the specifications are incorrect, a counterexample is reported. We allow adding specifications only to internal functions. We assume (without loss of generality, since functions could be duplicated) that external function are not called by any internal or external function.
**Call Graph** Consider the _call graph_ of a program. We define a function \( \text{level}(f) \) as the shortest distance from an external function to \( f \).

## 5.4 Inference

Now, we describe the inference algorithm. In Section 5.4.1, we show the inference algorithm, and prove its soundness and completeness for finite specification languages. In Section 5.4.2, we will show how to apply iterative deepening to achieve completeness for an infinite specification language.

### 5.4.1 Inference Algorithm

Algorithm 5.1 shows the inference algorithm. \textit{initInfer}_\( T \) is provided as an easy way to call the main loop. It takes a program \( P \) and user written specifications (or the user environment) \( E_U \), and it returns either an additional environment \( E_S \), which supplements \( E_U \) to allow verification, or a concrete counterexample to the specifications in \( E_U \).

\( \text{traverseCG}_T \) and \( \text{generateSpec}_T \) decompose finding a correct \( E_S \) into two pieces. \( \text{traverseCG}_T \) walks over the levels of the program call graph, beginning at the external functions (level 0), and continuing until we reach the functions furthest from the external functions (the greatest level). At each level \( L \), \( \text{traverseCG}_T \) calls \( \text{generateSpec}_T \) to search for a set of specifications for the functions at level \( L + 1 \) that, if true, would be sufficient to verify the specifications of the functions at level \( L \).

**Inputs, Output, and Invariants** In addition to \( P \) and \( E_U \), both \( \text{traverseCG}_T \) and \( \text{generateSpec}_T \) take four additional arguments. \( fs \) is the set of functions at the current level \( L \). \( E_S \) is a specification environment, containing specifications already synthesized for functions in levels 0 to \( L \). \( C \) and \( C_{sz} \) are two sets of specification constraints, which must be satisfied by any synthesized specifications. \( C \) is the _counterexample constraint set_, and contains constraints from generated counterexamples. \( C_{sz} \) is the _synthesizer constraint set_, and contains constraints from the synthesizer returning SynthFail. The need to separate these two sets of constraints will be explained in Section 5.4.2. In addition to the arguments passed to \( \text{traverseCG}_T \), \( \text{generateSpec}_T \) takes one additional argument: \( sf \), the set of functions in the level \( L + 1 \), which must be assigned specifications.

Both functions return one of three constructs. First, they may return \( \text{SEnv}E_SE_{sz} \). When returned from \( \text{traverseCG}_T \) this indicates that \( E_S \) is sufficient to verify the specifications in \( E_U \). When returned from \( \text{generateSpec}_T \), it indicates that, if the specifications in \( E_S \) could themselves be verified they would be sufficient to verify the specifications in \( E_U \). \( C \) and \( C_{sz} \) are the constraints used while synthesizing \( E_S \).

Second, the functions may return \( \text{CEx}cexs \). This indicates that the counterexamples in \( cexs \) are concrete counterexamples to a specification in \( E_U \) with external functions as callers. Finally, the functions may return \( \text{Raise}C_{sz} \). This indicates that there
Algorithm 5.1: traverseCG\(_T\) synthesizes specifications for use by a modular verifier.

is no specification environment drawn from \(T\) that would allow verification of the current \(E_U\) and \(E_S\), and thus we must back track and find a new \(E_S\). As in the SEnv constructor, \(C\) and \(C_{sz}\) are two constraint sets for use at the previous level- in particular, \(C_{sz}\) must always contain an interpolant, to block resynthesizing the same environment.

At each call to traverseCG\(_T\) and generateSpec\(_T\), we maintain an invariant:

**Invariant 1.** Let \(L\) be the level of \(f \in fs\). Any functions in the verifier error set must have a level greater than or equal to \(L\). That is, \(\text{verify}(P, E_U \cup E_S) = \text{Error}_{\text{err}} \implies \forall f \in \text{err}. L \leq \text{level}(f)\).

To maintain this invariant, we *never* change a specification in \(E_S\). Instead, we backtrack in traverseCG\(_T\) to revert incorrect or inadequate specifications. If in traverseCG\(_T\) the functions in \(fs\) are at level \(L\), then generateSpec\(_T\) will assign specification to *exactly* the functions at level \(L+1\) of the call graph. By the verifier’s modularity, these specifications cannot affect whether verify reports an error in any function in levels 0 to \(L-1\). Further, we call traverseCG\(_T\) with \(fs\) containing the
functions in $L+1$ only if the verifier’s error set contains none of the functions at level $L$. Thus, the invariant is maintained.

Thus, as $\text{traverseCG}_T$ walk over the call graph, we need only concern the algorithm with synthesizing specifications that eliminate the errors at the current level. Assuming a sound modular verifier, if we traverse all the level of the call graph, and no functions in the final (largest) level are in the error set, we will have found a specification environment $E_S$ that is sufficient to verify $E_U$.

In the following, we detail the operation of both $\text{generateSpec}_T$ and $\text{traverseCG}_T$. Then, we will present soundness and completeness theorems.

$\text{traverseCG}_T$ - Top down search $\text{traverseCG}_T$ walks down the levels of the call graph of the program. At each level, $\text{traverseCG}_T$ calls $\text{generateSpec}_T$ to try to synthesize new specifications that eliminate all functions in $fs$ from the error set returned by the verifier. At line 6, $\text{traverseCG}_T$ sets $sf$ to be the functions to synthesize specifications for. It might seem tempting to include in $sf$ all functions called by a function $fs$. However, to maintain Invariant 1, we exclude functions already in $E_S$ from $sf$. If this makes it impossible to find a set of specifications to verify $sf$, we rely on the $\text{Raise}$ constructor being returned.

We now consider what happens when $\text{generateSpec}_T$ at line 7 returns:

Synthesis Succeeds Assuming synthesis succeeds, we reach line 8 having found some $E'_S$ containing specifications for the functions in $sf$, such that, if the specifications of $sf$ in $E'_S$ are true, so are the specifications in $E_S$. We also have new constraint sets $C''$ and $C''_{sz}$, gathered while synthesizing $E'_S$. In order to verify the full program, we must now verify the newly synthesized specifications in $E'_S$. To do this, we make a recursive call, at line 11, to $\text{traverseCG}_T(P,E'_U,fs,E'_S,C,C'_{sz})$.

If this recursive call succeeds in producing a specification environment $\text{SEnv}E'_S C'' C'_{sz}$, or produces concrete counterexamples $\text{CEx}e$ to $E_U$, this result is directly returned. If the recursive call returns $\text{Raise} C'' C'_{sz}$, then we recall $\text{traverseCG}_T$ to find new specifications for the functions in $sf$ with the new constraint sets $C''$ and $C'_{sz}$ in place of our previous constraint sets.

Synthesis Fails Now, suppose that synthesis failed, and $\text{generateSpec}_T$ returned either a counterexample or a $\text{Raise}$ constructor. Then, the result of $\text{generateSpec}_T$ will be returned by $\text{traverseCG}_T$, at line 14. $\text{Raise}$ constructors may be caught at some smaller level, at line 12, and, as previously described, returned concrete counterexamples will be passed back to the user.

$\text{generateSpec}_T$ - Synthesizing Specifications To synthesizes specifications for the functions in $sf$, we define $\text{generateSpec}_T(P,E_U,fs, sf, E_S, C, C_{sz})$. If an environment is returned, that environment is sufficient to eliminate the functions in $fs$ from the error set return by $\text{verify}$. At line 17, $\text{generateSpec}_T$ calls $\text{synth}_T(sf, E_U, E_S, C \cup C_{sz})$ to attempt to synthesize a specifications for the functions in $E_S$. There are two possibilities:

Synthesis Fails If $\text{synth}_T$ returns $\text{SynthFail} C'_{sz}$, then $\text{generateSpec}_T$ returns $\text{Raise} C' C'_{sz}$. This will be caught in $\text{traverseCG}_T$, allowing $C'_{sz}$ to constrain a search
Counterexample

\((f, i, o, F_{abs})^\text{post}_A\)

Constraint

\[ s^\text{pre}_f(i) \land \neg s^\text{post}_f(i, o) \implies \bigvee_{(g, i, o) \in F_{\text{abs}}} (s^\text{pre}_g(i) \implies \neg s^\text{post}_g(i, o)) \]

\((f, i, g, i, F_{abs})^\text{pre}_A\)

\[ s^\text{pre}_f(i) \land \neg s^\text{pre}_f(i, g) \implies \bigvee_{(h, i, o) \in F_{\text{abs}}} (s^\text{pre}_h(i) \implies \neg s^\text{post}_h(i, o)) \]

\((f, i)^\text{post}_C\)

\(\begin{aligned} &\text{if } f \in \text{external}(P) \\ &\text{otherwise} \end{aligned}\)

Return Counterexample

\(s_f(i, f(i))\)

\((f, i, g, i, g, F_{\text{abs}})^\text{pre}_C\)

\(\begin{aligned} &\text{if } f \in \text{external}(P) \\ &\text{otherwise} \end{aligned}\)

\(\begin{aligned} &s^A_g = \text{lookup}(g, E_U) \\ &s^A_{\text{pre}}(i) \\ &s^A_{\text{pre}}(i, g) \end{aligned}\)

Return Counterexample

\(\begin{aligned} &s^\text{pre}_f(i) \\ &s^\text{pre}_f(i, f) \implies s^\text{pre}_g(i) \end{aligned}\)

\(\text{Figure 5.3: evalCE}(E_U, cex)\) maps counterexamples to either constraints on the specifications, or returns a counterexample if it indicates a flaw in the users specification.

for a different environment in smaller levels. By definition of a synthesizer, \(C'_{sz}\) is an interpolant of \(C \cup C_{sz}\) and the specifications in \(E_U \cup E_S\). Thus, we are guaranteed to not every retry the same (incorrect) environment \(E_S\).

**Synthesis Succeeds** Now suppose \(\text{synth}_T\) returns an environment \(E_N\). To maintain Invariant 1, the algorithm checks if the functions in \(fs\) are in the error set of \(\text{verify}(P, E_U \cup E_S \cup E_N)\). If they are not, the algorithm returns \(E_S \cup E_N\). Otherwise, it searches for counterexamples showing why verification failed.

To check sufficiency of \(E_N\), we define \(\text{verifyCEx}(P, E, fs)\). \(\text{verifyCEx}\) first calls \(\text{verify}(P, E)\). If \(\text{verify}\) return \text{Verified}, there are no counterexamples, and the empty set is returned. Otherwise, \(\text{verify}\) returns some \text{Error} \(err\). Since we are concerned only with errors at our current level (involving functions in \(fs\)) we run \(\text{symex}\) on the functions in \(err \cap fs\). Assuming a complete \(\text{symex}\) function, if \(err \cap fs\) is nonempty, we can be sure at least one counterexample will be found.

If \(\text{verifyCEx}\) returns an empty set, then the synthesized specification environment \(E'_S\) would be sufficient to verify the specifications of \(fs\). Thus, we simply return \(E'_S\). Otherwise, we use \(\text{evalCE}\) to convert the counterexamples into constraints, to be used in the next round of synthesis (or, to determine that show a specification in \(E_U\) is false, in which case they are reported to the user.)

**Generating Constraints** Figure 5.3 outlines \(\text{evalCE}\), which turns counterexamples into constraints. The constraints satisfy two keys properties. First, they are \text{strong}.
enough to block the synthesizer from ever resynthesizing the same specification environment. This ensures we make progress, and do not loop with the same environment in \texttt{generateSpec}_{\mathcal{T}}. Second, they are weak enough to not block any correct environments. That is, any environment that does not satisfy the constraint must also allow either a concrete or abstract counterexample. This ensures that our synthesizer will not miss a correct specification environment.

**Soundness and completeness**

We now give definitions of soundness and completeness for an inference algorithm. We then state the soundness and completeness theorems for Algorithm 5.1

**Definition 1** (Soundness). Consider a verifier \texttt{verify}, along with a program \(P\) and environment \(E_U\). An inference function is sound if (1) when it produces an environment \(E_S\), \texttt{verify}(\(P, E_U \cup E_S\)) = \texttt{Verified}, and (2) when it produces a counterexample \(cex\), \texttt{caller}(cex) \in \texttt{external}(P) \text{ and } cex \not\in E_U.

**Definition 2** (Completeness). Consider a verifier \texttt{verify} and synthesis function \texttt{synth}_{\mathcal{T}}, along with a program \(P\) and environment \(E_U\). An inference function is complete if (1) when there exists a environment \(E_S\) drawn from \(\mathcal{T}\) such that \(n \in \texttt{external}(P) \implies n \not\in E_S\) and \texttt{verify}(\(P, E_U \cup E_S\)) = \texttt{Verified} the inference function terminates with an environment, and (2) when there exists a concrete counterexample to an external function the inference function produces a concrete counterexample.

**Theorem 2** (Sound of \texttt{initInfer}_{\mathcal{T}}). Given sound \texttt{verify}, \texttt{symex}, and \texttt{synth}_{\mathcal{T}} functions, \texttt{initInfer}_{\mathcal{T}}(\(P,E_U\)) is sound.

*Proof Sketch.* To return some \(E'_S\), execution must reach line 10, so \(sf = \{\}\), and we have reached the highest level in the call graph. By Invariant 1, there are no errors in a smaller level, and since \texttt{generateSpec}_{\mathcal{T}} returned an environment there are no errors in the highest level. Thus, \texttt{verify}(\(P,E'_S\)) = \texttt{Verified}. If a concrete counterexample is returned, the soundness of \texttt{initInfer}_{\mathcal{T}} follows from the soundness of \texttt{symex}.

**Theorem 3** (Completeness of \texttt{initInfer}_{\mathcal{T}}). Suppose there are a finite number of specifications in \(\mathcal{T}\). Then, given sound and complete \texttt{verify}, \texttt{symex}, and \texttt{synth}_{\mathcal{T}} functions, \texttt{initInfer}_{\mathcal{T}}(\(P,E_U\)) is complete.

*Proof Sketch.* The algorithm repeatedly blocks incorrect environments. Since all constraints come from \texttt{evalCE} or an unrealizable call to \texttt{synth}_{\mathcal{T}}, by \texttt{evalCE}'s properties and the definition of an interpolant no correct environment will be blocked. Since there are only a finitely many specifications, if there is an environment such that verification can succeed, it is eventually returned.

If there is a concrete counterexample, the algorithm return it at line 23. Otherwise, all environments will be blocked, and the complete \texttt{symex} function will run to an infinite depth at line 2.

Full proofs of both Theorem 2 and 3 are provided in Appendix A.1.3.
iterateInfer\(_{T^k}(P, E_U, d, C) = \)
case \(\text{traverseCG}_{T^k}(P, E_U, \text{external}(P), \{\}, C, \{\})\) of
  SEnv \(E_N C^C C'_{sz} \rightarrow \text{SynthEnv} E_N C'C'_{sz}\)
  CEx\(cexs \rightarrow \text{CEx cexs}\)
  Raise \(C_N \rightarrow \text{case } \cup_{f \in \text{external}(P)} \text{symex}(P, E_U, d, f) \text{ of } \{\} \rightarrow \text{iterateInfer}_{T^{k+1}}(P, E_U, d+1, C_N)\)
  \(cexs \rightarrow \text{CEx cexs}\)

Algorithm 5.2: iterateInfer\(_{T^k}\) uses \(\text{traverseCG}_{T^k}\) to synthesize environments in \(\cup_k T^k\).

5.4.2 Iteratively Deepening Inference Algorithm

\(\text{initInfer}_T\) is only complete for finite specification domains. In Figure 5.2, we show the iteratively deepening \(\text{iterateInfer}_{T^k}\) algorithm. This algorithm uses a size-bounded synthesizer (Section 5.3.2) to search over an infinite set of specifications, while preserving both soundness and completeness.

**Input and Output** As with \(\text{initInfer}_T\), \(\text{iterateInfer}_{T^k}\) takes a program \(P\) and user environment \(E_U\) as arguments. It also accepts a depth \(d\), used by \(\text{symex}\) to search for counterexamples, and a set of constraints, which should be initialized to empty. The return values are the same as those of \(\text{initInfer}_T\).

**Finding specification environments** \(\text{iterateInfer}_{T^k}\) relies on \(\text{traverseCG}_{T^k}\) to synthesize specifications. First, \(\text{iterateInfer}_{T^k}\) calls \(\text{traverseCG}_{T^k}\) with some fixed \(k\). If \(T^k\) is a size-bounded theory, there are a finite number of specifications in \(T^k\). Thus, if there exists an environment drawn from \(T^k\) such that verification would succeed, Theorem 3 guarantees that \(\text{traverseCG}_{T^k}\) will find it. Then, \(\text{iterateInfer}_{T^k}\) can simply directly return the found environment. Otherwise, if \(\text{traverseCG}_{T^k}\) returns \(\text{Raise}\), then \(\text{iterateInfer}_{T^k}\) calls \(\text{iterateInfer}_{T^{k+1}}\), increasing the synthesizer’s size-bound. Thus, if an environment of some size enables verification, \(\text{iterateInfer}_{T^k}\) will eventually find it.

**Preserving Constraints** When \(\text{iterateInfer}_{T^{k+1}}\) is called, it is passed all counterexample constraints from \(\text{traverseCG}_{T^k}\). Thus, it does not start from scratch— it uses constraints learned while searching environments of smaller sizes. This is sound, because the process to turn counterexamples into constraints does not rely on the theory being considered by the synthesizer.

Conversely, synthesizer constraints might depend on the theory, and therefore be invalidated by an increased size bound. This is the motivation for the separate tracking of synthesizer constraints \(C_{sz}\). When we call \(\text{iterateInfer}_{T^{k+1}}\), we discard the synthesizer constraints, as we do not know if they are still relevant.

**Finding counterexamples** Supposing there is a concrete counterexample to the specifications in \(E_U\), there are two ways it may be found. First, it may be returned from \(\text{traverseCG}_{T^k}\). Second, each time \(\text{traverseCG}_{T^k}\) returns a \(\text{Raise}\) constructor, \(\text{symex}\) is used to search to depth \(d\). This depth is increased on each call to \(\text{iterateInfer}_{T^k}\), which is sufficient to ensure (given a complete \(\text{symex}\) function) eventually a concrete counterexample will be found.
Theorems Theorems 4 and 5 follow from the corresponding theorems for \( \text{initInfer}_T \):

**Theorem 4** (Soundness of \( \text{iterateInfer}_{T_1} \)). Consider a size-bounded theory \( T \). Given sound \( \text{verify}, \text{symex}, \) and \( \text{synth}_{T_k} \) functions, \( \text{iterateInfer}_{T_1}(P, E_U, 1, \{\}) \) is sound.

**Theorem 5** (Completeness of \( \text{iterateInfer}_{T_1} \)). Consider a size-bounded theory \( T \). Given sound and complete \( \text{verify}, \text{symex}, \) and \( \text{synth}_{T_k} \) functions, \( \text{iterateInfer}_{T_1}(P, E_U, 1, \{\}) \) is complete over specification in \( \bigcup_k T_i \).

Proofs of both Theorem 4 and 5 are provided in Appendix A.1.3.

### 5.4.3 Optimizations and Heuristics

Our completeness theorems rely on a complete \( \text{symex} \) function, but in practice \( \text{symex} \) does not have a completeness proof. Fortunately, we can work around this, and so \( \text{generateSpec}_T \) and \( \text{iterateInfer}_T \) are still complete.

**\( \text{verifyCEx} \) Level Descent** Consider a function \( f \) at level \( L \) that calls a function \( g \) at level \( L + 1 \). Suppose we are trying to verify the functions at level \( L \). If we have synthesized a candidate environment \( E_N \), but \( \text{verify}(P, E_U \cup E_S \cup E_N) = \text{Error err}; f, g \in err \), then we call \( \text{verifyCEx} \) to attempt to find a counterexample to \( f \). Any \( \text{symex} \) function relies on heuristics to search through the state space of a function. Thus, rather than searching for a counterexample to \( f \), it might be quicker to speculate that \( g \)'s newly synthesized specification is wrong, and try to find a concrete counterexample to \( g \). Despite \( g \) not being in level \( L \), a concrete counterexample to \( g \) is sufficient to block the environment \( E_N \) at level \( L \). Motivated by this, we use a timeout when searching for counterexamples to \( f \). If no counterexamples are found within the timeout, we switch to searching for concrete counterexamples to functions in the error set called by \( f \).

**Model negation** As a last resort, we fall back on directly blocking the incorrect specifications in the synthesizer. In Section 5.5 we will see that the LIA synthesizer is based on an SMT encoding, so this is doable by adding the negation of the SMT model as a constraint. In general, this is an impractically slow way to narrow the specification space. However, we have found that it is occasional useful to jolt the synthesizer out of a problematic area for the (incomplete) \( \text{symex} \) function. It is also sufficient to maintain completeness of \( \text{generateSpec}_T \) and \( \text{iterateInfer}_T \).

**Constraint Generation** The inference algorithm relies on turning counterexamples into blocking constraints, i.e. constraints that block some specific \( E_S \). These constraints ensure that we do not repeatedly synthesize the same environment. However, we can also extract non-blocking constraints, which sometimes help in guiding synthesis. For example, for an abstract counterexample, we generate extra constraints by concretely executing each abstracted function on its input.
5.5 Size-Bounded Linear Integer Arithmetic Synthesizer

Completeness of the iterative synthesis in Section 5.4.2 requires a size-bound synthesis function. In Section 5.5.1, we describe how we can build such a function, \( \text{synth}_{\text{LIA}}^k \), for the infinite space of LIA specifications. We consider functions with inputs that are a tuple of integers, \((x_1, \ldots, x_k)\), and which output an integer \( r \). In Section 5.5.2, we will describe a synthesizer \( \text{synth}_{\text{LH}}^k \) for LiquidHaskell’s refinement types specifications (including features such as measures and specifications over polymorphic data structures) via a reduction to a synthesis problem solvable by \( \text{synth}_{\text{LIA}}^k \).

5.5.1 Synthesizing LIA specifications

Sizes of LIA formulas A size bounded synthesizer for LIA formulas requires a size function for LIA formulas. \( \text{size} \) must map LIA formulas to the natural numbers, and ensure that for all sizes \( s \), only finitely many formulas have size less than \( s \). Consider a conjunction of \( i \) LIA formulas \( F \) with coefficients \( c_0, c_1, \ldots, c_n \). We pick some \( d \), then let \( c_m = \max(\lceil |c_0| d \rceil, \lceil |c_1| d \rceil, \ldots, \lceil |c_n| d \rceil) \). We define the size of such a formula as \( \max(c_m, i) \). Thus, we have a minimal size of 1, and the number of formulas less than any size is finite.

Specification Templates We use templates to implement a size bounded synthesizer, \( \text{synth}_{\text{LIA}}^k(sf, E_U, E_S, C) \). The constraints and templates are encoded in the decidable linear integer arithmetic with uninterpreted functions logic [141] which is then solved with an SMT solver. As described in Section 5.3.1 \( \text{synth}_{\text{LIA}}^k(sf, E_U, E_S, C) \) synthesizes an environment \( E_N^f \) for the functions in \( sf \) written in the language LIA\(^k\), such that \( \text{isSat}_{U}(E_U, E_S \cup E_N^f, C) \). We introduce two uninterpreted functions for each specification to capture its existing pre and postcondition, based on \( E_U \) and \( E_S \). In addition, we introduce multiple variables for each function in \( sf \), to form a template to synthesize its pre and postcondition.

Encoding Existing Specifications First, we encode the existing behavior of the specification environments \( E_U \) and \( E_S \). By definition of the \( \text{synth} \) function we are looking for an environment \( E_N \) such that \( \text{isSat}_{U}(E_U, E_S \cup E_N, C) \). Thus, we are looking for some \( E_N \) such that:

\[
\exists E'_N \subset UF. (n \in E'_N \implies n \notin E_S \cup E_N) \land E_U \cup E_S \cup E_N \cup E'_N \models C
\]

Thus, in synthesizing \( E_N \), the specifications in \( E_S \) are unchangeable. In contrast, the specifications in \( E_U \) for functions not already in \( E_S \) are inflexible only where they evaluates to false.

To capture this in the synthesizer, for each \( f \in E_U \cup E_S \) we define uninterpreted functions \( F_{U,f}^{\text{pre}} \) and \( F_{U,f}^{\text{post}} \). Then, we concretely check each pre and postcondition in \( E_U \) and \( E_S \) against each input/output pair in a constraint. If the condition-
either environment - evaluates to false, we assert that the appropriate uninterpreted function is false for that input/output. If the specification is true and in $E_S$ we assert that the functions are true for that input/output. Otherwise we make no assertion about the functions, as their value will depend on some future synthesis query.

**Synthesizing New Specifications** Now, we consider synthesizing specifications for functions $f$ in $sf$, using LIA specifications templates of up to size $k$. Suppose $f$ has arity $n$. We must generate a template that can yield any conjunction of up to $k$ LIA formulas, with largest coefficient at most $\frac{k}{d}$, over the variables $x_1, \ldots, x_n, r$. From our examples, we know values for $x_1, \ldots, x_n, r$, and we must find coefficients that satisfy the constraints. We need to find a precondition and a postcondition. Each pre or postcondition has an associated LIA specification template. We introduce integer variables to represent the coefficients in each of the LIA formulas in our template. We also introduce three boolean variables per LIA formula. One of the booleans, $b_{off}^i$, is or-ed with the formula, to control whether the formula is used in the specification. One, $b_{eq}^i$ controls whether we have an equality or inequality. The final boolean, $b_{str}^i$, is used in the case of an inequality to control if that inequality is strict. As an example, a postcondition formula’s equation will have the form

$$\bigwedge_{i=1}^k (b_{off}^i \lor (\text{if } b_{eq}^i \text{ then } (c_{0}^i + c_{1}^i x_1 + \ldots + c_{n}^i x_n + c_{n+1}^i r = 0) \text{ else if } b_{str}^i \text{ then } (c_{0}^i + c_{1}^i x_1 + \ldots + c_{n}^i x_n + c_{n+1}^i r \leq 0) \text{ else } (c_{0}^i + c_{1}^i x_1 + \ldots + c_{n}^i x_n + c_{n+1}^i r < 0)))$$

To apply the formula to an input/output pair, we plug the concrete input/output values into the formula as $x_1, \ldots, x_n$ and $r$. Even though the coefficients are variables, the inputs and outputs are known integers. Thus, this formula is in the LIA theory. To enforce that each coefficient is not larger than allowed by $d$, we add an additional constraint per coefficient that $|c_{ij}| \leq d \cdot k$. Since $d$ and $k$ are both constants, $d \cdot k$ is also a constant. Thus, these constraints also are in the LIA theory.

**Translating Constraints** Once we have the correct uninterpreted functions for each specification, we can encode the constraints. As the constraints are already in first order logic, this is straightforward. We replace each specification constraint, with an appropriate combination of $F_{U,pre}^j$ and $F_{U,post}^j$, and, in the case of a specification in $sf$, a specification template.

**Unrealizability and interpolants of LIA formulas** We require that, if the provided synthesis problem is unrealizable, synth$_{LIA}(sf, E_U, E_S, C)$ return SynthFail$C'$, where $C'$ is an interpolant of $C$ and $E_U \cup E_S$. Determining that there is no solution less than or equal to some given size is straightforward: we can check the satisfiability of the LIA formula given solutions of at most that size. Once we have concluded that a formula is unsatisfiable, we must obtain an interpolant of $C$ and $E_U \cup E_S$. That is,
we must find some \( C' \) such that
\[
\forall E. \text{isSat}\text{LIA}_k(E_U, E_G, C) \implies \text{isSat}\text{LIA}_k(E_U, E_G, C')
\]
\[
\neg \text{isSat}\text{LIA}_k(E_U, E_S, C \cup C')
\]

We find \( C' \) by extracting the unsat core of the LIA formula, and negating the conjunction of the formulas in the unsat core that originated from the true or false assignment to \( F_{U,pre} \) and \( F_{U,post} \).

We now check that this satisfies the two requirements of an interpolant. Satisfying the first requirement requires us to consider all \( E_G \) such that \( \text{isSat}\text{LIA}_k(E_U, E_G, C) \). Clearly, all such \( E_G \) must differ from \( E_U \cup E_S \) at one or more of the points that led to \( \neg \text{isSat}\text{LIA}_k(E_U, E_S, C) \). Thus, the first requirement is satisfied. Regarding the second requirement, we note that the negation of the conjunction of the unsat core is forced to be false by \( E_U \cup E_S \) (by construction.) Thus, \( \neg \text{isSat}\text{LIA}_k(E_U, E_S, C') \) holds.

**MaxSMT** If unconstrained, the LIA synthesizer often overfits the counterexamples. We use Z3’s [64] soft assertions to find formulas that are more likely to generalize. Z3 finds a solution which satisfies all hard assertions, and as many soft assertions as possible. We use this to minimize the number of non-zero coefficients, thus generating simpler formulas. We also favor coefficients of -1, 0, or 1, as, in our experience, these arise more in real world assertions.

**Disjunctions** Although the described synthesizer only finds conjunctions of LIA formulas, extending it to also find disjunctions is straightforward. Theoretically, we adjust the definition of size for LIA formulas, so that a LIA formula has size \( i \) if it is a conjunction of \( i \) LIA \( i \)-clauses, where a LIA \( i \)-clause is a disjunction of \( i \) LIA formulas. Practically, we adjust the template so that at size \( i \), rather than having \( i \) conjoined LIA formula templates, one has \( i \) conjoined LIA \( i \)-clause templates. Our practical implementation, Lynx, finds both conjunctions and disjunctions of LIA formulas.

### 5.5.2 Refinement Types Synthesis

Now, we turn our attention to the design of synth\text{LH}^k, which builds on the synth\text{LIA}^k synthesizer to construct refinement types. We begin with a (highly simplified) description of the LiquidHaskell refinement type syntax. Then, we will detail the design of our synthesizer.

Figure 5.4 shows a limited subset of LiquidHaskell’s grammar [197]. A LiquidHaskell type \( \tau \) is either a refined type or a function type. A refined type \( \{ x : D \mid r \} \) refines a regular Haskell type (such as \text{Int} or \text{[a]}), with a predicate (or refinement) \( r \). In order to type check, values of the regular Haskell type are required to satisfy \( r \). For instance \( \{ x : \text{Int} \mid x > 0 \} \) states that the Int value \( x \) is required to be positive. Function types, \( x : \tau_1 \rightarrow \tau_2 \), assign refinement types to functions. The binding \( x \) on the argument type can be used in the refinement in the return type \( \tau_2 \).
\( \tau ::= \{ v : D \tau | r \} \) refined type
\( \tau ::= x : \tau \to \tau \) function type
\( r ::= x \) variable
\( r ::= m \, r \) measure application
\( r ::= r == r \) equality comparison
\( \ldots \)

Figure 5.4: LiquidHaskell grammar

**Integer specifications** Consider a Haskell function taking \( n \) \( \text{Int} \) arguments, and returning an \( \text{Int} \). Each argument may be constrained by a separate predicate. For example, consider an example function \( \text{reqPos} \) that requires two positive numbers may be given the type:

\[
\text{reqPos} ::= \{ x: \text{Int} | x > 0 \} \to \{ y: \text{Int} | y > 0 \} \to \text{Int}
\]

Thus, \( \text{synth}_{\text{LIA}}^k \) creates \( n \) precondition templates, \( F_{U_1}^{f,\text{pre}}, \ldots, F_{U_n}^{f,\text{pre}} \), for a Haskell function with \( n \) \( \text{Int} \) arguments. The first precondition template, \( F_{U_1}^{f,\text{pre}} \) has only a single argument, for the leftmost \( \text{Int} \) argument. Each successive template has one more argument than the last, until \( F_{U_n}^{f,\text{pre}} \) accepts all the \( \text{Int} \)'s as arguments.

This encoding requires minor changes to the \( \text{synth}_{\text{LIA}}^k \) synthesizer to support having conjoined precondition templates for the same function in the original code. This is easily achieved, by simply having the \( \text{synth}_{\text{LIA}}^k \) synthesizer conjoin all such precondition templates together to form \( F_{U_1}^{f,\text{pre}}, \ldots, F_{U_n}^{f,\text{pre}} \):

\[
F_{U_1}^{f,\text{pre}}(x_1, \ldots, x_n) = F_{U_1}^{f,\text{pre}}(x_1) \land F_{U_2}^{f,\text{pre}}(x_1, x_2) \land \ldots \land F_{U_n}^{f,\text{pre}}(x_1, \ldots, x_n)
\]

**Challenges** Of course, real programs do not just use \( \text{Ints} \): algebraic datatypes are essential to writing Haskell. LiquidHaskell allows types that apply measures to algebraic datatype and specifying refinements on type arguments of polymorphic types. A usable synthesis tool for LiquidHaskell refinements must also account for these features. However, this presents a clear challenge: how can we translate requirements about algebraic data types into templates and constraints usable by \( \text{synth}_{\text{LIA}}^k \)? We describe how we account for both these challenges in the following.

**Measures** The refinements may use *measures*, a restricted (as fully described in [2]) class of functions. For our purposes, it is sufficient to note that measures take only a single argument. Often, measures map algebraic datatypes, such as lists, to simple types, such as \( \text{Int} \), which can be directly used in refinements. The \( \text{size} \) and \( \text{sumSize} \) functions from Section 5.2 are example of such measures. However, measures can also map algebraic datatypes to other algebraic datatypes. For instance, \( \text{fst} \) and \( \text{snd} \) are measures which extracts the first and second values in a tuple, respectively. Thus, given a tuple \( t ::= ([a], [b]) \), we can enforce that the two lists must have the same length via the refinement \( \text{size} (\text{fst} \ t) == \text{size} (\text{snd} \ t) \).
To convert problems with measures into problems over $\text{Ints}$ we exploit having a finite set of functions and constraints, with known function inputs and outputs. Consider searching via $\text{synth}_{\text{LIA}}^k$ for a set of specifications with size at most $k$. Then, we consider at most $k$ measure compositions. We find all compositions of at most $k$ measures that typecheck when applied to each argument in each specificaton, and which evaluate to an $\text{Int}$. Then, we precompute applying each measure composition to each applicable argument in each constraint, and replace the original argument with new arguments, one for each measure composition. Thus, we obtain functions that are simply over integers, and can be solved by the $\text{synth}_{\text{LIA}}^k$ synthesizer. When we find a solution, we translate it back to use measures, simply by reversing the substitution.

**Example 4.** Consider the $\text{concat}$ function, and the $\text{size}$ and $\text{sumSize}$ measures, as introduced in Section 5.2. $\text{concat}$’s argument has the type $[[a]]$. Suppose we have a constraint that $s_{\text{concat}}^{\text{pre}}([[1], [5, 6]])$ must be true. Both $\text{size}$ and $\text{sumSize}$ are applicable to the type $[[a]]$, so both $\text{size}([[1], [5, 6]])$ and $\text{sumSize}([[1], [5, 6]])$ will be evaluated, yielding 2 and 3 respectively. Then, in the formula for $\text{synth}_{\text{LIA}}^k$, this constraint will be translated to include the result of both evaluations as arguments: that is, the constraint will be $s_{\text{concat}}^{\text{pre}}(2, 3)$.

**Polymorphic refinements** Second, instantiations of polymorphic type arguments may be refined. For instance, $\{ \text{xs:[x:} \text{Int} \mid x > 0 \} \mid \text{size} \text{xs} > 0 \}$ is a type describing a non-empty list of positive Ints. As with measures, synthesizing these polymorphic refinements relies on us having only constraints over known input and output values. For each polymorphic type argument that is instantiated with an $\text{Int}$, we introduce a new function template $F_{f, \text{poly}}^k$ in the problem for the $\text{synth}_{\text{LIA}}^k$ synthesizer. Then, we extract each value that the polymorphic refinement would be applied to from the known inputs and outputs, and apply $F_{f, \text{poly}}^k$ to it in the $\text{synth}_{\text{LIA}}^k$ synthesizer.

**Example 5.** Consider the function $f :: \text{[Int]} -> \text{[Int]}$. Suppose we have a constraint that $s_{f}^{\text{pre}}([1, 4, 8])$ must be true. Assuming the $\text{size}$ measure is available, we will search for a LIA function satisfying the specification $s_{f}^{\text{pre}}(3)$. To search for a specification for the values in the list, we introduce an additional function $s_{f_2}^{\text{pre}}$, and require that $s_{f_2}^{\text{pre}}(1) \land s_{f_2}^{\text{pre}}(4) \land s_{f_2}^{\text{pre}}(8)$ holds.

### 5.6 Evaluation

To evaluate Lynx, we ran it on two sets of benchmarks.

Our first benchmark set is mostly focused on verifying properties of list functions. We refer to these benchmarks as the **list benchmarks**. We collected these benchmarks from a variety of sources, including a homework assignment in a graduate class where the students wrote code and modular specifications to verify properties. This
set of benchmarks requires synthesizing modular specifications needed to verify functions over integers, lists, and sets. In addition to synthesizing LIA specifications (as discussed in Section 5.5), we also implemented a simple synthesizer capable of finding specifications involving sets and set operations. This is discussed further below. Furthermore, we ran one larger LIA specification benchmark (kmeans1) as an interesting case study of applying Lynx to a larger, more complicated code base.

The second set of benchmarks is 46 programs based on the benchmarks from [70]. We refer to these as the **comparison benchmarks**. As these benchmarks were originally written in C, we translated the programs into Haskell so they could be run through Lynx. In Section 5.6.3 we explain the differences between verification in imperative and functional languages, and why a direct comparison between Lynx and existing tools targeting C is challenging. However, our translation allows a comparison, in broad strokes, to the existing tools Hola [70] and Horn-DT-CHC [79].

### 5.6.1 List Benchmarks

Our first benchmark set consists of 26 programs containing functions that manipulate lists. The largest benchmark, kmeans1, is an implementation of the inner loop of a KMeans algorithm. Section 5.6.2 provides a more detailed overview of this benchmark.

**Results** Figure 5.5 shows the performance of Lynx on the list benchmarks. The benchmarks were run with a timeout of 4 minutes, except kmeans1, which was given 18 minutes. Lynx successfully verified 23 of the 26 benchmarks.

**Optimizations and Heuristics** To measure the impact of the optimizations and heuristic in Section 5.4.3, we reran each benchmark without non-blocking constraint generation. We also reran each benchmark which took advantage of level descent with the level descent turned off.

The non-blocking constraints give a speed up of at least 5 seconds to 7 of the benchmarks. Four benchmarks (takeRelaxed, halves.take/replicate, and nearest) that terminate with non-blocking constraints timed out when this optimization was turned off. Most other benchmarks are not significantly affected by the non-blocking constraints. Interestingly, the largest decrease in run time from not using the extra constraints occurs for kmeans1, which went from taking 804.9 seconds with the extra constraints to 749.1 seconds without the extra constraints. The second largest decrease is for zipUnsafe, which dropped from taking 75.5 seconds to 71.4. Level descent was used only twice: without it insertSortElems takes over twice as long to terminate, and kmeans1 times out.

**Set specifications** In addition to the LIA specification synthesizer, we implemented a synthesizer for specifications about integer sets, using Z3’s extended theory of arrays [65]. Sets can be used in LiquidHaskell specifications either by creating them via a measure (often fromElts, which creates a Set from a list), or by applying a singleton function, Set_sng to an Int. Set membership can be checked, or sets can be compared
via equality or subset relations. Sets can be combined via union or intersection, using functions called `Set_cup` and `Set_cap`, respectively.

As a simple example of sets in LiquidHaskell, consider `add`, which adds a single element to a list:

```haskell
add x xs = x:xs
```

`add` can be assigned the postcondition refinement:

```haskell
{ r:[Int] | fromElts r == Set_cup (Set_sng x) (fromElts xs) }
```

to express that the output list `r` has the same elements as the input list `xs`, plus the element `x`.

Like our LIA synthesizer, our set synthesizer works via an SMT formula template that allows for various specifications, and uses MaxSMT to find the small solutions. Performance of the SMT solver is much more of a challenge synthesizing set specifications than it is synthesizing LIA specifications. As a representative example, on the `halves` benchmark, 107.6 seconds (72% of the total runtime) is spent running the synthesizer. In contrast, for the LIA benchmark `kmeans1`, 213.3 seconds, only 26% of the total runtime, is spent on synthesis. In the future, we hope to explore alternative synthesis techniques which might improve the runtime, such enumerative SyGuS solvers [24].

### 5.6.2 Case Study: kmeans1

The `kmeans1` benchmark implements the K-Means clustering algorithm [132]. We define `PointN N` as `N`-dimensional values containing type `Double`, and `Centers K N` as a collection of `K`-centers each of which is an `N`-dimensional point:

```haskell
type PointN N = {v:List Double | size v = n}
type Centers K N = M.Map {v:Int |0<=v && v<K} (PointN N)
```

`kmeans1` implements a single iteration of the clustering algorithm. `kmeans1`’s type says it takes a collection of `n`-dimensional points, and a starting `k`-centering and returns an updated `k`-centering:

```haskell
kmeans1 :: k:Nat -> n:Nat -> List (PointN n) -> Centers k n
           -> Centers k n
kmeans1 k n ps cs = normalize (mapReduce fm fr ps)
where normalize = M.map ((sz, p) -> centroid n p sz)
    fm p = singleton (nearest k n cs p, (1::Int, p))
    fr wp1 wp2 = mergeCluster n wp1 wp2
```

When verifying `kmeans1`, 214 seconds, (about a fourth of the total time), is spent symbolically executing `kmeans1`. This is attributable to `kmeans1`’s complexity. Verifying `kmeans1` requires synthesizing specifications for three of its direct callees, each with multiple preconditions. For example, the specification synthesized for `nearest`, which computes the center closest to a point `p`, is:

```haskell
nearest :: k:Int -> n:Int
Despite these specifications complexity, our inference approach scales to synthesize them. Previously a user verifying kmeans1 would have had to tediously determine the exact specifications to give to each of the functions, whereas with Lynx, the specifications can be automatically synthesized.

### 5.6.3 Comparison benchmarks

To compare the techniques implemented in Lynx to existing techniques, we took the 46 C benchmarks originally written or collected to evaluate Hola [70] and manually rewrote them in Haskell. We will explain why these benchmarks may not be the best fit for Haskell. Nevertheless, they allow us to compare and position Lynx relative to Hola [70], and Horn-DT-CHC [79].

**Translation, Limitations, and Compensations** The [70] benchmarks are not a natural fit for Haskell. The benchmarks are all single non-recursive functions containing at least one while or for loop. In contrast, Haskell has no loops at all: instead programs must be written with recursive functions. In Haskell a while loop can be simulated via a higher order *while* function:

\[
\text{while} :: (a \to \text{Bool}) \to (a \to a) \to a \to a
\]

\[
\text{while~cond~body~x~}=~\text{if~cond~x}
\]

\[
\text{then~while~cond~body~(body~x)}
\]

\[
\text{else~x}
\]

*while* is passed a value x of type a and two functions: *cond*, which has type a -> Bool and corresponds to the loop conditional, and *body*, which has type a -> a and corresponds to the body of the loop. The *while* function then calls *body* on x until *cond* x is false. To use *while* to translate a C loop into Haskell, we pass all variables modified in the C program in a tuple as x, and write appropriate functions for *cond* and *body*. Figure 5.6 shows an example of one such translation.

Due to details of verification condition generation for imperative versus functional code, this translation often makes verification harder for a functional tool than an imperative tool. These differences arise from the fact that, in an imperative program, loops are a language construct, with special handling during verification condition generation. In a imperative program, an loop invariant I must be provided for each loop. This loop invariant must be true at the initial call to the loop, and must be maintained by each iteration of the loop. Thus, the synthesizer knows it must synthesize a single invariant.

In contrast, Lynx, working in a functional language, might try and synthesize different specifications for the pre and postcondition of a *body* function. To control for this, and try to focus our analysis on the difference in effectiveness of underlying techniques (as opposed to the verification challenges posed by imperative versus functional code) we passed a flag to Lynx when running the comparison benchmarks,
which indicated that **body** functions should be assigned the same pre and postcondition.

Unfortunately, this is not always a viable method to address the additional challenge faced in a functional context. After a loop has terminated, an imperative verifier knows not only that the loop invariant holds, but also that the negation of the loop condition holds. But in our functional context, it is impossible (at least, given the subset of LiquidHaskell specifications we consider) to communicate that after **while** returns a value, **cond** evaluates to false. Thus, when translating benchmarks where this is an important property, we create special versions of **while** with an appropriate **cond** and **body** inlined, and allow **while** to be assigned different pre and post conditions. While this increases the search space (relative to the search space in an imperative context), it does at least allow for the existence of a solution.

**Results** In total, Lynx succeeds in verifying 30 benchmarks (with a timeout of 10 minutes.) Figure 5.7 shows the running times for Lynx to verify the benchmarks.

Horn-DT-CHC [79] is the most similar existing work to Lynx, as it synthesizes invariants and specifications sufficient for verification using a CEGIS loop and counterexamples. Horn-DT-CHC succeeds on 29 out of 45 benchmarks, with a timeout of 10 minutes (one benchmark is discarded, because a conversion to their input format, CHCs, made it trivial.) The benchmarks solved by Horn-DT-CHC and Lynx are not a perfect overlap. There are 7 benchmarks — in particular, 7, 9, 11, 13, 15, 33, and 38 — which we are able to verify and which Horn-DT-CHC cannot.

The tool in the original source of the benchmarks, Hola [70] works by using logical abduction to find possible loop invariants. Hola verifies 43 of the 46 benchmarks (with a 200 second timeout.) However, Lynx verifies benchmark 15, which Hola fails to verify. Hola also timeouts when run on 19 and 34, which cause Lynx to timeout as well. Hola performs better overall on these benchmarks than both Lynx and [79]. However, as will be elaborated in Section 5.7, Hola’s approach is restricted to logics, such as LIA, in which quantifier elimination is possible.
<table>
<thead>
<tr>
<th>Main Function(s)</th>
<th>LoC</th>
<th>Funcs</th>
<th>Lvl</th>
<th>Time (s)</th>
<th>Loops</th>
<th>Lvl Dec.</th>
<th>Negated Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zipUnsafe</td>
<td>14</td>
<td>3</td>
<td>2</td>
<td>75.5</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>takeRelaxed</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>51.8</td>
<td>36</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>take</td>
<td>12</td>
<td>4</td>
<td>3</td>
<td>5.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>drop</td>
<td>12</td>
<td>4</td>
<td>3</td>
<td>13.7</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>weightedAverage</td>
<td>30</td>
<td>12</td>
<td>5</td>
<td>54.0</td>
<td>23</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>risers</td>
<td>15</td>
<td>4</td>
<td>4</td>
<td>13.1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>zipUnlessEmpty</td>
<td>13</td>
<td>6</td>
<td>4</td>
<td>TO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>quickSortSize</td>
<td>19</td>
<td>3</td>
<td>2</td>
<td>14.4</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>reverse</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>14.6</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>concat (1)</td>
<td>26</td>
<td>12</td>
<td>3</td>
<td>TO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>concat (2)</td>
<td>29</td>
<td>15</td>
<td>3</td>
<td>27.8</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>concat (3)</td>
<td>22</td>
<td>5</td>
<td>3</td>
<td>15.8</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>zipWith</td>
<td>24</td>
<td>10</td>
<td>3</td>
<td>19.2</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>size</td>
<td>22</td>
<td>13</td>
<td>5</td>
<td>15.9</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>map</td>
<td>17</td>
<td>9</td>
<td>3</td>
<td>11.8</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>replicate</td>
<td>22</td>
<td>10</td>
<td>3</td>
<td>13.9</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>drop/dropEnd</td>
<td>17</td>
<td>5</td>
<td>3</td>
<td>27.5</td>
<td>13</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>take/replicate</td>
<td>17</td>
<td>6</td>
<td>3</td>
<td>21.1</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nearest</td>
<td>33</td>
<td>15</td>
<td>5</td>
<td>61.4</td>
<td>18</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>kmeans1</td>
<td>99</td>
<td>34</td>
<td>5</td>
<td>804.9</td>
<td>79</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sets</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>halves</td>
<td>12</td>
<td>3</td>
<td>2</td>
<td>147.9</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>append</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>7.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>elem</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>12.5</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>insertSortElems</td>
<td>10</td>
<td>2</td>
<td>2</td>
<td>23.9</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>mergeSortElems</td>
<td>22</td>
<td>4</td>
<td>2</td>
<td>TO</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>reverse</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>17.4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 5.5:** Evaluation of Lynx: **Main Function** is the benchmark name; **LoC** and **Funcs** are the number of lines of code and functions in the benchmark; **Lvl** is the maximum call-graph depth in the benchmark; **Time** is the number of seconds needed to infer modular specifications that verify the benchmark; **Loops** is the number of CEGIS loop iterations needed to infer specifications; **Lvl Dec** and **Negated Models** is the number of times the backtracking and negated models heuristics were used.
while(x!=0) {
    x--;
    y++;
}

(b) A C function

cond1 (x, y) = x /= 0
body1 (x, y) = (x - 1, y + 1)
use (x, y) = while cond1 body1 (x, y)

Figure 5.6: Translation of a C loop (in 5.6a) to a Haskell recursive function (in 5.6b.)

<table>
<thead>
<tr>
<th>#</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>9.4</td>
</tr>
<tr>
<td>02</td>
<td>TO</td>
</tr>
<tr>
<td>03</td>
<td>23.4</td>
</tr>
<tr>
<td>04</td>
<td>TO</td>
</tr>
<tr>
<td>05</td>
<td>74.2</td>
</tr>
<tr>
<td>06</td>
<td>132.3</td>
</tr>
<tr>
<td>07</td>
<td>139.8</td>
</tr>
<tr>
<td>08</td>
<td>28.0</td>
</tr>
<tr>
<td>09</td>
<td>48.4</td>
</tr>
<tr>
<td>10</td>
<td>23.8</td>
</tr>
</tbody>
</table>

Figure 5.7: Running times of Lynx on the benchmarks from [70] with a timeout of 10 minutes.
5.7 Related Work

**Modular Verification Specification Generators** Houdini [82] automatically generates specifications for the ESC/Java modular verifier [81]. Houdini is a preprocessor, which discovers specifications by generating a large number of candidate templates, and then using ESC/Java to find a fix-point of valid specifications. LiquidHaskell uses a similar scheme to infer specifications using user-supplied templates (qualifiers). Unfortunately, it can be slow to find a valid fix-point, so, the number of templates must not be too large. Consequentually, the set of templates might be missing a specification that is required to verify user-written code. In contrast, our work discovers and proves specifications as needed by the verifier, allowing it to search for specifications that are useful to verify specific user-written specifications.

**Verification Techniques** Inlining techniques [126, 125, 184, 98] are effective at proving properties involving no loops, or checking properties up to a loop-unrolling bound. In general, though, inlining is ineffective to verify functions with input-dependent recursion, since the number of recursive calls might be unbounded.

[60, 59] describe precondition inference techniques based on shifting assertions backwards, closer to the entry point of the program, and using widening to overapproximate loops. A distinction is drawn between preconditions that are *sufficient* and preconditions which are *necessary*. A precondition is *sufficient* if it can be used to prove that no assertion will be violated, while a precondition is *necessary* if it must hold to avoid violating an assertion. [60, 59] focus on finding necessary preconditions to avoid violating assertions, but makes no guarantees that the preconditions are sufficient to avoid assertion violations. In contrast, our work finds (in their language) necessary and sufficient specifications to verify a user provided specification.

Work such as [22, 165] synthesizes *maximal specifications* for undefined functions. Such maximal specifications describe the largest set of allowable behaviors in order to guarantee satisfying some specification for a known function. In contrast, our work assumes all function procedures are known, and aims to find specifications for each level of the call graph to fully verify user provided specifications. Techniques developed for maximal specification synthesis could possibly complement our approach to synthesizing specifications for a particular call graph level.

**(Counter)example Guided Inference** Like our work, [176] synthesizes refinement types for a higher order modular verifier, automating the process of modular verification. Traces through the program (which can be seen, in our terminology, as a sort of abstract counterexample, in which every called function is abstracted) are used to guide the synthesizer. Specifications are synthesized all at once, rather than modularly, as in our approach. [176] proves a progress property for their algorithm, which is sufficient to show that it is complete for finite sets of possible specifications (along the lines of our Theorem 3) however, it does not consider completeness in the case of infinite possible specifications (as we do with Theorem 5.)

[79] finds specifications for modular verification using Horn-ICE and decision tree
learning. Whereas in Lynx, specification synthesis is itself largely modular (only specifications for functions on the same call graph level must be synthesized together), in [79] all specifications must be synthesized together. [79] and Lynx take different tradeoffs when actually synthesizing specifications. [79]'s decision tree learning algorithm finds specifications over an arbitrary set of predicates, whereas Lynx uses synthesizers built to support specific theories. While [79] is more easily generalizable, Lynx is able to benefit from SMT solvers domain specific reasoning and heuristics in finding specifications. Our evaluation compares [79] and Lynx, and finds that both are able to verify programs that the other can not.

PIE [151] generates preconditions and loop invariants. Counterexamples are gathered when a specification is insufficient. Example-guided synthesis is used to learn predicates dividing positive and negative examples. NumInv [144] and SymInfer [145] search for loop invariants using symbolic execution (Klee [46] and Symbolic Pathfinder [154], respectively) to generate, in our parlance, concrete counterexamples. PreInfer [34] collects path constraints from symbolic execution to generate preconditions that prevent tests from failing. Provisio [33] uses machine learning techniques and symbolic execution to learn preconditions that prevent failing tests from a test generator. Alive-Infer [137] generates preconditions to ensure the correctness of LLVM optimizations. LinearArbitrary [208] uses machine learning techniques to solve for predicates in Constrained Horn Clauses, which can be used to verify safety properties of programs. The techniques used are limited to numerical constraints. Hola [70] strengthens loop invariants until they become inductive, by solving abduction problems over verification conditions. Quantifier elimination is used to find solutions to abduction problems, thus applying Hola to specifications logics that do not admit quantifier elimination (such as the theory of arrays) would require developing new approaches to solving abduction problems.

Our work differs in its focus on modular verification. In a modular setting, we contend with many specifications, for many separate functions. Whereas these existing tools synthesize invariants or preconditions for a single place in the code, we face the problem of localizing which of several specifications need to be strengthened or weakened. Lynx uses abstract counterexamples to determine which specifications needs to be strengthened, and how to do so. With the exception of [79], none of the existing work has a completeness result that applies to an infinite set of possible specifications.

Finally, these existing techniques generally focus on problems where all inputs are directly integers. In contrast, our synthesizer synthLH also searches over measures (including compositions of measures) and integer values nested in polymorphic types. As this is done by a decomposition to a problem that is purely over integers, it is largely orthogonal. If desired, we expect the decomposition could be used alongside any of these other existing specification inference techniques.

**Model Checking** Lazy abstraction [102] is a model checking technique that refines pieces of an abstract model using predicates learned by interpolating counterexample
traces. Whale [23] uses craig interpolants to refine abstract reachability graphs, and prove properties of recursive programs. Mochi [120, 177] applies counterexample-guided predicate abstraction to verify numerical properties of higher-order functional programs. All the above use interpolation on execution traces, as opposed to just input-output values as in Lynx. The use of input-output counterexample values crucially lets Lynx reliably synthesize specifications over functions like \texttt{size}, in contrast to the purely numerical properties to which trace-based interpolation is limited [108].

5.8 Conclusions and Future Work

Modular verification tools suffer from the drawback that they cannot verify a program unless all callee functions have explicitly stated strong enough specification. In this work, we developed formal foundations and an accompanying implementation for automatically inferring specifications for callee functions. To automate modular verification, we described a counterexample guided inference algorithm. We proved that our algorithm is sound and complete for finite sets of specifications. Furthermore, we introduced \textit{size-bound synthesizers} to extend our soundness and completeness result to infinite sets of specifications. We applied those general results and showed how to build a size-bound synthesizer for linear integer arithmetic specifications. Finally, our evaluation shows our techniques applicability to a range of benchmarks involving integers, lists, and sets.

As future work, we aim to extend our techniques to synthesize more types of specifications. We already have some preliminary work supporting set specification. Many otherwise appealing synthesizers (such as SyGuS solvers) are currently incapable of producing interpolants. In fact, often, synthesis techniques cannot determine unrealizability at all. Thus, one intriguing direction is to extend existing techniques to detect unrealizability and generate interpolants.

We believe that our results indicate that the automation of modular verification can be significantly improved for certain classes of programs. By combining and applying synthesis and automated reasoning techniques, we reduce the burden of writing annotations for modular verification, thus making the verification process more accessible for everyday programmers.
Chapter 6

Grammar Filtering For Syntax-Guided Synthesis

This chapter describes work completed in collaboration with Kairo Morton, Elven Shum, Ruzica Piskac, and Mark Santolucito. This work includes material originally published in [139].

6.1 Introduction

The term “program synthesis” refers to automatically generating code to satisfy some specification. That specification describes what the code should do, without going into details about how it should be done. The specification could be given as a set of constraints [134, 123], it can be deduced from the program and its environment [92, 80], or it can be inferred from a large corpus [37, 174].

One paradigm of program synthesis is called *programming by example* [61] (PBE). In the PBE approach, a user only provides a set of pairs of input-output examples that illustrate the desired behavior of the code. From these examples, the PBE engine should then generate code that generalizes from the examples to create a program which covers the unspecified examples as well.

The idea of automated code synthesis is an area of research with a long history (cf. the Church synthesis problem [56]). However, due to the problem’s undecidability and high computational complexity for decidable fragments, for almost 50 years the research in program synthesis was mainly focused on addressing theoretical questions and the size of synthesized programs was relatively small. However, the state of affairs has drastically changed in the last decade. By leveraging advances in automated reasoning and formal methods, there has been a renewed interest in software synthesis. The research in program synthesis has recently focused on developing efficient algorithms and tools, and synthesis has even been used in industrial software [90]. Today, machine learning plays a vital role in modern software synthesis and there are numerous tools and startups that rely on machine learning and big data to automatically generate code [16, 37].
With numerous synthesis tools and formats being developed, it was difficult to empirically evaluate and compare existing synthesis tools. The Syntax Guided Synthesis (SyGuS) format language [24, 167] was introduced in an effort to standardize the specification format of program synthesis, including PBE synthesis problems. The SyGuS language specifies synthesis problems through two components - a set of constraints (e.g., input-output examples), and a grammar (a set of functions). The goal of a SyGuS synthesis problem is to construct a program from functions within the given grammar that satisfies the given constraints. With this standardized synthesis format and an ever-expanding set of benchmarks, there is now a yearly competition of synthesis tools [25], which pushes the frontier of scalable synthesis further.

The SyGuS Competition splits synthesis problems into tracks, for example PBE Strings or PBE BitVectors, assigning a different grammar for each track - and sometimes even varying the grammar within a single track. As the grammar defines the search space in SyGuS, this allows benchmark designers to ensure problems are relatively in-scope of current tools. However, when synthesis is deployed in real-world applications, we must allow for larger grammars that account for the wide range of use-cases users require [173]. While larger grammars allow for more expressive power in the synthesis engine, it also slows down the whole synthesis process.

In our own experimentation, we found that by manually removing some parts of the grammar from the SyGuS Competition benchmarks, we can significantly improve synthesis times. Accordingly, we sought to automate this process. Removing parts of a grammar is potentially dangerous though, as we may remove the possibility of finding a solution altogether. In fact, understanding the grammar’s impact on synthesis algorithms is a complex problem, connected to the concept of overfitting [150].

In this paper, we utilize machine learning to automate an analysis of a SyGuS grammar and a set of synthesis constraints. We generate a large number of SyGuS problems, and use this data to train a neural network. Given a new SyGuS problem, the neural network predicts how likely it is for a given grammar element to be critical to synthesizing a solution to that problem. Our key insight is that, in addition to criticality, we predict how much time we expect to save by removing this grammar element. We combine these predictions to efficiently filter grammars to fit a specific synthesis problem, in order to speed up synthesis times. Even with these reduced grammars, we are still able to find solutions to the problems.

We implemented our approach in a modular tool, GRT, that can be attached to any existing SyGuS synthesis engine as a blackbox. We evaluated GRT by running it on the SyGuS Competition Benchmarks from 2019 in the PBE Strings track. We found GRT outperformed CVC4, the winner of the SyGuS Competition from 2019, reducing the overall synthesis time by 47.65%. Additionally, GRT was able to solve a benchmark for which CVC4 timed out.

In summary, the core contributions of our work are as follows:

1. A methodology to generate models that can reduce time needed to synthesize PBE SyGuS problems. In particular, our technique reduced the grammar by
identifying which functions to try to eliminate to increase the efficiency of a SyGuS solver. It also learns a model to predict which functions are critical for a particular PBE problem.

2. A demonstration of the effectiveness of our methodology. We show experiments on existing SyGuS PBE Strings track that demonstrates the speed up resulting from using our filtering as a preprocessor for an existing SyGuS solver. Over the set of benchmarks, our techniques decreases the total time taken by synthesis by 47.65%.

6.2 Background

A SyGuS synthesis problem is a tuple \((C, G)\) of constraints, \(C\), and a context-free grammar, \(G\). In our case we restrict the set of constraints to the domain of PBE, so that all constraints are in the form of pairs \((i, o)\) of input-output examples. We write \(G \setminus g\) to denote the grammar \(G\), but without the terminal symbol \(g\). The set of terminal symbols are the component functions that can be used in constructing a program (e.g. +, -, str.length). We also use the notation, \(\pi(G)\), to denote the projection of \(G\) into its set representation, which is the set of the terminal symbols in the grammar.

The problem statement of syntax-guided synthesis (SyGuS) is; given a grammar, \(G\), and a set of constraints \(C\), find a program, \(P \in G\), such that the program satisfies all the constraints – \(\forall c \in C.P \vdash c\). For brevity, we equivalently write \(P \vdash C\). If our synthesis engine is able to find such a program in \(t\) seconds or less, we write that \((G, C) \leadsto_t P\). We use the notation \(T^C_G\) to indicate the time to run \((G, C) \leadsto_t P\). If the SyGuS solver is not able to find a solution within the timeout \((T^C_G > t)\), we denote this as \((G, C) \not\leadsto_t P\). We typically set a timeout on all synthesis problems of 3600 seconds, the same value of the timeout used in the SyGuS competition. We write \((G, C) \leadsto P\) and \((G, C) \not\leadsto P\) as shorthand for \((G, C) \leadsto_{3600} P\) and \((G, C) \not\leadsto_{3600} P\), respectively.

We define \(G\) as the grammar constructed from the maximal set of terminal symbols we consider for synthesis. We call a terminal, \(g\), within a grammar, critical for a set of constraints, \(C\), if \((G \setminus g, C) \not\leadsto P\). For any given set of constraints, if a solution exists with \(G\), there is also a grammar, \(G_{crit}\), that contains exactly the critical terminal symbols required to find a solution. More formally, \(G_{crit}\) is constructed such that

\[(G_{crit}, C) \leadsto P \land \forall g \in G_{crit}. \ (G \setminus g, C) \not\leadsto P\]

Note that \(G_{crit}\) is not unique.

The goal of our work is to find a grammar, \(G^*\), where \(\pi(G_{crit}) \subseteq \pi(G^*) \subseteq \pi(G)\). This will yield a grammar that removes some noncritical terminal symbols so that the search space is smaller, but still sufficient to construct a correct program.
Figure 6.1: GRT uses the grammar $G$ and constraints $C$ to predict how critical each function is, and the amount of time that would be saved by eliminating it from the grammar. Then, it outputs a new grammar $G^*$, which it expects will speed up synthesis over the original grammar (that is, it expects that $T^*_G < T^C_G$).

6.3 Overview

Our system, GRT, works as a preprocessing step for a SyGuS solver. The goal of GRT is to remove elements from the grammar and thus, by having a smaller search space, save time during synthesis. To do this we combine two metrics, as shown in Figure 6.1: our predicted confidence that a grammar element is not needed, and our prediction of how much time will be saved by removing that element. We focus on removing only elements where we are both confident that the grammar element is noncritical, and that removing the grammar element significantly impacts synthesis times. By giving the constraints and the grammar definition to GRT, we predict which elements of the grammar can be safely removed. By analyzing running times we predict which of these elements are beneficial to remove. We describe GRT in three sections, addressing dataset generation, the training stage, and our evaluation.

6.4 Data Generation

In order to learn a model for GRT, we need to generate a labelled dataset that maps constraints to grammar components in $G_{crit}$. This will allow us to predict, given a new set of constraints $C''$, which grammar elements are noncritical for synthesis, and accordingly prune our grammar. The generation of data for application to machine learning for program synthesis is a nontrivial problem, requiring careful construction of the dataset [181]. We break the generation of this dataset into two stages: first, we generate a set of programs, $P$ from $G$. Then, for each program in $P$, we generate constraints for that program. We additionally need a dataset of synthesis times, in order to predict how long synthesis takes for a given set of constraints.
6.4.1 Criticality Data

To generate a set of programs \( P \), that can be generated from a grammar \( G \), we construct a synthesis query with no constraints. We then run CVC4 with the command `-sygus-stream`, which instructs CVC4 to output as many solutions as it can find. With no constraints, all functions satisfy the specification, and CVC4 will generate all permutations of (well-formed and well-typed) functions in the grammar, until the process is terminated (we terminate after generating \( n \) programs). Because CVC4 generates solutions of increasing size, we collect all generated programs, then shuffle the order to prevent data bias with respect to the order (size) in which CVC4 generated programs.

After generating programs, we generate corresponding constraints (in the form of input-output examples for PBE) for these functions. To do this, for each program, \( P \), we randomly generate a set of inputs \( I \), and compute the input-output pairs \( C = \{(i,P(i)) \mid i \in I\} \). We then form a SyGuS problem \((G,C)\), where we know that the program \( P \) satisfies the constraints, and is part of the grammar: \( P \vdash C \) and \( P \in G \). This amounts to programs that could be synthesized from the constraints (i.e. \( (G,C) \leadsto_{\infty} P \)). It is important that our dataset represent programs that could be synthesized, as opposed to what can be synthesized (i.e. \((G,C) \leadsto_{3600} P\)). This is important because we will use this data set to try to learn the “semantics” of constraints, and we do not want to use this data set to additionally, inadvertently learn the limitations of the synthesis engine.

At this point, we have now constructed a dataset of triples of grammars (fixed for all benchmarks), constraints, and programs, \( D = \{(G,C_1,P_1),\ldots,(G,C_n,P_n)\} \). In order to use \( D \) to helps us predict \( G_{\text{crit}} \), we break up each triple by splitting each constraint set \( C \) into its individual constraints. For a triple \((G,C,P)\), where \( C = \{c_1\ldots c_m\} \), we generate a new set of triples \( \{(G,c_1,P),\ldots,(G,c_m,P)\} \). The union of all these triples of individual constraints form our training set, \( \mathcal{T}R_{\text{crit}} \), that will be used to predict critical functions in the grammar for a given set of constraints.

6.4.2 Timing Data

In addition to a training set for predicting \( G_{\text{crit}} \), we also need a separate training set for predicting the time that can be saved by removing a terminal from the grammar. This dataset maps grammar elements \( g \in G \) to the effect on synthesis times, \( \mathbb{R} \), when \( g \) is dropped from the grammar. To do this we require synthesis problems that more closely model the types of constraints that humans typically write. We collect these set of benchmarks from users of the live coding interface for SyGuS [173]. Because we had limited number of human-generated constraint examples, we augmented this with constraints generated from \( \mathcal{T}R_{\text{crit}} \).

We run synthesis for each problem with the full grammar, as well as with all grammars constructed by removing one element, \( g \). For every synthesis problem benchmark, \( 1 \leq i \leq m \), we record the difference in synthesis times between running
with the full grammar, and removing $g$:

$$T_{G}^C - T_{G\backslash g}^C$$

Thus, we create a training set, $TR_{time}$, relating each terminal $g \in \pi(G)$ and a set of constraints, to the time it takes to synthesize a solution without that terminal.

## 6.5 Training

### 6.5.1 Predicting criticality

Our goal is to predict, given a set of constraints $C$, if a terminal $g$ belongs to the set of terminals $\pi(G_{crit})$ for $C$. To do this, we use a Feedforward Neural Network (Multi-Layer Perceptron), with an extra embedding layer to encode the string valued input-output examples into feature vectors. We train the neural network to predict the membership of each terminal $g \in \pi(G)$ to the critical set $\pi(G_{crit})$, based on a single constraint $c \in C$. This prediction produces a 1D binary vector of length $|\pi(G)|$, where 1 at position $i$ in the binary vector indicates the terminal in position $i$ is predicted to belong to the critical set.

When a SyGuS problem has multiple ($|C| \geq 2$) constraints, we run our prediction on each constraint individually. We then use a voting mechanism to come to consensus on the construction of $G^*$. After computing $|C|$ binary vectors across all constraints, the vectors are summed to produce a final voting vector. The magnitude of each element in this final voting vector represents the number of votes “from each constraint” that the terminal represented by that element is in the critical set. We then use this final voting vector in combination with our time predictions.

### 6.5.2 Predicting time savings

It is only worthwhile to remove a terminal symbol $g$ from a grammar $G$ if $T_{G\backslash g}^C$ is less than $T_{G}^C$. If a $g$ stands to only give us a small gain in synthesis times, it may not be worth the risk that we incorrectly predicted its criticality.

To predict the amount of time saved by removing a terminal $g$ we examine the distribution of times in our training set $TR_{time}$. For each terminal $g$, we calculate $A_g$, the average time increase that results from removing $g$ from the grammar. Denoting the time to run $(G, C) \leadsto P$ as $T_G^C$, we can write $A_g$ as:

$$A_g = \frac{\sum_{i=1}^{n} T_{G}^C - T_{G\backslash g}^C}{n}$$

If a terminal $g$ has a negative $A_g$, then removing it from the grammar actually slows down synthesis, on average. As such, dropping the terminal from the grammar is not generally helpful. Thus, we only consider those terminals with a positive $A_g$ in our second step.
6.5.3 Combining predictions

With our predictions of the criticality of a terminal $g$ and of time saved by removing $g$, we must make a final decision on whether or not we should remove $g$. To do this, we take the top three terminals with the greatest average positive impact on synthesis time over the training set, as computed with $A_g$. These tended to be terminals that mapped between types which saved more time due to the internal mechanisms and heuristics of the CVC4 solver. We then use the final voting vector from our criticality prediction to choose only two out of the three to remove from $G$ to form $G^\star$. We chose to remove only two terminals from $G$ in order to minimize the likelihood of generating a $G^\star$, such that $\pi(G^\star) \subseteq \pi(G_{crit})$. We conjecture that the number of terminals removed is a grammar-dependent parameter that must be selected on a per grammar basis, just as the number of terminals with $A_g > 0$ is grammar specific.

6.5.4 Falling back to the full grammar

There is some danger that $G^\star$ will, in fact, not be sufficient to synthesize a program. Thus, we propose a strategy that

- first, tries to synthesize a program with the grammar $G^\star$
- second, if synthesis with $G^\star$ is unsuccessful, falls back to attempting synthesis with the full grammar $G$.

We determine how long to wait before switching from $G^\star$ to $G$ by finding an $x$ that minimizes:

$$\sum_{i=1}^{n} \left\{ \min(x + T_{G^\star}^{C_i}, t) \begin{cases} T_{G^\star}^{C_i} < x \\ T_{G^\star}^{C_i} > x \end{cases} \right\}$$

where $C_1\ldots C_n$ are the constraints from the training set, and $t$ is the timeout for synthesis.

Ideally, as captured in the first line of the sum, $(C_i, G^\star) \leadsto_x P$ will finish before $T_{G^\star}^{C_i} = x$. However, if a benchmark does not finish in that time, it will fall back on the full grammar. Then, either $(C_i, G^\star) \leadsto_{t-x} P$ will succeed, and synthesize the expression in total time $x + T_{G^\star}^{C_i}$, or synthesis will timeout, in total time $(t-x)+x = t$.

6.6 Experiments

The SyGuS competition [26] provides public competition benchmarks and results from previous years. In particular, the PBE Strings dataset provides a collection of PBE problems over a grammar that includes string, integer, and Boolean manipulating functions. First, we describe our approach to generating a training set of PBE problems over strings. Then, we present our results running GRT against the 2019 competition’s winner in the PBE Strings track, CVC4 [147, 39, 25]. We are able to
reduce synthesis time by 47.65% and synthesize a new solution to a benchmark that was left unsolved by CVC4.

6.6.1 Technical details

The data triples generated during our initial data generation process of $\mathcal{T}R_{\text{crit}}$ are triples of strings. However, the neural network cannot process input-output pairs of type string as input. Thus, this data must be encoded numerically before it can be utilized to train the neural network. Each character in the input-output pairs is converted to its ASCII equivalent integer value. The size of each pair is then standardized by adding a padding of zeros to the end of each newly encoded input and output vector respectively. This creates two vectors: the encoded input and the encoded output, both of which have a length of 20. These two vectors are then concatenated to give us a single vector for training. By the end of this process the triples created in our first data generation step are now one vector of type $\mathbb{N}^{40}$ representing the input-output pair and a correct label $P$ that will be predicted.

To generate the training set for predicting synthesis times, $\mathcal{T}R_{\text{time}}$, we combine human generated and automatically generated SyGuS problems. Specifically, we use 10 human generated SyGuS problems, and 20 randomly selected problems from $\mathcal{T}R_{\text{crit}}$.

The overall architecture of our model can be categorized as a multi-layer perceptron (MLP) neural network. More specifically, our model is made up of five fully connected layers: the input layer, three hidden layers, and the output layer. By using the Keras Framework, we include an embedding layer along with our input layer which enables us to create unique vector embeddings of length 100 for any given input-output pair in the dataset. This embedding layer learns the optimal weights...
used to create these unique vectors through the training process. Thus, we create an encoding of the input-output pairs for training, while simultaneously standardizing the scale of the vector before it reaches the first hidden layer. The hidden layers of the model are all fully connected, and all use the sigmoid activation function. In addition, we implement dropout during training to ensure that overfitting does not occur. The size of the hidden layers was calculated using a geometric series to ensure that there was a consistent decrease in layer size as the layers get closer to the output layer. Specifically, the size of each hidden layer was calculated by:

\[ HL_{size}(n) = \text{input}_{size}(\frac{\text{output}_{size}}{\text{input}_{size}})^{n/\text{num}_L+1} \]

where \( \text{num}_L \) represents the total number of layers in the network. Our model used the Adam optimization method and the binary-cross entropy loss function as it is well suited for multi-label classification. Overall, our model was trained on 124928 data points for 15 epochs with a batch size of 200 producing a training time of 228 seconds.

### 6.6.2 Results

After generating our data sets and training our model, we wrote a wrapper script to run GRT as a preprocessor for CVC4’s SyGuS engine. We compared the synthesis results of GRT+CVC4 with the synthesis results of running CVC4 alone. All experiments were run on MacBook Pro with a 2.9 GhZ Intel i5 processor with 8GB of RAM. CVC4 uses a default random seed, and is deterministic over the choice of that seed, so the results of synthesis from CVC4 on a given grammar and set of constraints are deterministic. We note that our training data in no way used any of the SyGuS benchmarks.

GRT+CVC4 outperformed directly calling CVC4 on 32 out of 64 benchmarks (50%), with a reduction in total synthesis time over all benchmarks from 1304.87 seconds with CVC4 to 683.09 seconds with GRT+CVC4. On one benchmark, CVC4 timed out and was not able to find a solution (even when the timeout was increased to 5000 seconds), while GRT+CVC4 found a solution within the timeout specified by the SyGuS Competition rules (3600 seconds). On one benchmark, both CVC4 and GRT+CVC4 timeout (TO) and are not able to find a solution. On the other 31 benchmarks, CVC4 performed the same (within \( \pm 0.1 \) s) with and without the pre-processor. All the benchmarks for which CVC4 performed the same as GRT+CVC4 finish in under 2 seconds, and 28 of the 31 finish in under a second. In these cases there was little room for improvement even with GRT+CVC4.

Figure 6.4 shows the exact running times with both the full and reduced grammars from the benchmarks with the 30 largest running times with the full grammar. These are the benchmarks for which the synthesis times and size of the solution diverge most meaningfully, however all other data is available in the supplementary material for this paper. Figure 6.4 also shows \( |P| \) and \( |P^*| \), the sizes of the programs found by
the CVC4 and GRT+CVC4, respectively. We define size of a program as the number of nodes in the abstract syntax tree of the program. In terms of the grammar $G$, this is the number of terminals (including duplicates) that were composed to create the program.

In Figure 6.2, we present a visual comparison of the results for the 20 functions that took CVC4 the longest, while still finishing in the 3,600 second time limit. We note that we have the largest gains on the problems for which CVC4 is the slowest. Problems that CVC4 already handles quickly stand to benefit less from our approach.

In order to get a better baseline to understand the impact of GRT on running times, we ran a version of GRT with only the criticality prediction, which we call GRTC. In this case, GRTC+CVC4 actually performed worse than CVC4 by itself, increasing the running time on 53 out of the 62 benchmarks that did not timeout on CVC4.

On all but 5 benchmarks, CVC4 synthesized the same program when running with $G$ and $G^*$. The sizes of the programs (in terms of the number of terminal symbols used) for the benchmarks on which CVC4 synthesized different programs are shown in Figure 6.3. While on some benchmarks GRT+CVC4 produced a larger solution than CVC4, as a whole the sum of the size of all solutions for CVC4 was 806, while for GRT+CVC4 it was 789. Thus, overall, we were able to outperform CVC4 on size of synthesis as well.

The SyGuS competition scores each tool using the formula: $5N + 3F + S$, where $N$ is the number of benchmarks solved (non-timeouts), $F$ is based on a “pseudo-logarithmic scale” [26] indicating speed of synthesis, and $S$ is based on a “pseudo-logarithmic scale” indicating size of the synthesized solution. On all three of these measurements, GRT+CVC4 performed better than CVC4. There are number of other synthesis tracks available in the SyGuS competition, which do not involve PBE.
constraints. We note that our approach can selectively be applied as a preprocessing step for input in the PBE track without incurring an overhead on other synthesis tasks.

Although we implemented a strategy to manage a switch from the reduced grammar back to the full grammar, we found in practice that the optimal strategy for our system was to exclusively use the reduced grammar. Because we had conservatively pruned the grammar, we had no need to switch back to the full grammar.
| id | file                  | $T_C^C$ | $T_C^E$ | $|P|$ | $|P^*|$ |
|----|-----------------------|--------|--------|------|------|
| 34 | lastname-small.sl     | 1.80   | 1.84   | 4    | 4    |
| 35 | bikes-long.sl        | 1.97   | 1.76   | 3    | 3    |
| 36 | bikes-long-repeat.sl | 2.08   | 1.71   | 3    | 3    |
| 37 | lastname.sl          | 2.31   | 1.83   | 4    | 4    |
| 38 | phone-6-short.sl     | 3.23   | 1.22   | 11   | 11   |
| 39 | phone-7-short.sl     | 3.26   | 1.26   | 11   | 11   |
| 40 | initials-long-repeat.sl | 3.33 | 2.54   | 7    | 7    |
| 41 | phone-5-short.sl     | 3.72   | 1.51   | 9    | 9    |
| 42 | phone-7.sl           | 4.57   | 2.03   | 11   | 11   |
| 43 | phone-8.sl           | 4.72   | 2.17   | 11   | 11   |
| 44 | phone-6.sl           | 4.85   | 1.97   | 11   | 11   |
| 45 | phone-5.sl           | 4.88   | 2.20   | 11   | 11   |
| 46 | phone-9-short.sl     | 4.88   | 4.73   | 52   | 52   |
| 47 | phone-10-short.sl    | 8.81   | 8.28   | 49   | 49   |
| 48 | phone-9.sl           | 12.08  | 4.86   | 56   | 52   |
| 49 | phone-10.sl          | 31.23  | 8.49   | 97   | 49   |
| 50 | lastname-long.sl     | 32.40  | 25.49  | 4    | 4    |
| 51 | lastname-long-repeat.sl | 32.49 | 24.92  | 4    | 4    |
| 52 | phone-6-long-repeat.sl | 83.59 | 25.31  | 11   | 11   |
| 53 | phone-5-long-repeat.sl | 84.77 | 33.68  | 11   | 11   |
| 54 | phone-7-long.sl      | 87.83  | 26.15  | 11   | 11   |
| 55 | phone-7-long-repeat.sl | 89.13 | 26.23  | 11   | 11   |
| 56 | phone-5-long.sl      | 90.81  | 30.01  | 11   | 11   |
| 57 | phone-8-long-repeat.sl | 91.04 | 35.64  | 11   | 11   |
| 58 | phone-9-long-repeat.sl | 91.19 | 77.02  | 47   | 50   |
| 59 | phone-6-long.sl      | 98.15  | 24.75  | 11   | 11   |
| 60 | phone-8-long.sl      | 108.06 | 29.94  | 11   | 11   |
| 61 | phone-10-long-repeat.sl | 149.53 | 129.43 | 49   | 65   |
| 62 | phone-10-long.sl     | 153.32 | 133.22 | 49   | 65   |
| 63 | initials-long.sl     | TO     | TO     | -    | -    |
| 64 | phone-9-long.sl      | TO     | 3516.21| -    | 49   |

**Figure 6.4:** Synthesis results over the 30 longest running benchmarks from SyGuS Competition’s PBE Strings track.
6.7 Related

One approach to SyGuS is to directly train a neural network to satisfy the input/output examples [30, 67, 88, 112, 114, 53]. However, such approaches struggle to generalize, especially when the number of examples is small [66]. Some existing work [199, 45] aims to represent aspects of the syntax and semantics of a language in a neural network. In contrast to these existing approaches, which aim to outright solve SyGuS problems, our work acts as a preprocessor for a separate SyGuS solver. However, one could also explore using our work as a preprocessor for one of these existing neural network directed synthesis approaches. Other works have explored combining logic-directed and machine learning guided synthesis approaches [148]. This work sought to split synthesis tasks between generating high level sketches with neural networks, and fill in the holes of the sketch with an enumerative solver. Our work could be complementary to this, by assisting in pruning of the search space needed to fill in the holes.

Like our work, DeepCoder [37] and Neural-Guided Deductive Search (NGDS) [115] identify pieces of a grammar that should be removed from the grammar. However, in our parlance, these works only consider criticality, which measures how important a part of the grammar is to completing synthesis. Unlike our work, they do not consider the time savings from removing or keeping a part of the grammar. NGDS [115] does note that different models could be trained for different pieces of a grammar, however, it provides no means of automating this process. Rather, the user would have to manually elect to train individual neural networks for different grammatical elements. Work by Si et al [182] aims to learn an efficient solver for a SyGuS from scratch, rather than, as in our work, acting as a preprocessor for a separate solver.

6.8 Conclusions

In a way, by training on a dataset we generate from the output of the interpreter of the language, we are encoding an approximation of the semantics into our neural network. While the semantic approximation is too coarse to drive synthesis itself, we can use it to prune the search space of potential programs. By predicting terminals impact on synthesis time, we more conservatively remove only terminals likely to have a positive impact. In conjunction with analytically driven tools, we can then significantly improve synthesis times with very little overhead.

While we have presented GRT, which demonstrates a significant gain in performance over all existing SyGuS solvers, we still have many opportunities for further improvement. In our prediction of the potential time saved by removing a terminal from the grammar, we have simply used the average expected value over all samples in the dataset. By using a neural network here, we may be able to leverage some property of the SyGuS problem constraints to have more accurate potential time savings predictions. This would allow us, possibly in combination with a more ad-
vance prediction combination strategy, to more aggressively prune the grammar. The drawback to this approach is that we may then potentially remove too much from the grammar. One of the key features of GRT is that it introduces no new timeouts, that is, it does not remove any critical parts of the grammar.

Additionally, our prediction of criticality of a terminal uses a voting mechanism to combine the prediction based on each constraint. While this worked well in practice, this strategy ignores the potential for interaction between constraints. In our preliminary exploration, we were not able to construct a model that captures this inter-constraint interaction in a useful way. This may be a path for future work. In a similar vein, there exist a number of other works that define a criticality measure for each terminal in the SyGuS grammar [37, 115]. It may be possible to leverage these in place of our criticality measure, and in combination with our time savings prediction, to achieve better results.

So far we have only explored the PBE Strings track of the SyGuS Competition. The competition also features a PBE BitVectors track where our technique may have significant gains as well. This would require a new encoding scheme, but the overall approach would remain similar. In general, extending this work to allow for other PBE types, as well as more general constraints, would broaden the potential real-world application of SyGuS.
Chapter 7

Automated Verification and Repair of Firewalls

This chapter describes work completed in collaboration with Ennan Zhai and Ruzica Piskac. This work includes material originally published in [96, 97].

7.1 Introduction

Firewalls play an important role in today’s individual and enterprise-scale networks. A typical firewall is responsible for managing all incoming and outgoing traffic between an internal network and the rest of the Internet by accepting, forwarding, or dropping packets based on a set of rules specified by its administrators. Because of the central role firewalls play in networks, small changes can propagate unintended consequences throughout the networks. This is especially true in increasingly large and complex enterprise networks.

A single line in a firewall could, for example, allow anyone to access production services, and therefore it is critical to ensure the correctness of firewall rules. Broadly speaking, a firewall is correct if the rules of that firewall meet the specification of its administrator. There have been many efforts that aim to check the correctness of firewall rules through techniques such as firewall analysis [136, 192], verification [131], and root-cause troubleshooting [142, 200, 207]. For instance, systems like Margrave [142] and Fang [136] build an event tree recording states of an observed error, and backtrack through it to find the root causes.

While existing tools can identify the cause of an error, administrators still have to manually find an effective repair to the firewall so that it meets the specification. We propose a framework, called FireMason, that is the first to not only detects errors in firewall behaviors, but also automatically repair the firewall. Specifically, a user provides a list of examples of packet routing (e.g., all packets with a certain source IP address should be dropped) to describe what the firewall should do. The current firewall might or might not route the packets as specified in the examples. Given the complexity of enterprise-scale networks, finding such a repair requires considerable
expertise on the part of the administrator. To the best of our knowledge, there is no existing effort that automates firewall repair.

The main challenge of firewall repair is to show that a generated firewall is indeed repaired and that new rules do not change the routing of packets which are not described by the given examples. We employ an SMT solver for this task. In a nutshell, FireMason translates a given firewall into a sequence of first-order logic formulas falling into the EUF+LIA logic [141], thus allowing us to use an SMT solver for reasoning about the firewalls. By using SMT solvers, FireMason provides formal guarantees that the repaired firewalls satisfy two important properties:

- Those packets described in the examples will be routed in the repaired firewall, as specified.
- All other packets will be routed by the repaired firewall exactly as they were in the original firewall.

Taken together, these two properties allow administrators confidence that the repairs had the intended effect.

Furthermore, FireMason is also a stand-alone verification tool. The user specifies a property of interest, and FireMason will either prove that the given property holds, or if it does not hold, it produces counterexamples. As an illustration, if the user wants to verify that all packets with the IP address 1.2.3.4 should be dropped, FireMason either confirms that as correct, or it outputs an example of a packet with an IP address of 1.2.3.4 that would be accepted by the firewall.

By having a description of a firewall as a set of first-order logic formulas we reduce verification to the formula entailment problem, which we decide again using an SMT solver. Additionally, this description is useful as a formal specification of the correct behavior of a firewall implementation. The only existing specification for iptables is a man page [6], which, as a textual description, is inherently imprecise.

Due to this imprecision, developing the set of first-order logic formulas in this work required two steps. First, we carefully read the man page specification. When the man page was unclear, we turned to testing on actual implementations, to decide how to resolve the ambiguously. By specifying the behavior as first-order logic formulas, we provide future tool implementors with a precise description of iptables behavior.

Previous work has modeled firewalls using less expressive logics. For example, Zhang et al. [207] use SAT and QBF formulas, while Margrave [142], uses first-order relational logic (specifically, through the use of KodKod [192]). By using our formalism we are able to check some important and widely used, but previously out-of-scope, properties. In particular, the ability to reason about linear integer arithmetic with an SMT solver is invaluable in handling rate limits. Rate limits, which are frequently used in all modern firewalls, put a restriction on the number of packets matched in a given amount of time. Using SMT solvers we are able to efficiently reason about limiting rules. Due to the complexity of modeling limits, no previous work has considered firewalls with such rules. Such rules say, for example, that we can only accept
6 packets per minute from a certain IP address. As before, the user provides a list of examples, but with relative times. This requires reasoning about the priorities and permissions of each firewall entry, as well as the temporal patterns of the incoming packets.

We evaluated our tool using real-world firewall issues, and observed that FireMason is able to efficiently generate correct firewalls meeting administrators’ examples, without introducing any new problems. In addition, our evaluations show that FireMason scales well to enterprise-scale networks.

In summary, we make the following contributions:

- We describe a formalism to model firewalls and their behavior. This formalism allows us to use SMT solvers to prove formal guarantees, which is useful both for verification and repair.

- We explain the first method to automatically repair firewalls based on easily specified examples. Administrators can conveniently specify their desired behaviors, and automate the repair process.

- We describe using SMT solvers to efficiently reason about limit rules, which are not considered by any existing tool.

- We built a workable system that scales well with real-world examples and larger-scale datasets.

### 7.2 Preliminaries

**Repair by Example.** In this paper, we introduce the repair by example paradigm, which repairs faulty code so that it satisfies the given examples. In some ways, this resembles the programming by example paradigm [61, 130]. However, in programming by examples, the output is code which generalizes the given input examples. On the other hand, in the repairing by example paradigm the input is both an existing program and a set of examples. The goal is to adjust the input program to satisfy the examples, but otherwise to have only a small effect on the programs behavior. This allows a user to easily specify instances of faulty behavior, but have confidence that the program will continue to function as it did before. With repair by example, it is important that the effect of the changes is constrained, whereas in programming by example there is no such restriction.

**ACL-Based Firewalls.** We focus on one of the most representative types of firewalls, Access Control List (ACL) firewalls, such as iptables [6], Juniper [113], and Cisco firewalls [57], are widely used in practice. A typical ACL-based firewall contains an ordered list of rules, each of which has criteria and an action. A criterion describes which preconditions need to hold for the action to take place (e.g., dropping or accepting a packet) [166]. When a network packet is received by an ACL-based
firewall, the packet is evaluated against all the rules according to the order in which they appear. After the firewall finds the first rule with criteria satisfied by the packet, it performs the corresponding action. The criteria in a rule may refer to properties of the packet that is currently being processed, or to information tracked by the firewall. For instance

```
iptables -A INPUT -p 16 -s 123.23.12.1 -j DROP
```

has criteria denoting packets that have a protocol of 16 and a source IP address of 123.23.12.1, and an action specifying those packets should be dropped.

Actions are either terminating or non-terminating. Terminating actions end the packet’s traversal. For example, once a packet is accepted or dropped, it no longer checks other rules in the ACL. Non-terminating actions (such as printing to a log file) allow a packet to continue traversing the ACL rules and match more rules. An action might also refer to another ACL, which then needs to be used to evaluate the packet. We refer to this as a jump to a different ACL.

The ACL jumps cannot form a loop. That is, if ACL $A_1$ contains a jump to ACL $A_2$, there can be no jumps from $A_2$ back to $A_1$. However, suppose a packet is evaluated against all rules in an ACL $A_2$ and does not match any rule with a terminating action. The packet will then continue being evaluated at the next rule in $A_1$. If the packet started in $A_1$, and the packet does not match any rule in $A_1$ with a terminating action, the packet will be routed according to the policy of $A_1$. The policy is the default action on packets that start in a given ACL, and must be to either accept or drop the packet [14].

**Rate Limiting Rules.** Rate limiting rules are used when an administrator wants to restrict the amount of packets matching a certain rule, for example the amount of packets arriving from some IP address. We call a firewall with such rules a rate limiting firewall. In many firewalls, including iptables [14], Juniper [113], and Cisco [57] firewalls, a limit is a criterion that specifies how frequently a rule can be matched. A limit is implemented as a counter $l$, and a match is possible only if a packet satisfies rule’s criteria and $l > 0$. A rate limiting behavior is specified through two parameters: an average rate of packets per some time unit, $ra$, and a burst limit, $b$. Whereas other criteria are based solely on evaluating a single packet, a limit requires the firewall to maintain its counter, and hence warrants special consideration.

Rate limiting firewalls use the token bucket algorithm [187] to determine if a packet should be dropped or further processed. The counter $l$ decrements when a packet matches the rule, and increments every $1/ra$ time units. The counter can never exceed the burst limit $b$. The next example shows how limits work in practice:

**Example.** Suppose that we set a limit on incoming packets, with $ra = 6$ packet / minute and $b = 3$ packets. The firewall is initialized with $l = b = 3$. If we do not exceed the limit, we will accept incoming packets. If we do exceed it, we will drop them. As shown in Fig. 7.1 suppose that at times 0, 5, 9, and 17 seconds, we receive 1 packet, and at time 16 seconds we receive 2 packets.

At the end of the fifth second, $l = 3 - 1 = 2$ since 1 packet arrives. Similarly, at
the end of the ninth second, \( l = 2 - 1 = 1 \) since 1 packet arrives. At the beginning of the tenth second, \( l \) is incremented again to 2. At the sixteenth second, we receive two packets. Both will be accepted, but it drains the limit completely. Therefore, since \( l = 0 \) when the fourth packet arrives, that packet does not match the limit, and is dropped.

### 7.3 Motivating Examples

**Stateless Firewall Repairing Example 1.** An example given in Figure 7.2 demonstrates the basic functionality of FireMason. The example is inspired by a StackExchange post [2]. An administrator is maintaining firewall rules written in iptables [6], one of the most representative firewall script languages. The firewall initially contained rules labeled R1 to R5.

If the administrator wants to block TCP requests coming from the IP address 172.168.14.6, they may try expressing that as a rule and putting it at the end of the current firewall, cf. rule R6 in Figure 7.2. Such an action is very common in enterprise-scale firewall management, because administrators prefer appending a new rule to the existing rules [129].

FireMason can be used as a standard firewall analysis tool. To test the changes, the administrator can execute the query:

\[
\text{verify (INPUT, protocol = tcp, source_ip = 172.168.14.6 => DROP)}
\]

FireMason reports to the administrator that the specification is violated, and gives an example of a packet that will be incorrectly routed (For example, a TCP packet with the SYN flag set, a source ip address of 172.168.14.6, and a destination port of 22. Such a packet would be accepted by R3 or R4).

Knowing that the repair does not work as intended, the administrator can also use FireMason as a repair tool. They provide an example of what should be changed in the firewall and invokes FireMason as shown in Figure 7.2 (b).

FireMason returns a repaired firewall, Figure 7.2 (c), to the administrator. The new rule is positioned close to similar rules, namely, those rules related to the TCP
Figure 7.2: An example of a firewall repair problem.

protocol. This positioning is very important. While one may argue that directly appending a rule to the top of firewall can also make the firewall behave correctly (in terms of functionality), this method would, unfortunately, destroy the structure and organization of the firewall. Much like traditional code, keeping the firewall rules organized is important to facilitate later understanding and maintaining. Most importantly, the rule is positioned so that any packet matching the user provided example is guaranteed to be dropped. Rule R1 specifies a protocol other than TCP, and so never matches such a packet. A packet matching the example could match rule R2, but rule R2 drops any matching packet anyway.

This whole example also showed that placing a rule at a wrong place can change the behavior of a firewall. FireMason provides formal guarantees that for every packet, which is not covered by the user provided examples, the original firewall and in the repaired firewall will invoke the same action.

**Stateless Firewall Repairing Example 2.** Inspired by posts on ServerFault [8, 5], consider an administrator who wants to ensure that the local host, and only the local host, can access the web server at 1.2.3.4. Any traffic not from the local host, but trying to access the ip address 1.2.3.4, should be dropped. To solve this with FireMason, one approach would be to write two examples:

```
repair(INPUT, protocol = 6, 
       source_ip = 127.0.0.1 => DROP)
```

Unfortunately, this is redundant and hard to read: it is easy to miss the not on the second line. To make this sort of task easy, we introduce two keywords: onlyif and unless. We can demonstrate the desired behavior in a single example, using the
onlyif keyword, as follows:

\[
\text{repair(} \text{INPUT, destination\_ip = 1.2.3.4 } \Rightarrow \text{ ACCEPT}
\text{ onlyif source\_ip = 127.0.0.1) }
\]

Equivalently, the administrator could write the example with the unless keyword:

\[
\text{repair(} \text{INPUT, destination\_ip = 1.2.3.4 } \Rightarrow \text{ DROP}
\text{ unless source\_ip = 127.0.0.1) }
\]

In either case, FireMason will create and add two new rules:

\[
\text{iptables -A INPUT -d 1.2.3.4/32 -s 127.0.0.1/32 -j ACCEPT}
\text{iptables -A INPUT -d 1.2.3.4/32 ! -s 127.0.0.1/32 -j DROP}
\]

which ensure the firewall has the desired behavior.

**Rate Limiting Rule Repairing Example.** We next show how an administrator can use FireMason to add/repair rate limiting rules. To the best of our knowledge no existing firewall analysis tools can address this problem. Suppose an administrator wants to allow TCP connections with the SYN flag set once every 10 seconds (a task inspired by a forum post on StackExchange [9].) To do this, the administrator may provide a sequence of example packets and relative times, in seconds:

\[
\text{repair(} \text{INPUT, SYN, time = 0 } \Rightarrow \text{ ACCEPT;}
\text{INPUT, SYN, time = 5 } \Rightarrow \text{ DROP;}
\text{INPUT, SYN, time = 10 } \Rightarrow \text{ ACCEPT)}
\]

As a result FireMason creates and inserts two new rules:

\[
\text{iptables -A INPUT -m limit --limit 6/minute \ }
\text{--limit--burst 1 -p tcp --tcp-flags SYN SYN --j ACCEPT}
\text{iptables -A INPUT -p tcp --tcp-flags SYN SYN --j DROP}
\]

This limit satisfies the administrator’s requirement. Only one TCP SYN packet can be received every 10 seconds.

### 7.4 System Design

Figure 7.3 shows the overview of FireMason’s workflow. FireMason takes as input a firewall and a user command, which can be either a verification command or a repair command and contains a list of examples.

FireMason first translates the firewall and examples into a set of formulas belonging to a fragment of first-order logic. The translation (described in Sec. 7.4.1) produces two sets of EUF+LIA formulas [141], which means we can use an SMT solver to reason about firewalls.

The verification process (described in Sec. 7.4.2) checks if the rules specified in the examples are violated by the new firewall. If there are such rules, FireMason reports counterexamples to the user.
Figure 7.3: The workflow overview of FireMason.

The repair process first checks consistency of the input examples and reports to the user if they are contradictory (Sec. 7.4.4). This also allows us to detect sets of examples that can be used to generate rate limiting rules. FireMason creates any needed rate limiting rules to handle provided examples. (Sec. 7.4.6). FireMason next runs the repair algorithm (Sec. 7.4.7). Finally, FireMason adds the rules to the firewall (Sec. 7.4.3), checks if there are redundant rules in the newly generated firewall (Sec. 7.4.8), and outputs a correct firewall.

7.4.1 Encoding Firewalls and Examples as FOL Formulas

Translating Examples. FireMason starts with a list of examples provided by the user, either for a verification or a repair process. Those examples are expressed using the grammar:

\[
\begin{align*}
\text{comm} & := \text{verify}((\text{acl}, \text{rule})^+) \mid \text{repair}((\text{acl}, \text{rule})^+) \\
\text{rule} & := \text{precon}^+ \Rightarrow \text{action} \mid \text{precon}^+ \Rightarrow \text{action} \text{ cond} \\
\text{precon} & := \text{protocol} = \text{INT} \mid \text{source_ip} = \text{IP_ADDRESS} \\
& \quad \mid \text{destination_port} = \text{INT} \mid \ldots \mid \text{not precon} \\
\text{action} & := \text{ACCEPT} \mid \text{DROP} \mid \ldots \\
\text{cond} & := \text{onlyif precon} \mid \text{unless precon} \\
\text{acl} & := \text{STRING} \quad \text{\textbackslash \textbackslash ACL Name}
\end{align*}
\]

We represent every example by a tuple \((n, r, t)\), where \(n\) is the name of the ACL to which the rule \(r\) applies, and \(t\) is the time given in the example. If no time
was given, we set $t = \emptyset$. This tuple is then used in FireMason’s algorithms. For instance, the example $\text{repair(\text{protocol} = 16, \text{time} = 5 \Rightarrow \text{ACCEPT})}$ is translated to $(\text{INPUT, protocol} = 16 \Rightarrow \text{ACCEPT}, 5)$.

Adding a cond to a rule allows for stronger statements about the desired behavior. On forums, we noticed users would often ask for rules that implement a certain behavior only if (or, conversely, unless) some condition is met. To help users write down these types of conditions, we introduce two keywords: onlyif and unless. To define these precisely, we use a function on the actions:

$$\text{flipAction}(a) = \begin{cases} 
\text{ACCEPT} & a = \text{DROP} \\
\text{DROP} & a = \text{ACCEPT} 
\end{cases}$$

Then, we translate the example $p \Rightarrow a \text{ onlyif } q_1 \ldots q_n$ into the rules:

$$p \land q_1 \land \ldots \land q_n \Rightarrow a$$
$$p \land \neg q_1 \Rightarrow \text{flipAction}(a)$$
$$\ldots$$
$$p \land \neg q_n \Rightarrow \text{flipAction}(a)$$

and, similarly, translate $p_1 \Rightarrow a \text{ unless } q_1 \land \ldots \land q_n$ into the rules:

$$p_1 \land \neg q_1 \land \ldots \land \neg q_n \Rightarrow \text{flipAction}(a)$$
$$p_1 \land q_1 \Rightarrow a$$
$$\ldots$$
$$p_1 \land q_n \Rightarrow a$$

**Translating Firewall Scripts.** Broadly speaking, FireMason describes a firewall's behavior with a sequence of first-order logic formulas. The translation results in formulas that are amenable for reasoning with a SMT solver. Such encoding has two benefits: the computational burden of checking consistencies or finding redundant rules is done by a solver. In addition, we can easily formalize that the repaired firewall is indeed repaired and that only packets described by the examples will be treated differently and according to the specification.

While the majority of the rules could be easily translated to first-order formulas, one obstacle is when the firewall contains jumps. This becomes an issue especially when the ACL also uses limits. Consider, for example, an ACL that has at least two jumps to an ACL $A_1$. Let us assume that the ACL $A_1$ has some limit rules. If a packet has to go through both the jumps, then when it reaches the limit in $A_1$ the second time, the limit in $A_1$ will have counted the packet twice.

We introduce a data structure, called a FirewallMap, which simplifies modeling of jumps and limits. A FirewallMap $\mathcal{M}$ maps unique IDs (we use natural numbers) to tuples of ACL names and lists of the ACLs rules. A rule is modeled as an implication, where a set of criteria implies an action. Possible actions are \text{ACCEPT}, \text{DROP}, and \text{GO}(a). \text{GO} is parameterized by a natural number $a$, and represents a jump to the ACL
Figure 7.4: In a FirewallMap, ACL’s are duplicated for each point that they can be jumped to from.

with ID $a$. In the FirewallMap $\mathcal{M}$ there is at most one GO referring to a particular ACL ID. Every rule in $\mathcal{M}$ is assigned a tuple $(a, r)$, where $a$ is an ID of the ACL where the rule appears and $r$ is an ID of the rule in that ACL. This way there exists a single unique path through the FirewallMap to reach any individual rule. Without this property, it would be significantly more difficult to correctly model the order in which rules must be checked. Any ACL jumped to from more than one place in the original firewall is duplicated and assigned multiple IDs, as shown in Figure 7.4. The ACL mapped to by each of these IDs is identical, except any GOs in them must also have different IDs. We refer to these duplicated ACLs as equivalent to each other.

Language for Encoding Firewall Behavior into Formulas. We now describe a first-order language that we use to model firewalls and packets. Most of these predicates take a FirewallMap $\mathcal{M}$ as an argument. One can think of $\mathcal{M}$ as a firewall script.

Table 7.1 lists a selection of those predicates, functions, and their meanings. FireMason uses these functions and predicates to encode the firewall.

For example, if rule $r$ in ACL $a$ in a FirewallMap $\mathcal{M}$ had criteria specifying that it matched a packet $p$ with protocol 17 and destination port 8, then FireMason
translates that as follows:

\[
\text{matches\_criteria}(\mathcal{M}, p, a, r) \\
\Leftrightarrow (\text{protocol}(p) = 17 \land \text{destination\_port}(p) = 8)
\]

The predicates are designed to make it easy to write formulas with important properties. For example, \text{reaches} is used to describe which rules a packet is evaluated against, while \text{matches\_criteria} indicates whether a packet \textit{would} satisfy the criteria of a rule. Building on these, \text{matches\_rule} is true if and only if a packet both reaches a rule and satisfies the rule.

Table 7.2 shows some axioms describing general relationships between the predicates and functions, and encoding actual firewall behavior. All formulas in the table are implicitly universally quantified, with additional guards \(0 \leq p < \text{max\_packets}\) and \(\text{valid\_rule}(\mathcal{M}, a, r)\). Since the sets of values for \(\mathcal{M}, p, a,\) and \(r\) are finite, these formulas (as well as the definitions of \text{reaches\_end}, \text{reaches\_return}, \text{reaches\_exit},\) and \text{matches\_rule} from Table 7.1) can be finitely instantiated. Thus, no universal quantifiers are needed, and we encode the firewalls in the decidable EUF+LIA logic [141].

Largely, the axioms in Table 7.2 describe reachability, and how \text{reaches} interacts with the other predicates. As an example, consider:

\[
\text{reaches}(\mathcal{M}, p, a, r) \land \neg\text{matches\_criteria}(\mathcal{M}, p, a, r) \implies \text{reaches}(\mathcal{M}, p, a, r + 1)
\]

and

\[
\text{reaches}(\mathcal{M}, p, a, r + 1) \implies \text{reaches}(\mathcal{M}, p, a, r)
\]

The first axioms captures the property that, if a packet has reached a rule, and does not match (satisfy) the criteria of that rule, the packet will reach the next rule. The second axioms states that in order for a packet to reach a rule, a packet must have also reached the rule that directly precedes it.

\textbf{Modeling Limits.} Limits have two attributes: an average rate \(r_{a}\) in packets per time unit, and a burst limit of \(b\) packets. Each limit also uses a counter to decide if a packet can match the rule. Intuitively, it may seem one could easily model the behavior of a limit using linear integer arithmetic. However, \(r_{a}\) might not be an integer when the units are converted to seconds. For example, 31 packets per minute is \(51\frac{1}{6}\) packets per second. Therefore, we introduce a new \(sub\) variable, which represents the time unit used by the limit, converted to seconds. For example, a limit with an average rate of 31 packets per minute and a burst of 10 will be assigned \(r_{a} = 31, sub = 60,\) and \(b = 600\) in the formula. Essentially, this corresponds to multiplying the whole formula by \(sub\), to reduce the problem to integers. \(r_{a}\) is now 31 \textit{tokens} per second, we have a maximum of 600 tokens, and we require 60 tokens to send a single packet.

To have a correct counter of the number of packets, in our model we assign to each limit from the firewall two integer IDs, a main ID \(i\) and a secondary ID \(j\). Limits for the same rule in equivalent ACLs all have the same main ID. The secondary IDs
Table 7.1: Partial list of predicates and functions used to model firewalls.

<table>
<thead>
<tr>
<th>Pred/Func</th>
<th>Meaning of the pred/func</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid_acl((M, a))</td>
<td>There exists an ACL with ID (a) in FirewallMap (M)</td>
</tr>
<tr>
<td>valid_rule((M, a, r))</td>
<td>valid_acl((M, a)) and there exists a rule with ID (r) in (a)</td>
</tr>
<tr>
<td>matches_criteria((M, p, a, r))</td>
<td>Packet (p) satisfies the criteria of rule (r) in ACL (a) in FirewallMap (M)</td>
</tr>
<tr>
<td>reaches((M, p, a, r))</td>
<td>Packet (p) reaches rule (r) in ACL (a) in FirewallMap (M)</td>
</tr>
<tr>
<td>starting_acl((M, a))</td>
<td>Returns true if ACL (a) is not jumped to from some other ACL</td>
</tr>
<tr>
<td>is_go((act))</td>
<td>Returns whether the action (act) is \text{GO}(a) for some arbitrary (a)</td>
</tr>
<tr>
<td>reaches_end((M, p, a))</td>
<td>reaches((M, p, a, acl_length((M, a))))</td>
</tr>
<tr>
<td>reaches_return((M, p, a))</td>
<td>reaches((M, p, a, r))&amp;&amp; \text{rule_action}((M, a, r)) == \text{RETURN}</td>
</tr>
<tr>
<td>reaches_exit((M, p, a))</td>
<td>reaches_end((M, p, a))\lor reaches_return((M, p, a))</td>
</tr>
<tr>
<td>matches_rule((M, p, a, r))</td>
<td>matches_criteria((M, p, a, r))&amp;&amp; reaches((M, p, a, r))</td>
</tr>
<tr>
<td>matches_example((p, e))</td>
<td>Packet (p) matches the criteria of an example (e)</td>
</tr>
<tr>
<td>protocol((p))</td>
<td>The protocol of packet (p)</td>
</tr>
<tr>
<td>acl_length((M, a))</td>
<td>Returns the number of rules in ACL (a)</td>
</tr>
<tr>
<td>max_packets</td>
<td>Returns the maximum number of packets to be considered</td>
</tr>
<tr>
<td>terminates_with((M, p))</td>
<td>Returns if the FirewallMap (M) would ACCEPT or DROP packet (p)</td>
</tr>
<tr>
<td>rule_action((M, a, r))</td>
<td>Returns the action of rule (r) in ACL (a) in FirewallMap (M)</td>
</tr>
<tr>
<td>insert_rule((M, R, a, r))</td>
<td>Returns FirewallMap (M), but with rule (R) inserted in ACL (a) as rule (r)</td>
</tr>
<tr>
<td>equivalent((M, n))</td>
<td>Returns the set of IDs in FirewallMap (M) for the ACL named (n)</td>
</tr>
<tr>
<td>go_acl((act))</td>
<td>For (act = \text{GO}(a)) returns (a), otherwise -1</td>
</tr>
</tbody>
</table>
Table 7.2: Formulas to model a firewall, and packets that firewall is processing.

<table>
<thead>
<tr>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_1 \neq a_2 \land \text{reaches}(\mathcal{M}, p, a_1, 0) \land \text{starting}<em>\text{acl}(\mathcal{M}, a_1) \land \text{starting}</em>\text{acl}(\mathcal{M}, a_2) \implies \neg \text{reaches}(\mathcal{M}, p, a_2, 0) )</td>
</tr>
<tr>
<td>( \text{reaches}(\mathcal{M}, p, a, r) \land \neg \text{matches}_\text{criteria}(\mathcal{M}, p, a, r) \implies \text{reaches}(\mathcal{M}, p, a, r + 1) )</td>
</tr>
<tr>
<td>( \text{reaches}(\mathcal{M}, p, a, r + 1) \implies \text{reaches}(\mathcal{M}, p, a, r) )</td>
</tr>
<tr>
<td>( \text{matches}<em>\text{rule}(\mathcal{M}, p, a, r) \land \text{is}</em>\text{go}(\text{rule}<em>\text{action}(\mathcal{M}, a, r)) \equiv \text{reaches}(\mathcal{M}, p, \text{go}</em>\text{acl}(\text{rule}_\text{action}(\mathcal{M}, a, r)), 0) )</td>
</tr>
<tr>
<td>( \text{reaches}<em>\text{exit}(\mathcal{M}, p, \text{go}</em>\text{acl}(\text{rule}_\text{action}(\mathcal{M}, a, r))) = \text{reaches}(\mathcal{M}, p, a, r + 1) )</td>
</tr>
<tr>
<td>( \text{reaches}_\text{return}(\mathcal{M}, p, a) \implies \neg \text{reaches}(\mathcal{M}, p, a, r + 1) )</td>
</tr>
<tr>
<td>( \text{matches}<em>\text{rule}(\mathcal{M}, p, a, r) \land \text{terminating}(\text{rule}</em>\text{action}(\mathcal{M}, p, a, r)) \implies \neg \text{reaches}(\mathcal{M}, p, a, r + 1) )</td>
</tr>
<tr>
<td>( \text{terminates}<em>\text{with}(\mathcal{M}, p) = \text{rule}</em>\text{action}(\mathcal{M}, p, a, r) )</td>
</tr>
<tr>
<td>( \text{reaches}<em>\text{end}(\mathcal{M}, p, a, r) \land \text{starting}</em>\text{acl}(\mathcal{M}, a) \implies \text{terminates}_\text{with}(\mathcal{M}, p) = \text{policy}(\mathcal{M}, a) )</td>
</tr>
</tbody>
</table>

Table 7.3: Logical formulas related to limits, all variables are implicitly universally quantified with additional constraints that rule \( r \) in ACL \( a \) has a limit with main ID \( i \) and secondary ID \( j \), and \( 0 \leq p < \text{max}_\text{packets} \). We use \( j_{\text{max}}(i) \) to denote the maximum secondary ID for the limit with main ID \( i \).

<table>
<thead>
<tr>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall p. p \geq 1 \implies \text{arrival}<em>\text{time}(p) \geq \text{arrival}</em>\text{time}(p - 1) )</td>
</tr>
</tbody>
</table>
| \( \Delta t(p) = \begin{cases} 
\text{arrival}_\text{time}(p) - \text{arrival}_\text{time}(p - 1) & \text{if } 1 \leq p < \text{max}_\text{packets} \\
0 & \text{otherwise} 
\end{cases} \) |
| \( \text{counter}_\text{pre}(\mathcal{M}, i, j, p) = \begin{cases} 
\text{counter}_\text{post}(\mathcal{M}, i, j - 1, p) & \text{if } j \geq 1 \\
\text{min}(\text{counter}_\text{post}(\mathcal{M}, i, j - 1, p), \text{counter}_\text{post}(\mathcal{M}, i, j, p)) & \text{if } p \geq 1 \text{ and } j = 0 \\
j_{\text{max}}(i), p - 1 + ra \ast \Delta t(p), b & \text{otherwise} 
\end{cases} \) |
| \( \text{counter}_\text{post}(\mathcal{M}, i, j, p) = \begin{cases} 
\text{counter}_\text{pre}(\mathcal{M}, i, j, p) - \text{sub} & \text{if } \text{counter}_\text{pre}(i, j, p) \geq \text{sub} \land \\
\text{counter}_\text{pre}(\mathcal{M}, i, j, p) & \text{otherwise} 
\end{cases} \) |
start from 0, and they increase every time a packet could meet that limit. We define two functions, \( \text{counter}_\text{pre}(\mathcal{M}, i, j, p) \) and \( \text{counter}_\text{post}(\mathcal{M}, i, j, p) \), parameterized by the limit’s main and secondary IDs, and the packet ids. They are used to track the value of the counter at any given point in time. \( \text{counter}_\text{pre}(\mathcal{M}, i, j, p) \) is the value of counter \((i, j)\) immediately before packet \( p \) reaches the rule containing that limit. \( \text{counter}_\text{post}(\mathcal{M}, i, j, p) \) is the value of that counter immediately after. To check if a limit will allow a packet to match, we check if \( \text{counter}_\text{pre}(\mathcal{M}, i, j, p) \geq \text{sub} \).

The SMT formulas related to computation of limits are given in Table 7.3 Note that, since we multiply \( ra \) and \( \Delta t(p) \), we must know one of their values for this formula to be in LIA. Fortunately, when reading a limit from an existing firewall script we know \( ra \). In Sec. 7.4.6 we explain how \( \Delta t(p) \) is known in advance from the examples, so we can obtain \( ra \) from the SMT solver.

### 7.4.2 Firewall Verification

Since firewalls are not annotated with standard specifications, systems for verifying firewalls, such as Margrave [142], verify firewalls against user provided queries. When performing the verification process, FireMason also checks if the given examples violate the firewall rules. In particular, it is helpful to be able to check that packets with certain attributes will be accepted or dropped by a firewall. For example, an administrator might want to verify that any packet received from a certain IP address will be dropped by the firewall.

We first explain the verification process for examples without time (limit) constraints. Given an example, \( e = (n, c \Rightarrow \text{act}, \emptyset) \) (as described in Sec. 7.4.1), and a firewall \( \mathcal{M} \), we verify \( e \) against \( \mathcal{M} \) by showing that the following formula \( F \) is valid:

\[
\forall p, a. \ a \in \text{equivalent}(\mathcal{M}, n) \land \text{reaches}(\mathcal{M}, p, a, 0) \\
\land \text{matches}_\text{example}(p, e) \Rightarrow \text{terminates}_\text{with}(\mathcal{M}, p) = \text{act}
\]

Formula \( F \) states that every packet arriving to ACL \( n \) and satisfying criteria \( c \) terminates with action \( a \). Note that when negated, the formula is only existentially quantified.

To verify a list of examples with times, \( e_k = (n_k, c_k \Rightarrow \text{act}_k, t_k) \), for \( 0 \leq k \leq N \) we apply a similar procedure. After setting up all packets with appropriate times, the verification condition states that at least one packet does not terminate as desired (expressed already in the negated form):

\[
\forall k \exists a. 0 \leq k \leq N \land a \in \text{equivalent}(\mathcal{M}, n_k) \\
\land \text{reaches}(\mathcal{M}, p_k, a_k, 0) \land \text{matches}_\text{example}(p_k, e_k) \\
\land ( \bigvee_{0 \leq j \leq N} \text{terminates}_\text{with}(\mathcal{M}, p_j) \neq \text{act}_j )
\]
7.4.3 Adding Rules

Here we outline how to create firewall rules from the provided examples. We first focus on stateless rules. Generating rate limiting rules is described in Sec. 7.4.6. The repair algorithm, Algorithm 2 from Sec. 7.4.7, assigns each rule a position where it should be placed. After positions are assigned, translating user provided examples to the iptables language is rather straightforward. For example, the tuple

\[(\text{INPUT}, \text{protocol} = 6, \text{source}\_ip = 1.2.3.4 \Rightarrow \text{ACCEPT}, \emptyset)\]

translates to a rule

\[\text{iptables} -A \text{INPUT} -p 6 -s 1.2.3.4 -j \text{ACCEPT}.\]

7.4.4 Consistency Checking

The purpose of consistency checking is both to let the administrator know whether the provided examples contradict each other, and to detect when to invoke the algorithm for addressing limits. Consider the two examples below:

repair (\text{INPUT}, \text{protocol} = 17 \Rightarrow \text{ACCEPT}),
repair (\text{INPUT}, \text{source}\_ip = 1.1.0.0/16 \Rightarrow \text{DROP})

If a packet with \text{protocol} = 17 and a source IP address of 1.1.1.1 enters the INPUT ACL, it is not clear whether such a packet should be accepted or dropped. We consider these examples \text{rule inconsistent}.

Formally, we say two examples, \((n_1, c_1 \Rightarrow \text{act}_1, t_1)\) and \((n_2, c_2 \Rightarrow \text{act}_2, t_2)\) are rule inconsistent if \(n_1 = n_2, c_1 \land c_2\) is satisfiable by a single packet, and \(\text{act}_1 \neq \text{act}_2\). We find the contradictory examples by using an SMT solver and we inform the administrator about ambiguities. Note that this definition makes no reference to time, and handling of rule inconsistent examples with different times will be covered in Sec 7.4.6.

7.4.5 Formal Guarantees for Repaired Firewalls

FireMason offers two guarantees on the behavior of repaired firewalls. The first guarantee is the packets or sequences of packets described by the examples are correctly routed in the repaired firewall. The second guarantee is that the routing of every packet not described by the examples is the same as it was in the original firewall. Together, these guarantees allow an administrator to be confident that the repairs had the intended effect, and only the intended effect.

Here we give formulas which can be used by an SMT solver to check if the formal guarantees hold.
For given examples of the form \( e_k = (n_k, crit_k \Rightarrow act_k, \emptyset) \), for \( 0 \leq k < N \), the first guarantee can be written with Formula (7.4.5),

\[
\forall k, a. 0 \leq k < N \land a \in \text{equivalent}(M, n_k) \land \\
\text{matches\_example}(k, e_k) \land \text{reaches}(M', k, a, 0) \\
\implies \text{terminates\_with}(M', k) = act_k
\]

Now suppose we have examples with relative times, \( e_k = (n_k, crit_k \Rightarrow act_k, t_k) \).
Without loss of generality, assume that for \( k_1 < k_2 \), we have \( t_{k_1} < t_{k_2} \). In this case we ensure that packets arriving at the appropriate times, with the appropriate criteria, are correctly routed, given that no other packets matching the examples criteria are processed before their arrival. Formally, we write:

\[
\forall k, a. 0 \leq k < N \land a \in \text{equivalent}(M, n_k) \\
\bigwedge_{0 \leq m \leq k} \left( \text{arrival\_time}(m) = t_m \land \text{matches\_example}(m, e_m) \right) \\
\land \text{reaches}(M, m, a, 0) \\
\bigwedge_{m' > k} \text{nonexample}(M, m', k) \\
\implies \text{terminates\_with}(M', k) = act_k
\]

where we use the predicate \text{nonexample} to determine if the packet \( p \) either does not correspond to or arrives after the last relevant example.

\[
\text{nonexample}(M, p, e) = \\
\forall k, a. 0 \leq k < e \land a \in \text{equivalent}(M, n_k) \implies \\
t_e < \text{arrival\_time}(p) \lor \neg \text{reaches}(M, p, a, 0) \\
\lor \left( \bigwedge_{0 \leq m \leq k} \neg \text{matches\_example}(p, e_m) \right)
\]

The second guarantee, that the changes we make do not affect more packets than intended, is stated as Formula (7.4.5):

\[
\forall p. \text{terminates\_with}(M, p) = \text{terminates\_with}(M', p) \\
\lor \left( \exists k, a. 0 \leq k < N \land a \in \text{equivalent}(M, n_k) \\
\land \text{matches\_example}(p, e_k) \land \text{reaches}(M, k, a, 0) \right)
\]

### 7.4.6 Rate Limiting Rules Generation

After the consistency checking, some examples may have to be resolved via rate limiting. Specifically, this is required for rules that are rule inconsistent, but have relative times. Algorithm 1 generates rate limiting rules satisfying these examples.
input : $E$, the list of examples, all with relative times, optional parameters $minRulesAndLimits$ and $minTotalSub$ (both default to $\emptyset$)

output: $r$ a list of rules

$E' \leftarrow \text{[]}$;

foreach $(n, r, t) \in E$ do
    $r_2 \leftarrow r$, with a limit template, consisting of symbolic values for $ra$, $b$, $sub$, and $useLimit$, and a Boolean $enableRule$ added to the criteria
    $E'.append((n, r_2, t))$;
end

sortByNameByTime($E'$);

if $minRulesAndLimits \neq \emptyset$ and $minTotalSub \neq \emptyset$ then
    Assert $rulesAndLimits < minRulesAndLimits$
    $\lor (rulesAndLimits = minRulesAndLimits \land totalSub < minTotalSub)$
end

Convert $E'$ to SMT formulas, create formulas defining score and totalSub, run SMT Solver;

$sat \leftarrow \text{getSat}$;

if $sat = UNSAT$ then
    $r \leftarrow \text{getRulesFromModel(model)}$;
    return $r$;
end
else
    model $\leftarrow \text{getModel}$;
    $(rulesAndLimits, totalSub) \leftarrow \text{getScore(model)}$;
    call this recursively, to lexicographically minimize $(rulesAndLimits, totalSub)$;
end

Algorithm 1: Limit Generating Algorithm
Our algorithm takes a list of rule inconsistent examples, \( E \), each with a time. It returns an ordered list of satisfying rules, which are later inserted into the firewall using Algorithm 2.

Recall that we may express an example as consisting of an ACL name, a rule, and a time. We create \( E' \) from \( E \), by adding two criteria to each examples rule. The first is a limit template, which uses variables in place of actual integers for \( ra, b, \) and \( sub \). It also has a Boolean variable \( useLimit \), which enables and disables the limit. The second criterion is a Boolean, \( enableRule \). Packets can match the rule if and only if \( enableRule \) is true. We will use this template with an SMT solver to search for the solution that requires the fewest limits and rules.

We sort \( E' \) into distinct groups according to which ACL the rules are meant to be added to, and then sort each group by ascending time, at line ???. We extract the rules from \( E' \) into lists (ACLs) to form a templated FirewallMap \( M \). This allows us to convert to an SMT formula, using exactly the same formulas and logic as in Sec. 7.4.1.

For each original example, \( e_p = (n_p, c_p \Rightarrow act_p, t_p) \), we pick \( a \in \text{equivalent}(M, n_p)\) and assert that the packet with ID \( p \) matches the requirements of that example:

\[
\text{arrival\_time}(p) \land \text{matches\_example}(p, e_p) \\
\land \text{reaches}(M, p, a, 0) \land \text{terminates\_with}(M, p) = act_p
\]  

(7.1)

For all the pairs \( 0 \leq r, q < \text{length}(E'), r \neq q \), we check if \( c_r \land \lnot c_q \) is satisfiable by a single packet. For each pair which is, we assert:

\[
\lnot \text{matches\_example}(r, e_q)
\]  

(7.2)

The SMT solver can then find values for each \( ra, b, sub, u, \) and \( enableRule \) that guide the packets as required by the examples. Formula (7.1) ensures that the found solution satisfies the requirements of the examples sequence. Formula (7.2) ensures that the SMT solver does not make assumptions about packets criteria that the user likely does not intend. For example, if the administrator provided the examples:

\[
\text{repair}(
\begin{align*}
\text{INPUT, protocol} &= 17, \text{time} = 0 \Rightarrow \text{ACCEPT}; \\
\text{INPUT, protocol} &= 17, \text{time} = 5 \Rightarrow \text{DROP}; \\
\text{INPUT, source\_ip} &= 1.1.0.0/16, \text{time} = 10 \Rightarrow \text{ACCEPT}; \\
\text{INPUT, source\_ip} &= 1.1.0.0/16, \text{time} = 15 \Rightarrow \text{DROP}
\end{align*}
)\]

Formula (7.2) would prevent the SMT solver finding a solution that required any of the packets satisfying protocol = 17 AND source\_ip = 1.1.0.0/16.

Such a model is always possible to find. One valid solution is to set all the enableRule to true, all the bursts to 1, and all the rates and subs such that the limit recharging even once takes longer than the total time between the first and last packet arriving. Then, each packet will be sorted according to the rule that came from its
modified example.

To make our solution capable of handling more general cases, we assign a lexicographic score to our formula. The first value is calculated by adding the number of limits and the number of non-ignored rules, which we call $\text{rulesAndLimits}$. The second value is the sum of the limit’s $\text{sub}$ values, which we call $\text{totalSub}$. We aim to make this score as small as possible. This can be done by repeatedly asserting there exists a formula with a better score. If $(\min\text{RulesAndLimits}, \min\text{TotalSub})$ is the current best score, we assert:

$$
\text{rulesAndLimits} < \min\text{RulesAndLimits} \lor \left(\text{rulesAndLimits} = \min\text{RulesAndLimits} \land \text{totalSub} < \min\text{TotalSub}\right)
$$

When the SMT solver returns UNSAT, we can guarantee we found the solution which minimizes the number of rules plus the number of limits used.

There are two small potential problems with this approach, and luckily, both have straightforward solutions. First, recall from Sec. 7.4.1 that the model involves the value of $ra \ast \Delta t(p)$, but to stay in the theory of LIA, we must avoid multiplying two variables. In that section, there was an assumption that the value of $ra$ was known, whereas here it clearly is not. Fortunately, while we do not know the value of $ra$, we can precompute, and fix as a constant, the time difference between neighboring packets, $\Delta t(p)$.

Second, some firewalls languages constrain the value of $\text{sub}$ to a fixed list of possible values $s_1, \ldots, s_v$. This can be handled through one additional assertion per $\text{sub}$ value, $\forall u=1 \rightarrow v, \text{sub} = s_u$. This occasionally leads to cases where there is no valid way to generate the limits, but such cases can be detected when the first call to the SMT solver is UNSAT.

### 7.4.7 Repair Algorithms

Given the formulas representing the target firewall and examples, we need to run a repair algorithm to generate a correct firewall based on the examples. We will first consider rule insertion for non-rule inconsistent examples. Then, we will explain how this same algorithm can be used to insert the rate limiting rules found by Algorithm 1. Suppose we have $N$ non-rule inconsistent examples, $e_1 = (n_1, r_1 = (c_1 \Rightarrow \text{act}_1), t_1), \ldots, e_N = (n_N, r_N, t_N)$. Given a firewall represented by a FirewallMap $\mathcal{M}$, our goal is to find a new FirewallMap $\mathcal{M}'$ which ensures all the examples are satisfied, but that guarantees all non-described packets maintain the same behavior. We also want $\mathcal{M}'$ to be well organized, meaning that "similar rules" all appear together. Our procedure (omitted due to space restrictions) to decide the similarity assigns a score based on the number and kinds of criteria used in the rules, but could be replaced by any desired scoring algorithm.

Consider the $k^{th}$ example, $1 \leq k \leq N$. We express the desired condition with respect to example $e_k$ by instantiating $k$ in Formulas 7.4.5 and 7.4.5. We then
input: $E$, the list of examples; $\mathcal{M}$, a FirewallMap
output: a FirewallMap with a rule for each $e \in E$ added

\[
\text{foreach (n, newR, t) } \in E \text{ do}
\]
\[
a' \leftarrow \text{ACL id of an arbitrary representation of the ACL } n \text{ in } \mathcal{M};
res \leftarrow \text{SAT ;}
maxR \leftarrow \text{acl_length}(a') - 1;
\text{while res = SAT do}
\]
\[
\text{Pick } r' \leq \text{maxR } , \text{ using a similarity measure to } newR;
\mathcal{M}' \leftarrow \text{insertRule}(\mathcal{M}, newR, a', r') ;
res \leftarrow \text{SMTCheckCorrectness}(\mathcal{M}, \mathcal{M}', e);
\text{if } res = \text{SAT} \text{ then}
\]
\[
\text{maxR } \leftarrow r' - 1;
\text{end}
\text{end}
\mathcal{M} \leftarrow \mathcal{M}'
\text{end}
\]

Algorithm 2: Rule Adding Repair Algorithm

show that Algorithm 2 outputs a firewall which satisfies this condition. For each example $e_i = (n_i, r_i, t_i)$, we take some $a' \in \text{equivalent}(\mathcal{M}, n_i)$ and find the ID $r'$ of the existing rule most similar to $r_i$ in ACL $a'$. Next we set $\mathcal{M}' = \mathcal{M}$, and run $\text{insert\_rule}(\mathcal{M}', r_i, a', r')$ to insert $r_i$ in all ACLs equivalent to $a'$ at position $r'$ in $\mathcal{M}'$.

We convert both $\mathcal{M}$ and $\mathcal{M}'$ to SMT formulas, and use an SMT solver to check that Formulas 7.4.5 and 7.4.5 are valid. To do this, we must eliminate the two universal quantifiers that remain after instantiating $k$. There are only a finite number of values that $a$ may attain - namely, it can only be the values in $\text{equivalent\_to\_name}(\mathcal{M}, a')$. Using this observation, we can easily eliminate the universal quantifier using finite instantiation. Once the formula is only universally quantified by $p$, we negate it, and try to show that its negation is unsatisfiable.

If the SMT solver does find the formula to be unsatisfiable, we know that the original formula was valid, i.e. the firewall satisfies the considered example. However, if the formula is satisfiable, we search for a different place to insert the rule, that comes before rule $r'$ in ACL $a'$. We do not consider any rule after this rule, as any route along which $\mathcal{M}$ and $\mathcal{M}'$ could incorrectly diverge would also exist if the new rule was inserted after $a'$. Also note that the condition is guaranteed to hold if the new rule is inserted as rule 0 in ACL $a'$; and although this placement is often not ideal for the structure of the firewall, it does guarantee termination.

When rules are from consistent examples, we can insert them in any order. By definition, two consistent examples cannot describe any of the same packets, so it does not matter which corresponding rule comes before the other in the firewall. However, the rules found by Algorithm 1 are rule inconsistent. In this case, insertion of the rules must be done in reverse order of the corresponding example’s times. This ensures that the inconsistent rules have the same relative order in $E'$ (from Sec. 7.4.6)
as in $\mathcal{M}'$, and thus we can expect the same behavior from the examples in both $E'$ and $\mathcal{M}'$.

### 7.4.8 Redundant Rule Detection

The final step in repairing the firewall is removing *redundant* rules – that is, rules which cannot be matched by any packet. An existing approach to redundant rule detection [207] can be adapted to and implemented in our SMT model. We briefly summarize this approach here.

As before, the firewall is converted to an SMT formula. Then, for each ACL name and rule ID, $n$ and $r$, respectively, check that there exists a packet that matches the rule, or some equivalent rule by asserting

$$\exists a'. a' \in \text{equivalent}(\mathcal{M}, n) \land \text{matches\_rule}(\mathcal{M}, p, a', r)$$

If this call returns SAT, then clearly there exists some packet that matches the rule, and the rule is therefore not redundant. If it returns UNSAT, then there was no packet that matched the rule, and it is therefore redundant. In this case, it can be commented out. This does involve a large number of calls to the SMT solver, but these calls tend to be fast.

### 7.5 Implementation and Evaluation

FireMason is developed in Haskell and fully implements the design described in Sec. 7.4. The default firewall language that we support is the iptables language [6], but the framework can be easily extended to other firewall languages, such as Juniper [113] and Cisco firewalls [57]. The syntax of these languages varies, but the semantics are largely the same. Therefore, only the translation step (essentially a parser) needs to be rewritten for a particular language, which means that FireMason can easily be adapted to repair firewalls written in other languages. As an SMT solver we used Microsoft’s Z3 [64]. The source code for our implementation is available at https://github.com/BillHallahan/FireMason.

The evaluation was conducted with an Intel Xeon Quad Core HT 3.7 GHz.

**Scalability Evaluation.** We first evaluated the scalability of FireMason with regard to real-world network sizes by using three examples as specification, and varying the number of rules in the target firewall between 100 and 500. These firewalls were randomly generated. As shown in Figure 7.5, FireMason scales well to large-scale firewalls.

One might expect the rate limiting rules insertion to be slower than the non rate limiting rules insertion, due to the additional runtime of Algorithm 1. However, Algorithm 1’s runtime depends only on the number of examples, and not on the number of rules in the original firewall, its runtime is constant across the rate limiting
tests. In the rate limiting case our three examples result in only two rules to insert, whereas in the non rate limiting case, we insert three rules. Thus, the additional runtime is due to Algorithm 2.

We also evaluated the performance of FireMason for different numbers of provided examples, as shown in Table 7.7. In the stateless case this scales linearly. In the rate limiting case, the time required increases rather sharply as the number of examples generating a single limit increases. However, this is not a major concern, as we have found that a small number of examples is typically sufficient to find an appropriate limit.

**Case Study: Repairing Real-World Firewalls.** We next demonstrate that FireMason can repair real-world firewalls. To do that, we found firewall repair problems on Server Fault [11] and Stack Overflow [12]. We recreated each scenario, and generated corrected firewalls using FireMason.

Tables 7.4, 7.5, and 7.6 present ten such problems. We list the examples which an administrator may provide to clarify how the firewall should be repaired and present the resulting repairs to the firewall. We also include the running time, the number of calls to the SMT solver, and the number of rules in the original iptables script.

We manually checked the correctness of each result and compared them to the repairs suggested on the forums. We found that the output returned by FireMason not only fixed the problems, but also avoided any side effects. Furthermore, we manually confirmed the “minimality” of the repairs, in terms of the impact on the firewalls overall behavior. In some cases, FireMason outputs a different solution from the posted solution. After manual comparison, we found that both solutions work correctly, but FireMason’s output required adding fewer new rules.

Interestingly, two of the case studies involving rate limits took significantly longer than those only involving stateless examples. This is not at odds with the results of the scalability evaluation. As shown in table 7.7, for a small number of examples, rate limit rule generation is generally faster, whereas for a larger number of examples, stateless rule generation is faster.
Table 7.4: Case study: Sampled stateless firewall repair problems and our solutions, benchmarks 1 to 4.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Description</th>
<th>Input example</th>
<th>Results</th>
<th>Original Rule Count</th>
<th>Repair Time</th>
<th>SMT Solver calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [2]</td>
<td>An administrator appended a rule <code>iptables -A INPUT -s 73.143.129.38 -j DROP</code> but can still receive packets from 73.143.129.38.</td>
<td>1. <code>repair(INPUT, source_ip = 73.143.129.38 =&gt; DROP)</code></td>
<td>Remove the appended rule, and insert a new rule <code>iptables -A INPUT -s 73.143.129.38/32 -j DROP</code> in front of an original rule <code>iptables -A INPUT -i lo -j ACCEPT</code>.</td>
<td>11</td>
<td>0.109 s</td>
<td>26</td>
</tr>
</tbody>
</table>
| 2 [3]      | An administrator wants to allow SSH access from the IP address 71.82.93.101, but does not know. | 1. `repair(INPUT, protocol = 22, source_ip = 71.82.93.101 => ACCEPT)`  
2. `repair(INPUT, protocol = 22, not source_ip = 71.82.93.101 => DROP)` | Insert new rules `iptables -I INPUT 0 -p 22 -s 71.82.93.101/32 -j ACCEPT` and `iptables -I INPUT 0 -p 22 ! -s 71.82.93.101/32 -j DROP` in front of an original rule `iptables -I INPUT -p icmp –icmp-type time-exceeded -j ACCEPT`. | 11 | 0.088 s | 23 |
| 3 [7]      | An administrator has the IP address 192.168.1.99, and wants to SSH to the IP address 192.168.1.15. She appended a rule `iptables -A INPUT -p tcp -i eth0 -dport 22 -m state –state NEW,ESTABLISHED -j ACCEPT` but still cannot SSH 192.168.1.15. | 1. `repair(OUTPUT, destination_ip = 192.168.1.15 => ACCEPT)`  
2. `repair(INPUT, source_ip = 192.168.1.15 => ACCEPT)` | Insert two new rules `iptables -A INPUT -s 192.168.1.15/32 -j ACCEPT` and `iptables -A OUTPUT -d 192.168.1.15/32` in front of the fourth and fifth rules in the original firewall, respectively. | 4 | 0.054 s | 14 |
| 4 [8]      | An administrator wants to allow only the localhost to have access to a given port, but is having trouble figuring out the right iptables commands. | 1. `repair(INPUT, destination_port = 44344 => ACCEPT onlyif destination_ip = 127.0.0.1)` | Inserted four new rules `iptables -A INPUT -p 17 -dport 44344 -d 127.0.0.1/32 -j ACCEPT`, `iptables -A INPUT -p 6 -dport 44344 -d 127.0.0.1/32 -j ACCEPT`, `iptables -A INPUT -p 17 -dport 44344 ! -d 127.0.0.1/32 -j DROP`, and `iptables -A INPUT -p 6 -dport 44344 ! -d 127.0.0.1/32 -j DROP` in the firewall. | 6 | 0.204 s | 14 |
Table 7.5: Case study: Sampled stateless firewall repair problems and our solutions, benchmarks 5 to 7.

<table>
<thead>
<tr>
<th>Case Study 5 [5]</th>
<th>An administrator wants to prevent all other users from using HTTP or HTTPS connections.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input example</td>
<td>1. repair(INPUT, protocol = 6, destination_port = 80 =&gt; DROP unless source_ip = 10.1.1.2)</td>
</tr>
<tr>
<td></td>
<td>2. repair(INPUT, protocol = 6, destination_port = 443 =&gt; DROP unless source_ip = 10.1.1.2)</td>
</tr>
<tr>
<td>Original Rule Count</td>
<td>6</td>
</tr>
<tr>
<td>Repair Time</td>
<td>.246 s</td>
</tr>
<tr>
<td>SMT Solver calls</td>
<td>13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case Study 6 [13]</th>
<th>An administrator wants to accept connections on a range of ports, but does not know how to do so.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input example</td>
<td>1. repair(INPUT, protocol = 17, 1000 &lt;= destination_port &lt;= 2000 =&gt; ACCEPT)</td>
</tr>
<tr>
<td>Results</td>
<td><code>iptables -A INPUT -p 17 -dport 1000:2000 -j ACCEPT</code></td>
</tr>
<tr>
<td>Original Rule Count</td>
<td>6</td>
</tr>
<tr>
<td>Repair Time</td>
<td>.057 s</td>
</tr>
<tr>
<td>SMT Solver calls</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case Study 7 [1]</th>
<th>An administrator wants to block a range of ip addresses, rather than a specific ip address.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input example</td>
<td>1. repair(INPUT, source_ip = 116.10.191.* =&gt; DROP)</td>
</tr>
<tr>
<td>Results</td>
<td><code>iptables -A INPUT -s 116.10.191.0/24 -j DROP</code></td>
</tr>
<tr>
<td>Original Rule Count</td>
<td>6</td>
</tr>
<tr>
<td>Repair Time</td>
<td>0.106 s</td>
</tr>
<tr>
<td>SMT Solver calls</td>
<td>7</td>
</tr>
</tbody>
</table>
Table 7.6: Case study: Sampled rate limiting firewall repair problems and our solutions.

<table>
<thead>
<tr>
<th>Case Study 8 [4]</th>
<th>An administrator is trying to limit the number of inbound SSH packets, but it just seems to lock her out.</th>
</tr>
</thead>
</table>
| Input examples   | 1. repair(INPUT, protocol = 22, time = 0 => ACCEPT)  
|                  | 2. repair(INPUT, protocol = 22, time = 20 => ACCEPT)  
|                  | 3. repair(INPUT, protocol = 22, time = 30 => ACCEPT)  
|                  | (In total, this repair uses 8 examples.) |
| Results          | Insert new rules iptables -A INPUT -m limit -limit 2/minute -limit-burst 4 -p 22 -j ACCEPT and iptables -A INPUT -p 22 -j DROP at the beginning of the original firewall. |
| Original Rule Count | 9                                                                                           |
| Repair Time      | 21.10 s                                                                                     |
| SMT Solver calls | 44                                                                                           |
| Case Study 9 [9] | A server is attacked by TCP SYN flooding, so the administrator wants a limit on SYN packets per second. |
| Input examples   | 1. repair(INPUT : source_ip = 192.132.209.0/24, SYN, time = 10  
|                  | => ACCEPT)  
|                  | 2. repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 11  
|                  | => ACCEPT)  
|                  | 3. repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 12  
|                  | => ACCEPT)  
|                  | 4. repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 13  
|                  | => DROP)  
|                  | 5. repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 19  
|                  | => DROP)  
|                  | 6. repair(INPUT, source_ip = 192.132.209.0/24, SYN, time = 21  
|                  | => ACCEPT) |
| Results          | Append two new rules, iptables -I INPUT 0 -s 192.132.209.0/24 -p 6 -tcp-flags SYN -j DROP and iptables -I INPUT 0 -m limit -limit 6/minute -limit-burst 3 -s 192.132.209.0/24 -p 6 -tcp-flags SYN -j ACCEPT, to the original firewall. |
| Original Rule Count | 11                                                                                       |
| Repair Time      | 0.046 s                                                                                     |
| SMT Solver calls | 42                                                                                           |
| Case Study 10 [10] | An administrator wants to rate limit the number of new TCP connections to there server. |
| Input examples   | 1. repair(INPUT, protocol = 6, destination_port = 22, SYN,  
|                  | time = 0 => ACCEPT)  
|                  | 2. repair(INPUT, protocol = 6, destination_port = 22, SYN,  
|                  | time = 0 => ACCEPT)  
|                  | 3. repair(INPUT, protocol = 6, destination_port = 22, SYN,  
|                  | time = 0 => ACCEPT)  
|                  | 4. repair(INPUT, protocol = 6, destination_port = 22, SYN,  
|                  | time = 1 => DROP) |
| Results          | Append two new rules, iptables -A INPUT -m limit -limit 54/minute -limit-burst 3 -p 6 -dport 22 -tcp-flags SYN SYN -j ACCEPT and iptables -A INPUT -p 6 -dport 22 -tcp-flags SYN SYN -j DROP, to the original firewall. |
| Original Rule Count | 6                                                                                       |
| Repair Time      | 0.509 s                                                                                     |
| SMT Solver calls | 10                                                                                           |
Table 7.7: Scalability for number of examples (when inserting into a firewall with 100 rules).

<table>
<thead>
<tr>
<th>Number of examples</th>
<th>Stateless Time (s)</th>
<th>Rate Limiting Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3.567</td>
<td>2.177</td>
</tr>
<tr>
<td>6</td>
<td>4.545</td>
<td>2.004</td>
</tr>
<tr>
<td>9</td>
<td>5.804</td>
<td>36.37</td>
</tr>
</tbody>
</table>

7.6 Related Work

This section presents existing efforts on firewall analysis, verification and generation, and discusses why these efforts are not helpful to our target.

Firewall Repair and Synthesis. Chen et al. [52] describes an approach to repair stateless firewalls. The paper develops techniques to localize specific forms of faulty rules, as opposed to our approach of building a general model. Unlike our approach, rate-limiting rules are not considered.

Zhang et al. [207] proposed a symbolic firewall synthesis approach such that the synthesized firewall has the same behavior as a given firewall, but with the smallest possible number of rules. As this approach focuses on automatically simplifying redundant rules, rather than repairing an observed error, it is not applicable to our goal.

As software defined networks (SDN) have become increasingly popular, automatic programming approaches for SDN have been proposed [163, 206]. Yuan et al. [206] proposed an automatic SDN policy generation approach, named NetEgg, based on a scenario-based programming technique. NetEgg can only generate a new policy, it cannot account for the effect of a new policy on existing policies in the network. Furthermore, NetEgg cannot synthesize rate limiting rules.

Firewall Analysis and Verification. Mayer et al. [136] developed the first systematic firewall analysis engine, Fang, to analyze diverse properties of firewalls. Fang and its sequel Lumeta [200] allow checking the correctness of firewall configurations by sending their analysis engines queries. Other efforts [76, 19] propose packet-filter based schemes to detect conflicting or violated rules. Frantzen et al. [83] and Kamara et al. [116] proposed different data-flow based approaches to analyze vulnerability risks in firewalls. Yuan et al. [205] used BDDs to detect policy violations and misconfigurations in firewalls. Wool [201] conducted a case study on understanding and classifying the configuration errors of firewalls.

The Margrave firewall verification tool [142] encodes firewall rules and queries into first-order logic. It uses KodKod [192] to search for finite state models. Compared with another firewall verification tool, NoD [131], Margrave cannot produce all differences between policies in a compact way, and does not scale for large firewall rule sets.

Firewall Testing. El-Atawy et al. [74] proposed targeting test packets for better fault coverage. Al-Shaer et al. [21] developed a system-wide framework to generate
targeted packets and obtain good coverage during firewall testing. Brucker et al. [44] proposed a formal firewall conformance testing approach, which uses Isabelle/HOL to generate test-cases from constraint satisfaction problems.

7.7 Conclusion

In this chapter, we have presented FireMason, a tool for verification and repair of firewalls. To this end, we use a first-order intermediary language to model firewalls, which allows us use of an SMT solver to obtain formal guarantees on the correctness of verification and repair. We showed that FireMason not only generates correctly repairs real-world firewall scripts, but also is able to scale to large-scale firewalls. Our empirical evaluation suggests that FireMason could be both practical and effective in assisting administrators with firewall management. Our goal is to inspire further work on reasoning about firewalls in the formal methods community.
Chapter 8

Conclusion

We have explored how the use of analysis, verification, and synthesis techniques can be eased, automated, and scaled. We have examined a variety of use cases, ranging from debugging and automation of modular verification, repair and analysis of firewalls, and scalability of synthesizers.

By augmenting lazy semantics with symbolic variables and the definition of symbolic weak head normal form, we introduced the first symbolic execution engine targeting non-strict functional languages, such as Haskell. Using symbolic execution as a backend technique enables a variety of applications. We developed a library, G2Q, that enables easier access to constraint solvers by directly writing Haskell code. We also introduced concrete and abstract counterexamples, and the counterfactual symbolic execution technique that allows finding both types of counterexamples to ease debugging of modular verifiers. Finally, we introduced an inference technique to automatically generate specifications enabling modular verification. This technique builds on counterfactual symbolic execution by using concrete and abstract counterexamples to guide a specification synthesizer.

We also describe a machine learning based technique to improve the efficiency of SyGuS solvers by filtering the grammar. This allows the solver to more quickly solve challenging problems, and scale to problems that were previously out of reach. We demonstrated the effectiveness of this technique on benchmarks from SyGuS-Comp.

Finally, we explore the use of automated repair and analysis in the domain of firewalls, via the development of FireMason. FireMason relies on a formal semantics defined for the iptables language via a translation to first order logic with uninterpreted functions and linear integer arithmetic. This allows encoding of firewalls into first order logic, enabling use of an SMT solver for analysis and sketch-based repair.
8.1 Future Directions

8.1.1 Automated verification

*Modular Verifiers* Chapter 5 explores techniques to automatically infer specifications that are needed for the purpose of modular verification. Further, it establishes a complete method to infer needed specifications, given an underlying synthesizer with specific properties—namely the synthesizer must be a complete, size-bounded, interpolant producing synthesizer. A technique to design such a synthesizer for linear integer arithmetic specifications is also described in Chapter 5. Making this technique applicable to a wider variety of programs requires developing synthesizers capable of producing a wider variety (or at least, different variety) of specifications. As briefly discussed in Chapter 5, some preliminary work has already begun experimenting with specifications over sets, however it is not yet as performant as one would like. Synthesizing specifications over strings and bit vectors also seem like clear targets.

An obvious approach to synthesizing specifications in a variety of domains is to make use of SyGuS solvers. However this is not without its challenges. The solving of SyGuS problems with limitations on the size of the solution, as is required for a size-bounded synthesizer, is challenging. Existing work on this topic [105] is successful on only 15 out of 26 evaluated benchmarks. Similarly, state of the art techniques to determine unrealizability of SyGuS problems [103, 104] are currently only able to successfully prove unrealizability on slightly over half of benchmarks. Further, existing techniques do not produce the interpolants required by our algorithm. Since automated specification inferences requires many synthesis calls, each of which must succeed, practical application of SyGuS solvers will depend on advances in SyGuS solvers capabilities.

*Fancy Types* Long term, I envision investigating techniques to automate verification mechanisms besides modular verifiers. Advanced typing features, such as GADTs and type families, allow programmers to specify and prove properties of their code. For example, such features allow checking at compile time that matrix operations are applied to matrices of appropriate dimensions [175] and that abstract syntax trees in an interpreter or compiler represent only correctly typed programs [72]. Unfortunately, adding additional safety properties via types is not a trivial process, especially if the existing code base is large. Further, writing new code that passes the typechecker can be tricky—in a mailing list discussion, this concern was cited by Simon Peyton Jones as a reason to not use stronger types in the internals of a widely used Haskell compiler, GHC [159]. Thus, the development of techniques to automatically- or at least, semiautomatically- update programs with stronger types would aid both new and experienced programmers.

*DeepSpec* The DeepSpec project [31] proposes the use of *deep specifications* to verify programs. Specifications made available by modular verifiers such as LiquidHaskell capture properties of the code, such as the size or unordered contents of lists. In contrast, deep specifications are typically precise enough to ensure that any two programs
satisfying the specification are contextually equivalent— that is, one program can be seamlessly replaced with the other in any context [89]. Proving deep specifications is typically done using a manual theorem prover such as Coq. Exploring techniques to automatically generate such proofs (or sections of such proofs) in a counterexample guided way would be an interesting direction.

8.1.2 Formal Methods via Machine Learning

The Grammar Reduction Tool (GRT), as discussed in Chapter 6, explores how machine learning techniques could be helpful to formal methods tools. Many formal methods problems are fundamentally about searching large spaces. In synthesis, one is searching over a large space of programs for a particular program that satisfies a specification. In symbolic execution, one is searching over a large number of possible states, for a particular state that corresponds to a counterexample. Using machine learning techniques to help direct such searches is a natural direction to explore.

Such techniques would also benefit from a deeper integration with the logic driven techniques traditionally used in formal methods. Every time the GRT is run, a risk is taken— if the grammar is filtered too much, the synthesis problem will become unrealizable. Integrating the predictions into the solver and using them to guide which solutions to search through first— without ruling out other solutions entirely— would eliminate this risk. Of course, this is likely more engineering effort, but the fact that even the somewhat crude application of machine learning techniques in the GRT produced positive results is an indication that the added effort is worth it.

8.2 Remarks

Wide spread adoption of formal methods techniques requires that they be accessible to mainstream programmers— not only domain experts. In this dissertation, we have explored how formalized language semantics, used in combination with SMT solvers, enable us to automatically analyze or repair programs.
Bibliography


[57] Cisco. Policing and Shaping Overview.


## Contents

A Proof of Mathematical Theorems 154

A.1 Appendix .................................................. 154
  A.1.1 verify properties ..................................... 154
  A.1.2 symex properties ...................................... 154
  A.1.3 Proofs .................................................. 155
Appendix A

Proof of Mathematical Theorems

A.1 Appendix

A.1.1 verify properties

If a verifier is modular, whether \( f \) is in an error set \( \text{err} \) is determined entirely by the specification of \( f \), and the specification of functions called by \( f \):

\[
\forall P, E, E', \text{err}, f. \\
\text{verify}(P, E) = \text{Error err} \\
\land f \in \text{err} \\
\land \text{lookup}(f, E) = \text{lookup}(f, E') \\
\land \forall g \in \text{calls}(f). \text{lookup}(g, E) = \text{lookup}(g, E') \\
\implies \exists \text{err}'. \text{verify}(P, E') = \text{Error err'} \\
\land f \in \text{err'}
\]

Similarly, when using a modular verifier, if the environment is held steady, but aspects of the program besides the definition of \( f \) change has no affect on whether \( f \) is in a returned error set:

\[
\forall P, P', E, \text{err}, f, f'. \\
\text{verify}(P, E) = \text{Error err} \\
\land f \in \text{err} \\
\land \text{lookup}(f, P) = \text{lookup}(f, P') \\
\implies \exists \text{err}'. \text{verify}(P', E) = \text{Error err'} \\
\land f \in \text{err'}
\]

A.1.2 symex properties

A symex function is complete if the following two conditions are met:
• whenever a set of specifications allows for a concrete postcondition to a function or precondition counterexample to a pair of functions, there is some depth such that a concrete counterexample for that function or pair of functions is in the returned set.

\[
\forall P, E, f, s_f. \exists i. s_f = \text{lookup}(f, E) \land \neg s_f(i, f(i)) \\
\implies \exists \text{cexs}, d, i'. \text{symex}(P, d, E, f) = \text{cexs} \land (f, i')^\text{post} \in \text{cexs}
\]

\[
\forall P, E, f, s_f, i_f, g, s_g, i_g. \exists i_f, i_g.
\]

\[
s_f = \text{lookup}(f, E) \land g = \text{lookup}(g, E) \\
\land s_f^{\text{pre}}(i_f) \land \neg s_g^{\text{pre}}(i_g) \land \text{evals}(f, i_f, g, i_g)
\implies \exists \text{cexs}, d, i_f, i_g. \text{symex}(P, d, E, f) = \text{cexs} \\
\land (f, i_f, g, i_g)^\text{pre} \in \text{cexs}
\]

• when \(\text{verify}(P, E) = \text{Error } err\), and \(f \in err\), there is some depth such that \(\text{symex}\) returns a non-empty counterexample set:

\[
\forall P, E. \text{verify}(P, E) = \text{Error } err \iff \forall f \in err. \exists d, \text{cex}. |\text{symex}(P, d, E, f)| \geq 1
\]

If \(\text{symex}\) returns a set containing a counterexample, increasing the depth results in it returning a set with that same counterexample:

\[
\forall P, d, d', E, f, \text{cexs}. \text{symex}(P, d, E, f) = \text{cexs} \\
\land \text{cex} \in \text{cexs} \land d < d' \\
\implies \exists \text{cexs}'. \text{symex}(P, d', E, f) = \text{cexs}' \\
\land \text{cex} \in \text{cexs}'
\]

If when called with environments \(E\) and \(E'\) \(\text{symex}\) returns sets of counterexamples \(\text{cexs}\) and \(\text{cexs}'\), respectively, calling \(\text{symex}\) on the union of those two environments will return all counterexamples in \(\text{cexs}\) and \(\text{cexs}'\) that still apply:

\[
\forall P, d, E, E', f, \text{cexs}, \text{cex}. \\
\text{symex}(P, d, E, f) = \text{cexs} \\
\land \text{cex} \in \text{cexs} \land \text{cex} \not\in E \cup E' \\
\implies \exists \text{cexs}'. \text{symex}(P, d, E \cup E', f) = \text{cexs}' \land \text{cex} \in \text{cexs}'
\]

A.1.3 Proofs

We prove the following lemma about \(\text{verify}\):
Lemma 1.

\[ \forall P, P', E, E', err, f, f'. \]
\[ \text{verify}(P, E) = \text{Error} \ err \]
\[ \land f \in err \]
\[ \land \text{lookup}(f, P) = \text{lookup}(f', P') \]
\[ \land \forall g \in \text{calls}(f). \text{lookup}(g, P) = \text{lookup}(g, P') \]
\[ \land \text{lookup}(f, E) = \text{lookup}(f', E') \]
\[ \land \forall g \in \text{calls}(f). \text{lookup}(g, E) = \text{lookup}(g, E') \]
\[ \implies \exists err'. \text{verify}(P, E') = \text{Error} \ err' \]
\[ \land f' \in err' \]

Proof. We assume the left hand side of the implication holds.

By verify’s property A.1.1, we know there is some err' such that verify(P', E) = Error err' and f' \in err'. Then, by verify’s property A.1.1, we know there is some err'' such that f' \in err''.

We prove the following lemma’s about symex:

Lemma 2. If symex is sound and complete then

\[ \forall P, E, E', f. \]
\[ |\text{symex}(P, 1, E, f)| \geq 1 \]
\[ \land \text{lookup}(f, E) = \text{lookup}(f, E') \]
\[ \land \forall g \in \text{calls}(f). \text{lookup}(g, E) = \text{lookup}(g, E') \]
\[ \implies |\text{symex}(P, 1, E', f)| \geq 1 \]

Proof. We assume the left hand side of the implication.

Since |symex(P, 1, E, f)| \geq 1 and since symex is complete there exists some err such that verify(P, E) = Error err and f \in err. Then by verify’s property A.1.1, there is some err’ such that verify(P, E') = Error err' and f \in err'. Then again by symex’s completeness we have |symex(P, 1, E', f)| \geq 1, so the lemma is satisfied.
**Lemma 3.** If $\text{symex}$ is sound then

\[
\forall P, d, E, E', f, s, cex, cexs, cexs'. \\
\text{s}_f = \text{lookup}(f, E) \\
\land f \notin E' \\
\land \text{symex}(P, d, E, f) = cexs \\
\land cex \in cexs \land \text{isConcrete}(cex) \\
\implies \text{symex}(P, d, E \cup E', f) = cexs'
\]

**Proof.** We consider the case where $cex$ is a precondition counterexample, and where $cex$ is a postcondition counterexample.

- $cex = (f, i, g, i_g)_{C}^{\text{pre}}$: Since $f \notin E'$, we have $s_f = \text{lookup}(f, E) = \text{lookup}(f, E \cup E')$. By the definition of union of environments, if $s_g = \text{lookup}(g, E)$ and $s'_g = \text{lookup}(g, E')$, and $s_g^\cup = \text{lookup}(g, E \cup E')$, then $s_g^{\text{pre}}(i) = s_g^\cup(i) \land s_g^{\text{pre}}(i)$. Since $\text{symex}(P, d, E, f) = cexs$ and $(f, i, g, i_g)_{C}^{\text{pre}} \in cexs$, by the definition of a concrete precondition counterexample and the soundness of $\text{symex} s_f^{\text{pre}}(i_f) \land \neg s_g^{\text{pre}}(i_g)$. Thus, we have that $s_f^{\text{pre}}(i) \land \neg(s_g^{\text{pre}}(i) \land s_g^{\text{pre}}(i))$. Then, by $\text{symex}$'s property A.1.2, $(f, i, g, i_g)_{C}^{\text{pre}} \in cexs'$.

- $cex = (f, i)_{C}^{\text{post}}$: Since $f \notin E'$, we have $s_f = \text{lookup}(f, E) = \text{lookup}(f, E \cup E')$. Since $\text{symex}(P, d, E, f) = cexs$ and $(f, i)_{C}^{\text{post}} \in cexs$, by the definition of a concrete postcondition counterexample and the soundness of $\text{symex} s_f^{\text{pre}}(i) \land \neg s_g^{\text{pre}}(i)$. Then, by $\text{symex}$’s property A.1.2, $(f, i)_{C}^{\text{post}} \in cexs'$.

\[\blacksquare\]

**Lemma 4.**

\[
\forall d, P, E_U, E_S, f, C. cexs = \text{symex}(P, d, E_U \cup E_S, f) \\
\land \text{Right } C = \text{evalCE}(E_U, cexs) \\
\implies \neg \text{iSat}_{\text{abs}}(E_U, E_S, C)
\]

**Proof.** We assume the left hand side of the implication, and show it implies the right.

- Consider $cex = (f, i, o, F_{\text{abs}})_{A}^{\text{post}}$. $\text{evalCE}$ returns a constraint $c$:

\[
s_f^{\text{pre}}(i) \land \neg s_f^{\text{post}}(i, o) \implies \bigvee_{(g, i_g, o_g) \in F_{\text{abs}}} (s_g^{\text{pre}}(i_g) \implies \neg s_g^{\text{post}}(i_g, o_g))
\]

By the definition of an abstract postcondition counterexample, in $E_U \cup E_S$ we have $s_f^{\text{pre}}(i) \land \neg s_f^{\text{post}}(i, o)$ and $\forall (g, i, o) \in F_{\text{abs}}, s_g = \text{lookup}(g, E_U \cup E_S) \implies s_g(i, o)$. Thus, when instantiated with $E_U \cup E_S$, the left hand side of $c$ is true, but the right hand side is false. Thus $c \in C \implies \neg \text{iSat}_{\text{abs}}(E_U, E_S, C)$.  

157
Consider $cex = (f, i_f, g, i_g, F_{abs})_A$. evalCE returns a constraint $c$:

$$s^\text{pre}_f(i_f) \land \neg s^\text{pre}_g(i_g) \implies \bigvee_{(h, i_h, o_h) \in F_{abs}} (s^\text{pre}_h(i_h) \implies \neg s^\text{post}_h(i_h, o_h))$$

By the definition of an abstract precondition counterexample in $E_U \cup E_S$ we have $s^\text{pre}_f(i_f) \land \neg s^\text{pre}_g(i_g)$ and $\forall (h, i, o) \in F_{abs}$. $s_h = \text{lookup}(h, E_U \cup E_S) \implies s_h(i, o)$. Thus, when instantiated with $E_U \cup E_S$, the left hand side of $c$ is true, but the right hand side is false. Thus $c \in C \implies \neg \text{isSat}_{UF}(E_U, E_S, C)$.

Consider $cex = (f, i)_C^\text{post}$. In the case where evalCE returns a constraint, that constraint is $c = s_f(i, f(i)) = s^\text{pre}_f(i) \implies s^\text{post}_f(i, o)$. By the definition of a concrete postcondition counterexample, in $E_U \cup E_S$ we have $s^\text{pre}_f(i) \land \neg s^\text{post}_f(i, o)$. Thus $c \in C \implies \neg \text{isSat}_{UF}(E_U, E_S, C)$.

Consider $cex = (f, i_f, g, i_g)_C^\text{pre}$. There are two cases where evalCE returns a constraint.

In one case, the constraint is $c = s^\text{pre}_g(i_g)$. By definition of a precondition counterexample in $E_U \cup E_S$ we have $\neg s^\text{pre}_g(i_g)$. Thus $c \in C \implies E_U \cup E_S \not\models C$.

In the second case, the constraint is $c = s^\text{pre}_f(i_f) \implies s^\text{pre}_g(i_g)$. By the definition of a precondition counterexample, we have $s^\text{pre}_f(i_f) \land \neg s^\text{pre}_g(i_g)$. Thus $c \in C \implies \neg \text{isSat}_{UF}(E_U, E_S, C)$.

\[\Box\]

**Corollary 1.**

$$\forall P, d, E_U, E_S, E'_S, f, C. cexs = \text{symex}(P, d, E_U \cup E_S, f) \wedge \text{Right } C = \text{evalCE}(E_U, cexs) \wedge \exists (f, i^\text{post}_C) \in cexs \wedge \text{lookup}(f, E_S) = \text{lookup}(f, E'_S) \implies E_U \cup E'_S \not\models C$$

**Corollary 2.**

$$\forall P, d, E_U, E_S, E'_S, f, g, i, i_g, C. cexs = \text{symex}(P, d, E_U \cup E_S, f) \wedge \text{Right } C = \text{evalCE}(E_U, cexs) \wedge \exists (f, i_g, i^\text{pre}_C) \in cexs \wedge \text{lookup}(f, E_S) = \text{lookup}(f, E'_S) \wedge \text{lookup}(g, E_S) = \text{lookup}(g, E'_S) \implies E_U \cup E'_S \not\models C$$
Lemma 5. Given a sound verify and some theory $\mathcal{T}$,

$$\forall P, d, E_U, E_S^R, f, E_S^G \subset \mathcal{T}. cex = symex(P, d, E_U \cup E_S^R, f)$$

$$\land \text{Verified} = \text{verify}(P, E_U \cup E_S^G)$$

$$\land \text{Right } C = \text{evalCE}(E_U, cexs)$$

$$\implies isSat_{\mathcal{T}}(E_U, E_S^G, C)$$

Proof. We assume the left hand side of the implication, and show it implies the right.
We consider each $c \in C$, and in each case show that $\neg isSat_{\mathcal{T}}(E_U, E_S^G, \{c\})$ leads to a contradiction.

- Consider $cex = (f, s_f, i, o)^{\text{post}}_A F^{\text{abs}}$. evalCE returns a constraint $c$:

$$s_f^{\text{pre}}(i) \land \neg s_f^{\text{post}}(i, o) \implies \bigvee_{(g, i_g, o_g) \in F^{\text{abs}}} (s_g^{\text{pre}}(i_g) \implies \neg s_g^{\text{post}}(i_g, o_g))$$

Suppose $\neg isSat_{\mathcal{T}}(E_U, E_S^G, \{c\})$. Then, we must have:

$$isSat_{\mathcal{T}}(E_U, E_S^G, \{s_f^{\text{pre}}(i) \land \neg s_f^{\text{post}}(i, o) \land \bigwedge_{(g, i_g, o_g) \in F^{\text{abs}}} (s_g^{\text{pre}}(i_g) \land s_g^{\text{post}}(i_g, o_g))\})$$

But then $(f, i, o, F^{\text{abs}})^{\text{post}}_A \notin E_U \cup E_S^G$, so $\text{Verified} = \text{verify}(P, E_U \cup E_S^G)$ contradicts the soundness of verify. Thus we have a contradiction, and it must be that $isSat_{\mathcal{T}}(E_U, E_S^G, \{c\})$.

- Consider $cex = (f, i, g, i_g, F^{\text{abs}})^{\text{pre}}_A$. evalCE returns a constraint $c$:

$$s_f^{\text{pre}}(i_g) \land \neg s_g^{\text{pre}}(i_g) \implies \bigvee_{(h, i_h, o_h) \in F^{\text{abs}}} (s_h^{\text{pre}}(i_h) \implies \neg s_h^{\text{post}}(i_h, o_h))$$

Suppose $\neg isSat_{\mathcal{T}}(E_U, E_S^G, \{c\})$. Then, we must have:

$$isSat_{\mathcal{T}}(E_U, E_S^G, \{s_f^{\text{pre}}(i_g) \land \neg s_g^{\text{pre}}(i_g) \land \bigwedge_{(h, i_h, o_h) \in F^{\text{abs}}} (s_h^{\text{pre}}(i_h) \land s_h^{\text{post}}(i_h, o_h))\})$$

But then $(f, i, g, i_g, F^{\text{abs}})^{\text{pre}}_A \notin E_U \cup E_S^G$, so $\text{Verified} = \text{verify}(P, E_U \cup E_S^G)$ contradicts the soundness of verify. Thus we have a contradiction, and it must be that $isSat_{\mathcal{T}}(E_U, E_S^G, \{c\})$.

- Consider $cex = (f, i)^{\text{post}}_G$. In the case where evalCE returns a constraint, that constraint is $c = s_f(i, f(i)) = s_f^{\text{pre}}(i) \implies s_f^{\text{post}}(i, o)$. Suppose that $\neg isSat_{\mathcal{T}}(E_U, E_S^G, \{c\})$. Then we must have

$$isSat_{\mathcal{T}}(E_U, E_S^G, \{s_f^{\text{pre}}(i) \land \neg s_f^{\text{post}}(i, o)\}).$$
But then \((f, i)_{C}^{\text{post}} \not\in E_{U} \cup E_{S}^{G}\) so \(\text{Verified} = \text{verify}(P, E_{U} \cup E_{S}^{G})\) contradicts the soundness of \(\text{verify}\). Thus we have a contradiction, and it must be that \(\text{isSat}_{T}(E_{U}, E_{S}^{G}, \{c\})\).

- Consider \(cex = (f, i_{f}, g, i_{g})_{C}^{\text{pre}}\). There are two cases where \(\text{evalsCE}\) returns a constraint.

  In one case, if \(f \in \text{external}(P)\), the constraint is \(c = s_{g}^{\text{pre}}(i_{g})\). Let \(s_{f}^{A} = \text{lookup}(f, E_{U})\). Because we generated the counterexample, we know that \(s_{f}^{A}(i)\), and \(\text{evals}(f, i_{f}, g, i_{g})\). Suppose \(\neg \text{isSat}_{T}(E_{U}, E_{S}^{G}, \{c\})\). Then we must have \(\text{isSat}_{T}(E_{U}, E_{S}^{G}, \neg s_{g}^{\text{pre}}(i_{g}))\). But then \((f, i_{f}, g, i_{g})_{C}^{\text{pre}} \not\in E_{U} \cup E_{S}^{G}\) so \(\text{Verified} = \text{verify}(P, E_{U} \cup E_{S}^{G})\) contradicts the soundness of \(\text{verify}\). Thus we have a contradiction, and it must be that \(\text{isSat}_{T}(E_{U}, E_{S}^{G}, \{c\})\).

  In the second case, the constraint is \(c = s_{f}^{\text{pre}}(i_{f}) \implies s_{g}^{\text{pre}}(i_{g})\). Because we generated the counterexample, we know that \(\text{evals}(f, i_{f}, g, i_{g})\). Suppose \(\neg \text{isSat}_{T}(E_{U}, E_{S}^{G}, \{c\})\). Then we must have \(\text{isSat}_{T}(E_{U}, E_{S}^{G}, s_{f}^{\text{pre}}(i_{f}) \land \neg s_{g}^{\text{pre}}(i_{g})))\). But then \((f, i_{f}, g, i_{g})_{C}^{\text{pre}} \not\in E_{U} \cup E_{S}^{G}\) so \(\text{Verified} = \text{verify}(P, E_{U} \cup E_{S}^{G})\) contradicts the soundness of \(\text{verify}\). Thus we have a contradiction, and it must be that \(\text{isSat}_{T}(E_{U}, E_{S}^{G}, \{c\})\).

\(\Box\)

The following invariant holds for \(\text{generateSpec}_{T}\), when it is called from \(\text{traverseCG}_{T}\):

**Invariant 2.** \(sf = \bigcup_{f \in fs}\{g | g \in \text{calls}(f) \land g \notin E_{S}\}\)

**Proof.** Trivial, by initial definition of \(sf\) at line 6. Note that \(fs, sf,\) and \(E_{S}\) are never changed inside of \(\text{generateSpec}_{T}\). \(\Box\)

**Lemma 6.** Consider a call to \(\text{generateSpec}_{T}(P, E_{U}, fs, sf, E_{S}, C, C_{sz})\) such that Invariant 2 holds. As defined at line 19, \(E_{S}^{G}\) satisfies the following two properties:

- \(\forall f \in fs. \forall g \in \text{calls}(f). g \in E_{S}^{G}\)
- \(\forall f \in E_{S}. \text{lookup}(f, E_{S}) = \text{lookup}(f, E_{S}^{G})\)

**Proof.** By Invariant 2,

\[sf = \bigcup_{f \in fs}\{g | g \in \text{calls}(f) \land g \notin E_{S}\}.\]

Thus,

\[\forall f \in fs. g \in \text{calls}(f). g \in sf \iff g \notin E_{S}^{G}.\]

At line 17, \(\text{synth}_{T}(sf, E_{U}, E_{S}, C)\) synthesizes \(E_{N}\). By definition of \(\text{synth}\), \(\forall g. g \in sf \iff g \in E_{N}\). Thus,

\[\forall f \in fs. g \in \text{calls}(f). g \in E_{N} \iff g \notin E_{S}^{G}.\]
At line 19, $E'_S$ is defined as $E_S \cup E_N$. Thus we have the first property,

$$\forall f \in fs. g \in \text{calls}(f). g \in E_S \cup E_N = E'_S.$$ 

Since $g \in E_S \implies g \notin E_N$, we have the second property:

$$\forall f \in E_S. \text{lookup}(f, E_S) = \text{lookup}(f, E'_S).$$

We show that all further Invariants are satisfied by an initial call

$$\text{traverseCG}_T(P, E_U, fs, E_S, C, C_{sz})$$

such that $fs = \text{external}(P)$, $E_S = \{\}$, and $\forall E^G_S \subseteq \mathcal{T}.\text{verify}(P, E_U \cup E^G_S) \implies \text{isSat}_T(E_U, E^G_S, C)$ and are maintained by $\text{traverseCG}_T$ and $\text{generateSpec}_T$. Note that the calls in initInfer and in iterateInfer both satisfy all the required initial conditions.

**Invariant 3.** $\forall g \in fs. g \in \text{external}(P) \vee g \in E_S$

**Proof.** *Initialization* At line 1, $fs$ is initialized to $\text{external}(P)$, trivially satisfying the invariant.

*Maintenance* The call to $\text{traverseCG}_T$ at line 12 and the calls to $\text{generateSpec}_T$ at lines 7 and 24 trivially maintain the invariant, since $fs$ and $E_S$ are passed directly.

Now consider the call at line 11. $\text{traverseCG}_T$ is recursively called with $E'_S$, which must have come from line 19 of $\text{generateSpec}_T$. Thus, $E'_S = E_S \cup E_N$. $E_S$
trivially satisfies the invariant. $E_N$ is the result of $\text{synth}_\tau(sf,E_U,E_S,C)$, where $sf = \bigcup_{f \in fs} \{g | g \in \text{calls}(f) \land g \notin E_S\}$. Since $sf$ contains only functions called by functions in $fs$, and by definition external functions cannot be called by other functions in $P$, the invariant is maintained.

**Invariant 5.**

$$\forall f \in P. (f \in \text{external}(P) \lor f \in E_S) \land f \notin fs \implies \forall g \in \text{calls}(f). g \in E_S$$

**Proof. Initialization** At line 1, $\text{traverseCG}_T$ is called with $fs = \text{external}(P)$ and $E_S = \{\}$. Thus, the left hand side of the implication is false for all $f$, so the invariant is satisfied.

**Maintenance** The calls to $\text{traverseCG}_T$ at line 12 and to $\text{generateSpec}_T$ at lines 7 and 24 trivially maintain the invariant, since $fs$ and $E_S$ are passed directly.

Now consider the call to $\text{traverseCG}_T$ at line 11. Consider some $f'$ satisfying:

$$(f' \in \text{external}(P) \lor f' \in E'_S) \land f' \notin fs'$$

$E'_S$ must have come from line 19 of $\text{generateSpec}_T$. By construction of $E'_S$, either $f' \in E_S$ or $f' \in E_N$. Consider these two cases separately:

- $f' \in E_S$. Consider two further cases.
  - If $f' \notin fs$, then by this invariant $\forall g \in \text{calls}(f'). g \in E_S$. Thus, since $E'_S = E_S \cup E_N, \forall g \in \text{calls}(f'). g \in E'_S$.
  - If $f' \in fs$, then by lemma 6,
    $$\forall g \in \text{calls}(f'). g \in E'_S.$$  
    Thus, the invariant is satisfied.

- $f' \in E_N$. By construction of $E_N$, we know $f' \in sf$. At line 11, $\text{traverseCG}_T$ is called recursively with $fs = sf$. Thus, in the new call, $f' \in fs$. Thus, if $f' \in E_N$, the left hand side of the implication is false. Therefore, the invariant is satisfied.

Thus, in either case, the invariant is satisfied.

**Lemma 7.** Given a complete symex function, $\text{verifyCEz}(P,E,fs)$ will always terminate. If $\text{verify}(P,E) = \text{Error err}$, the returned set will have at least one counterexample per function in $err \cap fs$.

**Proof.** This is trivial if $\text{verify}$ returns $\text{Verified}$. If it returns an error set, the lemma follows from the completeness of symex.
Invariant 6.

$$verify(P, E_U \cup E_S) = Error \; err \implies \forall f \in err. (f \notin external(P) \land f \notin E_S) \lor f \in fs$$

Proof. **Initialization** Initially, $fs = external(P)$ and $E_S = \{\}$, so the invariant is trivial.

**Maintenance** The calls to $traverseCG_T$ at line 12 and to $generateSpec_T$ at lines 7 and 24 trivially maintain the invariant, since $fs$ and $E_S$ are passed directly.

Now consider the call at line 11. We proceed via proof by contradiction. Assume there is some $f' \in err$ satisfying:

$$verify(P, E_U \cup E'_S) = Error \; err'$$

$$\land (f' \in external(P) \lor f' \in E'_S)$$

$$\land f' \notin sf$$

(Note that, in the new call, $fs$ is initialized to the current $sf$.)

$E'_S$ must have come from line 19 of $generateSpec_T$. By construction of $E'_S$, either $f' \in E_S$ or $f' \in E_N$. Consider these two cases separately:

- $f' \in E_S$. Consider two further cases.
  - If $f' \notin fs$, then by Invariant 5, $\forall g \in calls(f').g \in E_S$. By lemma 6,
    $$\forall g \in calls(f').lookup(g, E_S) = lookup(g, E'_S).$$
    Then by $verify$’s property A.1.1 it must be that $verify(P, E_U \cup E_S) = Error\; err$ and $f' \in err$. However, this contradicts our invariant, since it implies that:
    $$verify(P, E_U \cup E'_S) = Error \; err' \land f' \in err \land f \in E_S \land f' \notin fs$$
    and thus our assumption must be wrong.

  - Now consider if $f' \in fs$. Then, by lemma 6, $\forall g \in calls(f').g \in E'_S$. To reach the recursive call at line 11 of $traverseCG_T$, $verifyCEx$ must return an empty set at line 20 of $generateSpec_T$, which by lemma 7 will only happen if $f' \notin err$. But this is a contradiction to our assumption, and so it must be that our assumption is wrong, and $f' \notin err$.

- $f' \in E_N \land f' \notin E_S$. In this case, it must be that $f' \in sf$, and therefore, in the call to $traverseCG$, $f' \in fs$. Thus, the right hand side of the invariant’s implications is trivially true, so the invariant is true.

Thus in either case, the invariant is satisfied. $\square$
Invariant 1 1. Let $L$ be the level of the functions in $fs$. Then:

$$\text{verify}(P, E_U \cup E_S) = \text{Error} \text{ err} \iff \forall f \in \text{err}. L \leq \text{level}(f)$$

Proof. **Initialization** Initially, $fs = \text{external}(P)$, which is the set of functions at level 0. Thus, the invariant is trivial.

**Maintenance** The calls to $\text{traverseCG}_T$ at line 12 and to $\text{generateSpec}_T$ at lines 7 and 24 trivially maintain the invariant, since $fs$ and $E_S$ are passed directly.

Now consider the call at line 11. Note that, by construction of $E'_S$, every function of level less than or equal to $L$ must either be external or have a specification in $E'_S$. Thus, this invariant follows from Invariant 6.

Invariant 7. Consider a call to either the function $\text{traverseCG}_T(P, E_U, fs, E_S, C, C_{sz})$ or the function $\text{generateSpec}_T(P, E_U, fs, sf, E_S, C, C_{sz})$, using some $\text{synth}_T$ and some specification language $T'$ such that $T'$ is a superset of $T$. Then:

$$\forall E'^G_S \subset T. \text{verify}(P, E_U \cup E'^G_S) = \text{Verified} \implies \text{isSat}_T(E_U, E'^G_S, C)$$

Proof. **Initialization** By assumption, initially $\forall E'^G_S \subset T. \text{verify}(P, E_U \cup E'^G_S) \implies \text{isSat}_T(E_U, E'^G_S, C)$. Since $T'$ is a superset of $T$, trivially we also have $\forall E'^G_S \subset T. \text{verify}(P, E_U \cup E'^G_S) \implies \text{isSat}_T(E_U, E'^G_S, C)$.

**Maintenance** We assume the left hand side of the invariant, since otherwise the invariant is trivial.

Now consider the recursive call to $\text{traverseCG}_T$ at line 11. We will reach this line only if the call to $\text{generateSpec}_T$ at line 7 returned $\text{SEnv}E'_S, C', C'_{sz}$. In turn, we can see that when $\text{generateSpec}_T$ returns such a constructor, $C'$ is exactly the set of constraints it was passed. Thus, the invariant must hold for this set of constraints, and is therefore maintained at line 11.

Now consider the recursive call to $\text{generateSpec}_T$ at line 24. We must show the invariant is true for the constraint set $C \cup C'$. Since $C'$ is the result of apply $\text{evalCE}$ to counterexamples from $\text{symex}$, by lemma 5, for all $E'^G_S \subset T$ such that $\text{verify}(P, E_U \cup E'^G_S) = \text{Verified}$, we have $\text{isSat}_T(E_U, E'^G_S, C')$. Because $T'$ is a superset of $T$, trivially $\text{isSat}_T(E_U, E'^G_S, C \cup C')$. Thus the invariant is maintained.

Finally, consider the call to $\text{generateSpec}_T$ at line 12, with the constraint set $C \cup C'$. We will reach this only if the call to $\text{traverseCG}_T$ at line 11 returns some $\text{Raise} C' C'_{sz}$. Note that the $\text{Raise}$ constructor must have come from some call to $\text{generateSpec}_T(P, E_U, fs', sf', E'_S, C', C'_{sz})$. By the invariant, this means that if $\text{verify}(P, E_U \cup E'_S) = \text{Verified}$ then $\text{isSat}_T(E_U, E'_S, C')$. Therefore, it must also be the case that $\text{isSat}_T(E_U, E'_S, C \cup C')$. Because $T'$ is a superset of $T$, trivially $\text{isSat}_T(E_U, E'_S, C \cup C')$. Thus, the invariant is maintained. □
Invariant 8. Consider a call to the function \( \text{traverseCG}_T(P, E_U, f_s, E_S, C, C_{sz}) \), or the function \( \text{generateSpec}_T(P, E_U, f_s, sf, E_S, C, C_{sz}) \), where Invariant 7 initially holds, and we are using some \( \text{synth}_T \). Then:

\[
\forall E_S^G \subset T. \text{verify}(P, E_U \cup E_S^G) = \text{Verified} \implies \text{isSat}_T(E_U, E_S^G, C \cup C_{sz})
\]

Proof. Initialization By assumption, initially \( \forall E_S^G \subset T. \text{verify}(P, E_U \cup E_S^G) \implies \text{isSat}_T(E_U, E_S^G, C) \).

Maintenance Throughout, we assume the left hand side of the implication is true, since otherwise, the whole invariant is trivially true.

The call to \( \text{generateSpec}_T \) at line 7 maintains the user environment \( E_U \) and constraint sets \( C \) and \( C_{sz} \). Thus the invariant is clearly maintained.

Now consider the recursive call to \( \text{traverseCG}_T \) at line 11. We will reach this only if the call to \( \text{traverseCG}_T \) at line 11 returns some \( \text{SEnv} E_S^G C' C_{sz}'' \). We can see that when \( \text{generateSpec}_r \) returns such a constructor, \( C' \) and \( C_{sz}' \) are exactly the sets of constraints it was passed. Thus, the invariant must hold for this set of constraints, and is therefore maintained at line 11.

Second, we consider the recursive call to \( \text{traverseCG}_T \) at line 12. We will reach this only if the call to \( \text{traverseCG}_T \) at line 11 returns some \( \text{Raise} C' C_{sz}'' \). By Invariant 7, for all \( E_S^G \) such that \( \text{verify}(P, E_U \cup E_S^G) = \text{Verified}, \text{isSat}_T(E_U, E_S^G, C \cup C') \). Now, consider the call to \( \text{generateSpec}_T(P, E_U, f_s', sf', E_S^G, C', C_{sz}''') \) that returned \( \text{Raise} C' C_{sz}''' \), and which satisfies our invariant. In order for the call to return \( \text{Raise} C' C_{sz}''' \), it must be that the call to \( \text{synth}_T \) at line 17 returned \( \text{SynthFail} C_{sz}''' \). By the definition of \( \text{synth}_T \), this means \( \forall E_G \subset T. \text{isSat}_T(C' \cup C_{sz}''', E_G, \implies) \text{isSat}_T(C' \cup C_{sz}''' C_{sz}''', E_G). \) By our invariant, it must be that for all \( E_S^G \) such that \( \text{verify}(P, E_U \cup E_S^G) = \text{Verified}, \text{isSat}_T(E_U, E_S^G, C \cup C' \cup C_{sz}'') \). Thus, it must also be that \( \text{isSat}_T(E_U, E_S^G, C \cup C' \cup C_{sz}'') \).

Finally, we consider the call to \( \text{generateSpec}_T \) at line 24. Since \( C_{sz} \) is not changed, we simply refer to Invariant 7. Thus, the invariant is maintained.

Lemma 8. Consider a call to \( \text{generateSpec}_T(P, E_U, f_s, sf, E_S, C, C_{sz}) \), using some \( \text{synth}_r \), where there are a finite number of specifications in \( T \), with a sound and complete \( \text{symex} \) function and a sound verifier. Then, \( \text{generateSpec}_T \) will terminate.

Proof. At line 17, we synthesize an environment for the functions in \( sf \). By the finiteness of \( T \), there are a finite number \( M > 0 \) of such environments. We will now show that every call to \( \text{generateSpec}_T \) satisfying the conditions in the lemma either makes another such call to \( \text{generateSpec}_T \) reducing \( M \), or directly returns either a \( \text{SEnv} \), \( \text{CEx} \) or \( \text{Raise} \) constructor. If \( \text{synth}_T \) fails to synthesize a model, a \( \text{Raise} \) constructor is returned at line 25, satisfying the lemma. Thus, \( M \) is a variant.

Now, we show that \( M \) does indeed decrease. Suppose execution reaches line 10. In this case, we return a \( \text{SEnv} \) constructor, and the lemma is satisfied. Now, suppose we
instead reached line 22. If a counterexample is directly returned at line 23, the lemma is satisfied. Otherwise, we add a constraint based on the generated counterexamples at line 24, which by lemma 4 must block the current synthesized $E_N$, and thus decreases $M$. Thus, the lemma is satisfied.

To prove lemmas 9 and 11, we proceed by induction on the levels of recursion through $\text{traverseCG}_T$ at line 11 which correspond to levels of the call graph.

**Lemma 9.** Consider a call to $\text{traverseCG}_T(P,E,U,f_s,E_S,C,C_{sz})$, using some $\text{synth}_T$, where there are a finite number of specifications in $T$, with a sound and complete symex function and a sound and complete verifier. Then, if $\not\exists E_G \subseteq T. \text{verify}(P,E_U \cup E_S \cup E_G) = \text{Verified}$, $\text{traverseCG}_T$ will terminate with some $\text{CEx} \text{cexs}$ or $\text{Raise} C' C'_{sz}$.

**Proof.** Consider the levels of the program, 0, \ldots $n$. We will begin by showing that the lemma is true when at level $k = n$, and then showing that, if the lemma is true when at level $k$, the lemma is also true when at level $k - 1$.

**Base case -** $k = n$: Because we are at the base level of the call graph, $sf$, as defined at line 6 of $\text{traverseCG}_T$, must be empty. Consider the call to $\text{generateSpec}_T$ at line 7. By lemma 8, this call is guaranteed to terminate. If it returns a $\text{Raise}$ or $\text{CEx}$ constructor, $\text{traverseCG}_T$ returns that same constructor, and the lemma is satisfied. Otherwise, it must have returned a $\text{SEnv}$ constructor. This means that, at line 20, $\text{generateSpec}_T$ must have returned an empty set of counterexample. However, lemma 7 and the fact that $\text{verify}$ must return an error set guarantees that we will generate a nonempty set of counterexamples $\text{cexs}$. So this is a contradiction, and the lemma is satisfied in the base case.

**Inductive step - True for $k$ implies true for $k - 1$:** At line 7, we call $\text{generateSpec}_T$. By lemma 8, this call is guaranteed to terminate. If it terminates with a $\text{CEx}$ or $\text{Raise}$ constructor, that same constructor is immediately returned by $\text{traverseCG}_T$, and the lemma is satisfied.

Now, suppose $\text{generateSpec}_T$ instead returns a $\text{SEnv}$ constructor. Since we are in the inductive step, $sf$ is nonempty, so we will reach line 11. Since $\not\exists E'_S \subseteq T. \text{verify}(P,E_U \cup E_S \cup E'_S) = \text{Verified}$ it must also be that (for $E'_S$ as defined in $\text{generateSpec}_T$) $\not\exists E'_S. \text{verify}(P,E_U \cup E'_S \cup E_G) = \text{Verified}$. Thus, by induction, this recursive call to $\text{traverseCG}_T$ will return either some $\text{CEx} \text{cexs}$ or $\text{Raise} C' C'_{sz}$. In the former case, the initial call to $\text{generateSpec}_T$ will also return $\text{CEx} \text{cexs}$, immediately satisfying the lemma. In the latter case, $\text{generateSpec}_T(P,E_U,f_s,sf,E'_S,C',C'_{sz})$ is recursively called. $C'_{sz}$ must include some $C_{sz}^F$ that came from a $\text{SynthFail} C_{sz}^F$ in some call to $\text{generateSpec}_T$.

At line 17, we synthesize an environment for the functions in $sf$. By the finiteness of $T$, there are a finite number $M > 0$ of such environments. We will now show that every call to $\text{generateSpec}_T$ satisfying the conditions in the lemma either makes another such call to $\text{generateSpec}_T$ reducing $M$, or directly returns either a $\text{CEx}$ or $\text{Raise}$ constructor. Thus, $M$ is a variant.
First, suppose execution reaches the recursive call to \texttt{generateSpec}_T at line 24. If we reach line 23 or 25, a \texttt{CEx} or \texttt{Raise} constructor will be directly returned. Notice that if we return \texttt{Raise} \( C_{sz} \cup C'_{sz} \), then by the definition of \texttt{synth} \( \neg \text{isSat}(E_U \cup E_S \cup E_G) \). Therefore when in the future our synthesis is guided by \( C'_{sz} \), we know that \( E_S' \) will not be returned, thus decreasing \( M \).

Otherwise we will reach line 24 and add a constraint based on the generated counterexamples at line 24. By lemma 4 this constraint blocks the current synthesized \( E_N \), decreasing \( M \). Thus, the lemma is satisfied.

**Lemma 10.** Suppose we are using some \( \text{synth}_T \), where there are a finite number of specifications in \( T \), and that we have a complete \texttt{symex} function. Consider a call to \texttt{generateSpec}_T \( (P,E_U,fs,sf,E_S,C,C_{sz}) \), where Invariant 8 holds. Then, if \( \exists E_G \subset T. \text{verify}(P,E_U \cup E_S \cup E_G) = \text{Verified} \), such that \( E_S \) and \( E_G \) are disjoint, \texttt{generateSpec}_T will terminate with some \texttt{SEnv} \( E'_S, C', C'_{sz} \).

**Proof.** We will show that \texttt{generateSpec}_T \( (P,E_U,fs,sf,E_S,C,C_{sz}) \) either directly returns some \texttt{SEnv} value, or it will eventually make a recursive call to \texttt{generateSpec}_T which will return return some \texttt{SEnv} value. In either case the theorem is satisfied.

By Invariant 2, \( sf = \cup_{f \in fs}\{g|g \in \text{calls}(f) \land g \notin E_S\} \). By the finiteness of our specification language, there is a finite number \( M \) of possible environments for the functions in \( sf \) satisfying the constraints in \( C \cup C_{sz} \). Note that \( M > 0 \), since by assumption there is at least one environment \( E^G \) that will allow verification to succeed, and by Invariant 8, we must have \( \text{isSat}(E_U, E^G, C \cup C_{sz}) \). In the rest of the proof, we will show that:

- At each call to \texttt{generateSpec}_T \( (P,E_U,fs,sf,E_S,C \cup C', C_{sz}) \) at line 24, the value of \( M \) decreases.

- Each call to \texttt{generateSpec}_T will always recursively call itself at line 24 with inputs that satisfy our lemma.

Thus, \( M \) acts as a variant- it will continually decrease, but not drop below 1, until we eventually return a specification environment.

Now, we will justify the above statements, and thus complete the proof:

- We will show that at each call to \texttt{generateSpec}_T \( (P,E_U,fs,sf,E_S,C \cup C', C_{sz}) \) at line 24, the value of \( M \) decreases.

At line 17, we synthesized some new \( E_N \). By definition of \texttt{synth} \( E_N \models C \). Then, \( C' \) is the result of calling \texttt{evalCE} on counterexamples generated from calling \texttt{synth}_T with \( E_U \cup E_S \cup E_N \). (the set is not empty, by the pattern match.) By lemma 4, \( E_U \cup E_S \cup E_N \not\models C' \). Thus, on at subsequent calls to \texttt{generateSpec}_T, where \( C' \) is included in the constraint set, \( E_N \) will not be synthesizable, this decreasing \( M \).
Now we will see that each call to \( \text{generateSpec}_T \) will always eventually recursively call itself at line 24 with inputs that satisfy our lemma.

This is mostly straightforward from inspection. Note that the call to \( \text{verifyCEx} \) at line 20 is guaranteed to terminate by lemma 7.

Thus, the lemma is satisfied.

**Lemma 11.** Suppose we are using some \( \text{synth}_T \), where there are a finite number of specifications in \( T \), and that we have a complete \( \text{symex} \) function. Consider a call to \( \text{traverseCG}_T(P, E_U, fs, E_S, C, C_{sz}) \), where Invariant 8 holds. Then, if \( \exists E^G_S \subset T.\text{verify}(P, E_U \cup E_S E_S \cup E^G_S) = \text{Verified} \), and such that \( E_S \) and \( E^G_S \) are disjoint \( \text{traverseCG}_T \) will terminate with some \( \text{SEnv} E'_S C' C'_{sz} \) such that \( \text{verify}(P, E_U \cup E'_S) = \text{Verified} \).

**Proof.** We will show that \( \text{traverseCG}_T(P, E_U, fs, E_S, C, C_{sz}) \) either directly returns some \( \text{SEnv} \) value, or it will eventually make some recursive call to \( \text{traverseCG}_T \) which will return return some \( \text{SEnv} \) value. In either case the theorem is satisfied.

Consider the levels of the program, 0, \ldots, \( n \). We will begin by showing that the lemma is true when at level \( k = n \), and then showing that, if the lemma is true when at level \( k \), the lemma is also true when at level \( k - 1 \).

**Base case - \( k = n \):** At line 6, we define \( sf = \bigcup_{f \in fs} \{ g | g \in \text{calls}(f) \land g \notin E_S \} \). By lemma 10, the call to \( \text{generateSpec}_T \) at line 7 will terminate with some \( \text{SEnv} E'_S C' C'_{sz} \). Because we are at level \( n \) of the call graph, \( E_S \) must contain all the internal functions in \( P \). Thus, \( sf \) will be the empty set, and so we will reach line 10, and return \( \text{SEnv} E'_S C' C'_{sz} \). This satisfies the theorem.

**Inductive step - True for \( k \) implies true for \( k - 1 \):** Consider \( sf \) as defined at line 6 of \( \text{generateSpec}_T \). By the finiteness of our specification language, there is a finite number \( M \) of possible environments for the functions in \( sf \) satisfying the constraints in \( C \cup C_{sz} \). Note that \( M > 0 \), since by assumption there is at least one environment \( E^G \) that will allow verification to succeed, and by Invariant 8, we must have \( \text{isSat}_T(E^G, C \cup C_{sz}) \). In the rest of the proof, we will show that:

- At each call to \( \text{traverseCG}_T(P, E_U, fs, E_S, C \cup C', C_{sz}) \) at line 12, the value of \( M \) decreases.

- Each call to \( \text{traverseCG}_T \) will always eventually either recursively call itself at line 12, or it will call itself at line 11 with inputs that satisfy our lemma.

Thus, \( M \) acts as a variant- it will continually decrease, but not drop below 0, until we eventually make a call satisfying our inductive hypothesis to \( \text{generateSpec}_T \) at line 11.

Now, we will justify the above statements, and thus complete the proof:

- We will show that at each call to \( \text{traverseCG}_T(P, E_U, fs, E_S, C \cup C', C_{sz}) \) at line 12, the value of \( M \) decreases.
If we reach line 12, it means that we synthesized some $E'_S$, then recursively called $\text{traverseCG}_T(P, E_U, f's, E'_S, C, C_{sz})$. This recursive call returned some $\text{Raise } C^F C^F_{sz}$. This means that, there was a call to $\text{generateSpec}_T$ in which a call to $\text{synth}_T$ with the environment $E_U \cup E'_S$ returned $\text{SynthFail } C^F_{sz}$. By the definition of $\text{synth}_T$, this means that $\neg \text{isSat}_T(E_U, E'_S, C^F_{sz})$. $C^F_{sz}$ is returned to our initial call to $\text{generateSpec}_T$ in the Raise constructor, and added to our set of size constraints in the recursive call. Thus, on subsequent calls to $\text{generateSpec}_T$, $E'_S$ will not be synthesizable, this decreasing $M$.

Now we will show that each call to $\text{traverseCG}_T$ will always eventually either recursively call itself at line 12, or it will call itself at line 11 with inputs that satisfy our lemma. This is mostly straightforward from inspection. The only tricky point is that we must show that if we recursively call $\text{traverseCG}_T$ at line 11 with inputs that do not satisfy our lemma, this call will return with some Raise that will result in a recursive call to $\text{traverseCG}_T$ at line 12. Lemma 9 is sufficient to establish that the call will either return some CEx or Raise constructor. Our soundness lemma 13 shows that, given these two possibilities it must in fact return a Raise constructor, as a CEx constructor would be directly returned, violating soundness. Thus, we will recursively call $\text{generateSpec}_T$ at line 12 in this case.

Thus, the lemma is satisfied.

**Theorem 6** (Completeness of $\text{initInfer}_T$). Consider a call to $\text{initInfer}_T(P, E_U)$, where there are a finite number of specifications in $T$, using sound and complete $\text{verify}$, $\text{symex}$, and $\text{synth}_T$ functions. Then, $\text{initInfer}_T(P, E_U)$ is complete.

**Proof.** Follows from lemma 11, and the observation that, if there is a counterexample but $\text{traverseCG}_T$ returns $\text{Raise}$, we will fall into a loop searching for the counterexample at line 2.

**Lemma 12** (Soundness of environments). Assuming a sound verify function $\text{verify}$ and a sound counterfactual symbolic execution engine $\text{symex}$, and inputs satisfying Invariant 6, if $\text{traverseCG}_T(P, E_U, f's, E_S, C, C_{sz}) = \text{SEnv} E'_S C' C'_{sz}$, then $\text{verify}(P, E_U \cup E'_S) = \text{Verified}$.

**Proof.** In order to terminate with a SEnv constructor, $\text{traverseCG}_T$ must reach line 10. In order to force a specific call to $\text{generateSpec}_T$ to directly reach this line, $sf = \{\}$. By our assumption that all functions are reachable from an external function, and the definition of $sf$ at line 6, this implies that every function is either external or in $E_S$. If we instead executed line 11, calling $\text{traverseCG}_T(P, E_U, f's', E'_S, C, C_{sz})$ (where $f's' = sf = \{\}$) Invariant 6 would tell us that:

$$\text{verify}(P, E_U \cup E'_S) = \text{Error } err \iff \forall f \in err. \ (f \notin \text{external}(P) \land \neg f \in E'_S)$$
Suppose the left hand side of the implication was true. Then, there must be some \( f \) that is not external and is not in \( E'_S \). But this is a contradiction, so it must be that, so following the same logic as in the proof of the invariant, \( \text{verify}(P, E_U \cup E'_S) = \text{Verified} \). Thus our algorithm soundly returns only environments that allow verification.

\[ \text{Lemma 13 (Soundness of counterexamples). Assuming we have a sound verify function verify and a sound counterfactual symbolic execution engine symex, if} \]

\[ \text{traverseCG}_T(P, E_U, fs, E_S, C, C_{sz}) = \text{CEx cexs}', \]

\[ \text{then cexs}' \text{ contains concrete counterexamples contradicting the specifications in } E_U. \]

\[ \text{Proof.} \text{ Note that the user environment } E_U \text{ is never changed between recursive calls to} \]

\[ \text{traverseCG}_T \text{ or generateSpec}_T. \text{ Thus, it is sufficient to consider the only place that} \]

\[ \text{CEx } C' \text{ is constructed in generateSpec}_T, \text{ at line 23. Thus, we need only show that all} \]

\[ \text{counterexamples returned from that line contradict the specifications in } E_U. \text{ Indeed,} \]

\[ \text{this property is trivial from the soundness of symex and the definition of evalCE, which} \]

\[ \text{returns counterexamples only if they are concrete postcondition counterexample to a} \]

\[ \text{external function, or if they are a precondition counterexamples, where the caller is} \]

\[ \text{an external function, and the callee specification being violated is in } E_U. \]

\[ \text{Theorem 7 (Soundness of initInfer}_T). \text{ Consider a call to} \]

\[ \text{initInfer}_T(P, E_U), \text{ using sound verify, symex, and synth}_T \text{ functions. Then,} \]

\[ \text{initInfer}_T(P, E_U) \text{ is sound.} \]

\[ \text{Proof.} \text{ Follows from lemmas 12 and 13.} \]

\[ \text{Theorem 8 (Soundness of iterateInfer}_T). \text{ Consider a call to} \]

\[ \text{iterateInfer}_T(P, E_U, 1, \{} \}

\[ \text{for a size-bounded theory } T. \text{ using sound verify, symex, and synth}_T \text{ functions. Then,} \]

\[ \text{iterateInfer}_T(P, E_U, 1, \{) \text{ is sound.} \]

\[ \text{Proof.} \text{ iterateInfer}_T \text{ will return an SEnv only if line 3 is reached in (for some} \]

\[ k' \text{ iterateInfer}_{T^{k'}}), \text{ and will return a CEx only if line 4 or line 7 is reached in} \]

\[ \text{iterateInfer}. \text{ These will be only returned from iterateInfer}_{T^{k'}} \text{ if they are returned from} \]

\[ (\text{for some } k'') \text{ traverseCG}_{T^{k''}}, \text{ or directly returned from the sound symex function. Thus, the soundness of} \]

\[ \text{traverseCG}_{T^{k''}}, \text{ as established in lemmas 12 and 13, and} \]

\[ \text{the soundness of the symex function, ensures the soundness of iterateInfer}_T. \]

\[ \text{Lemma 14. Consider a call to} \text{traverseCG}_T(P, E_U, \{} C, \{) \text{ using some synth}_T \]

\[ \text{and some specification language } T' \text{ such that } T' \text{ is a superset of } T. \text{ Suppose that:} \]

\[ \forall E_S^G \subset T. \text{verify}(P, E_U \cup E_S^G) = \text{Verified} \implies \text{isSat}_T(E_U, E_S^G, C) \]
If the call to \texttt{traverseCG}\(_T\) returns \texttt{Raise} \(C'\) \(C'_sz\), then

\[
\forall E_S^G \subset T.\text{verify}(P,E_U \cup E_S^G) = \text{Verified} \implies \text{isSat}_T(E_U,E_S^G,C')
\]

**Proof.** Follows from the construction of a \texttt{Raise} in \texttt{generateSpec}\(_T\) at line 25 and Invariant 7. \hfill \Box

**Theorem 9** (Completeness of \texttt{iterateInfer}\(_T^\_1\)). Consider a call to

\[
\text{iterateInfer}_T(P,E_U,1,\{}\}
\]

for a size-bounded theory \(T\) using sound and complete \texttt{verify}, \texttt{symex}, and \texttt{synth}\(_T^k\) functions. Then, \(\text{iterateInfer}_T(P,E_U,1,\{}\}\) is complete over specification in \(\bigcup_1^m T^k\).

**Proof.** First we show part 1 of the definition of completeness. Consider a call to \(\text{iterateInfer}_T^\_1(P,E_U,1,\{}\}\) assuming there exists some \(E_S\) satisfying the requirements for part 1 of the definition of completeness. Suppose the minimal such environment is \(E_S^m\), and \(\text{size}(E_S^m) = m\).

For all \(k < m\), by lemma 9 and the soundness of \(\text{traverseCG}_T^k\) (lemma 13), \(\text{generateSpec}_T^k(P,E_U,fs,sf,\{}\},C,\{}\})\ will terminate with some \texttt{Raise} \(C_N\_\). By the soundness of \texttt{symex}, at line 6 the calls to \texttt{symex} must produce empty sets, and so we will call \(\text{iterateInfer}_T^{k+1}(P,E_U,d+1,C_N)\) by lemma 14, and by induction over \(1\ldots m\), we must have that \(\text{isSat}_T(E_U,E_S^m,C_N)\). Thus, we will eventually call \(\text{traverseCG}_T^m(P,E_U,d,\{}\},C'_N,\{}\})\) with inputs satisfying Invariant 8, and so by lemma 11 we will return a specification environment from \(\text{iterateInfer}_T^m\) (and thus, also from the initial call to \(\text{iterateInfer}_T^\_1\).) Thus \(\text{iterateInfer}_T^k\) satisfies part 1 of the definition of completeness.

Next, we show part 2 of completeness. By the soundness of \texttt{verify}, since there exists a concrete counterexample \(\exists E_S^G \subset T.\text{verify}(P,E_U \cup E_S^G) = \text{Verified}\). Thus, by lemma 9, every call to \(\text{traverseCG}_T^k\) will terminate with some \texttt{CEx cexs} or \texttt{Raise} \(C'\) \(C'_sz\). In the former case, we directly return \texttt{CEx cexs}, meeting the requirement of completeness. In the latter case, we call \texttt{symex} at line 5 with increasingly large depths on each iteration of \(\text{iterateInfer}_T^k\). By the definition of completeness of \texttt{symex}, whenever the exists a concrete counterexample \texttt{cex} \(E_U\) with \texttt{caller}(\texttt{cex}) = \text{f} there is some depth \(d\) such that \texttt{symex}(P,E_U,d,f) returns a concrete counterexample. Thus, we are guaranteed to eventually find a concrete counterexample once the depth is large enough. Therefore, \(\text{iterateInfer}_T^k(s,a,t,i)\) satisfies part 2 of the definition of completeness.

Thus, we have shown that given a a sound and complete verify function \texttt{verify}, a sound and complete counterexample generator \texttt{symex}, and a complete synthesizer \texttt{synth}\(_T\), \(\text{iterateInfer}_T^k\) is complete. \hfill \Box