GUARANTEES IN PROGRAM SYNTHESIS

by

Qinheping Hu

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2021

Date of final oral examination: 06/02/2021

The dissertation is approved by the following members of the Final Oral Committee:
Loris D'Antoni, Assistant Professor, Computer Sciences
Thomas W. Reps, Professor, Computer Sciences
Somesh Jha, Professor, Computer Sciences
Steffen Lempp, Professor, Mathematics

Acknowledgments

I have had the incredible privilege of being advised by Loris D'Antoni. Loris is profoundly creative and enthusiastic as a mentor, and nice and sincere as a friend. He has always been available to provide sound advice, crank through a proof, hack on a piece of code, or edit my writing. Thank you for encouraging, mentoring, and providing help along the way. I could not ask for a better mentor or friend.

Throughout my Ph.D., I have been lucky to collaborate with many wonderful people: Tom Reps, John Cyphert, Jinwoo Kim, Rong Pan, Rishabh Singh, Jason Breck, and Roopsha Samanta. Especially, I want to thank Tom, who is like my second mentor, for the collaboration for much of my Ph.D. and for setting the bar high. I am lucky to count these talented individuals as colleagues and friends, and I gratefully acknowledge their contributions to this dissertation.

To the members of the MadPL group, thank you for making my time in grad school a blast. You gave excellent seminars with many exciting topics, PL lunch which creates a friendly and stimulating research environment, and insightful feedback every time I gave a talk.

Loris D'Antoni, Thomas Reps, Somesh Jha, and Steffen Lempp all graciously agreed to serve on my dissertation committee. I am grateful for the time they spent reading my proposal document and this dissertation. Their many insightful comments and helpful suggestions have improved it in numerous ways.

Finally, I am forever grateful for the love of my family: Wenlin, Karen, Xiao, Kefu, and Xiaoyu. Thank you for believing in and supporting me every step of the way.

Contents

List of Tables

List of Figures

ABSTRACT

GUARANTEES IN PROGRAM SYNTHESIS

Qinheping Hu

Under the supervision of Professor Loris D'Antoni

Program synthesis is the classic problem of automatically finding a program in some search space that satisfies a given correctness specification. Program synthesis significantly impacts software development because it reduces programmers' efforts to produce programs with daunting detail. Unfortunately, it is usually not enough to produce any correct solution in program synthesis. For many program synthesis problems, there are multiple correct solutions in the search space, but some of them are not expected because, for example, their sizes are too large to be read. Besides, for synthesis problems that admit no solution—we call such synthesis problems *unrealizable*—, most of the enumeration-based synthesizers do not terminate.

Therefore, besides correctness and efficiency, one may want to ask two more questions about a synthesis solver: *Can the solver provide a good solution when there are multiple ones? Can the solver provide a proof when there is no solution?* In this dissertation, we introduce two types of guarantees in program synthesis: *quantitative objectives* and *proof* of unrealizability.

First, we present QSyGuS, a synthesis framework extending syntax-guided synthesis (SyGuS) with quantitative syntactic objectives, which allow users to prefer some solutions over others, and an algorithm QuaSi of solving QSyGuS problems. QuaSi reduces QSyGuS problems to SyGuS problems and solves the reduced problems by off-the-shelf solvers. Second, we introduce two algorithms Nope and Nay of proving the unrealizability of SyGuS problems. Nope is based on the idea of encoding an unrealizable problem as a verification problem. Nay proves the unrealizability with grammar-flow analysis. Finally, we focus on a particular kind of quantitative objectives: asymptotic resource usage. We develop a type-based algorithm SynPlexity for solving synthesis problems with asymptotic resource usage.

Chapter 1

Introduction

In the typical programming paradigm, programmers tell machines *what to do* via line-by-line codes. For example, if the programmer wants to write the `max2` function, she must first know the intent of `max2`—computing the maximum integers given two integers as input. Then, she needs to write the problem line-by-line. Here is a possible implementation she will write:

$$\texttt{if x} > \texttt{y then x else y}.$$

To determine if the program is correct, the programmer may also test the program with some test cases. For example, the test cases for the `max2` function can be $\{(1,2) \mapsto 2, (3,2) \mapsto 3\}$, that is, the output should be 2 when the input is $(1,2)$ and the output should be 3 when the input is $(3,2)$.

In the above example, coding is not that hard; hence we do not need to worry about debugging. However, in tasks more complicated than `max2`, coding and debugging can be tedious and error-prone for programmers. So a question then arises, "We have the intents of desired programs. We also have the test cases. Can we bypass the tedious work of coding and debugging with the intent and test cases?" The answer to this question is *program synthesis* [PR89]—the technique of automatically finding programs that meet user intent.

In program synthesis, users tell machines *what they want* via user intent de-

scribed by a variety of *specifications*, e.g., test cases [Gul16, Gul11], logical predicates [ABJ+13], traces [DSS16, HSSD19], natural language [DGH+16, GM14, LGS13], and partial programs [SL08]. The goal of a program synthesizer is to find a program from a given *search space* that satisfies the specifications. The graph shown in Fig. 1.1 illustrates the setting of a typical program synthesizer.

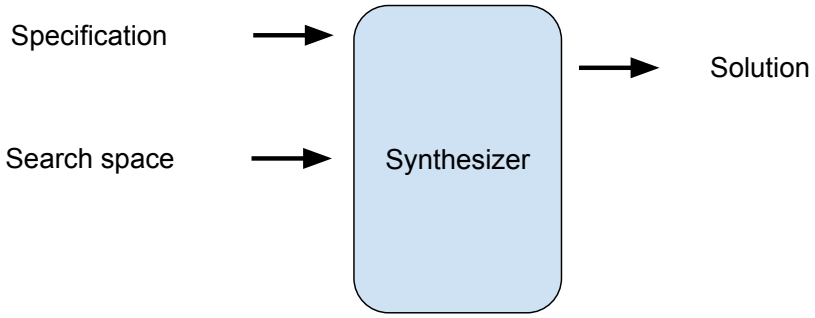


Figure 1.1: Program synthesizer.

For the `max2` example, the specification can be described by the following predicate:

$$\forall x, y.\, \texttt{max2}(x, y) \geqslant x \wedge \texttt{max2}(x, y) \geqslant y \wedge (\texttt{max2}(x, y) = x \vee \texttt{max2}(x, y) = y),$$

or the test cases we have seen above. The search space can be the language (described by the grammar shown in Fig. 1.2) consisting of all programs built from $+$, `if-then-else`, Boolean operators, constants, and variables. With these specification and search space, a program synthesizer can automatically find an implementation of `max2`; and hence the programmer does not need to code and debug by herself.

$$
\begin{array}{rl}
\text{Start} ::= & \text{Start} + \text{Start} \\
& \mid \texttt{if}\ \text{BExpr}\ \texttt{then}\ \text{Start}\ \texttt{else}\ \text{Start} \\
& \mid \texttt{x} \mid \texttt{y} \mid 0 \mid 1
\end{array}
\qquad
\begin{array}{rl}
\text{BExpr} ::= & \text{Start} > \text{Start} \\
& \mid \neg \text{BExpr} \\
& \mid \text{BExpr} \wedge \text{BExpr}
\end{array}
$$

Figure 1.2: Grammar that describes the search space of the `max2` problem.

1.1 When does Program Synthesis Fall Short of Expectations?

Program synthesis benefits software development in the sense that it significantly reduce the workload of producing programs that have concise intuition but daunting details, and it also support programming-by-intent for non-expert users. Unfortunately, a major inherent challenge in program synthesis restricts it from being used more widely: the user intent that describes the behavior of the synthesized program is usually ambiguous. There can be multiple programs that are *correct* with respect to user intent; synthesizers are usually unpredictable and can return any of them as a solution. However, an arbitrary *correct* program is not enough when the user expects a program, for example, with small size, high likelihood , or small complexity. The diagram shown in Fig. 1.3 illustrates the relation between the search space and the space of correct programs in synthesis problems. The intersection is the space of solutions. When there are multiple programs in the intersection, a synthesizer can return any of them as the solution.

For example, even with the full specification, a synthesizer may still find the following implementation of `max2`

$$\texttt{if}\ \texttt{x} > \texttt{y}\ \texttt{then}\ (\texttt{if}\ \texttt{x} > 0\ \texttt{then}\ \texttt{x}\ \texttt{else}\ \texttt{x})\ \texttt{else}\ \texttt{y}.$$

This implementation is also correct, but contains a redundant `if-then-else` operator—the inner one. One can see from this example implementation that a similar implementation can contain an arbitrary number of redundant `if-then-else`

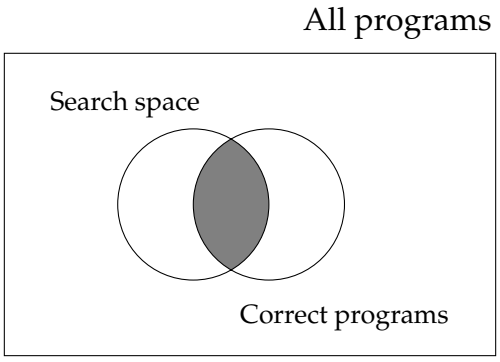


Figure 1.3: The search space of synthesis problems intersects with correct programs with respect to the specification. The gray part represent the space of solutions.

and still be correct. A program with too many redundant `if-then-else` will be unreadable and, of course, is not expected by the user.

Besides the cases where there are multiple solutions, program synthesis can also fall short of expectations when there is *no* solution—we call synthesis problems admitting no solutions *unrealizable* problems. It can happen for several reasons: specifications provided by users are buggy, or the search space is too restricted. An enumeration-based synthesizer will try to enumerate all candidates in the search space on an unrealizable synthesis problem with an infinite search space and never terminate. The diagram in Fig. 1.4 illustrates that the search space and the space of correct programs are disjoint in unrealizable synthesis problems.

To construct an example of unrealizable synthesis problems, we change the search space of the `max2` problem to the language described by the grammar shown in Fig. 1.5. Note that this new grammar produces only linear terms, which can not express the `max2` function! Therefore there is no solution in the search space; and this synthesis problem is unrealizable.

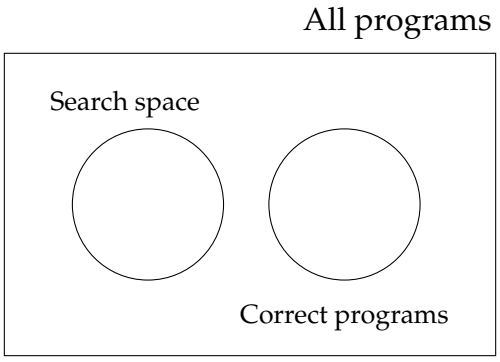


Figure 1.4: Unrealizable synthesis problems.

$$\text{Start} ::= \text{Start} + \text{Start} \mid x \mid y \mid 0 \mid 1$$

Figure 1.5: Grammar for linear terms.

1.2 Guarantees in Program Synthesis

This dissertation proposes two approaches to address the two drawbacks introduced in the previous section. The first one is *quantitative objectives*, which extend the correctness specifications to more syntactic or semantic characterizations of programs, e.g., the sizes of programs, the likelihood of programs, and the complexity of programs. Quantitative objective allows users to prefer one solution over another and ask the synthesizer to return the best one. The other one is to *prove the unrealizability* of synthesis problems. We call these two approaches *guarantees* in program synthesis because they provide guarantees about what solution is returned if multiple ones exist or why there is no solution, and make program synthesizers more predictable and reliable. The graph shown in Fig. 1.6 illustrates synthesizers with guarantees. The two boxed items are the guarantees proposed by this dissertation.

In this section, we will first introduce two kinds of quantitative objectives and then discuss the ability of synthesizers to prove unrealizability.

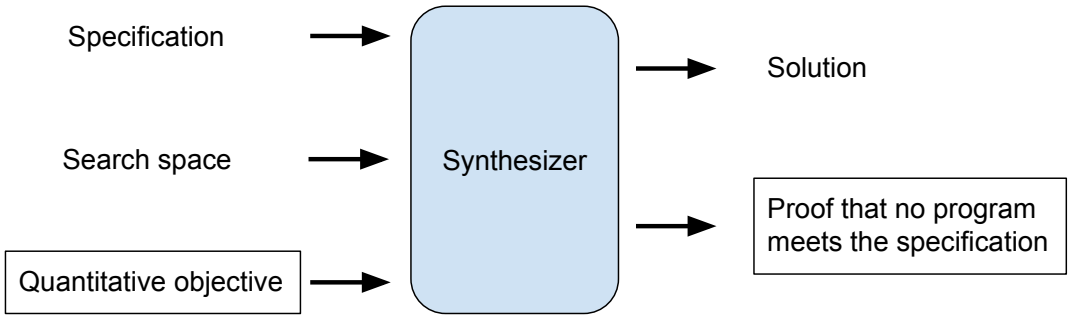


Figure 1.6: Program synthesis with guarantees.

Quantitative Syntactic Objectives

In program synthesis, besides correctness, users may also care about some syntactic characterizations of solutions. Quantitative syntactic objectives are a constraint on the syntactic of solutions, which allow users to express such beyond-the-correctness intent. For example, let us consider three implementations of `max2` functions.

$$
\begin{aligned}
max2_1(x, y) &= x \\
max2_2(x, y) &= \texttt{if } x > y \texttt{ then } (\texttt{if } x > 0 \texttt{ then } x \texttt{ else } x) \texttt{ else } y \\
max2_3(x, y) &= \texttt{if } x > y \texttt{ then } x \texttt{ else } y
\end{aligned}
$$

The first implementation $max2_1$ does not satisfy the correctness specification and hence is incorrect. The implementations $max2_2$ and $max2_3$ are both correct, and even semantically equivalent. However, they are syntactically different, i.e., the second `if-then-else` operator in $max2_2$ is redundant. If a user wants a solution with no more than one `if-then-else` operator to avoid redundant conditionals, she can specify such intent with a quantitative syntactic objective $\#_{\texttt{if}}(P) < 2$, which states that the number of `if-then-else` operators in a solution P is less than 2. Such quantitative objectives are commonly needed for program-synthesis techniques that tend to build decision trees as solutions, such as the state-of-the-art synthesizer

CVC4-SyGuS [ARU17a, RDK+15] to avoid returning a lookup table over the input domain as a solution.

Besides the number of occurrences of a specified operator, the size of programs, the likelihood of programs, and their composition can also be expressed as quantitative syntactic objectives. The size of a program can be computed by summing up the numbers of all operators. The likelihood of a program with respect to some given distribution can be computed as the product of the likelihood of all operators in the program.

Finally, users can also specify optimizing objectives—the solutions must be optimal with respect to some given metric. In the example above, a user may want an implementation of the `max2` function that contains the *least* number of `if-then-else` operators. In such a case, the quantitative syntactic objective $\mathtt{minimize}\ \#_{\mathtt{if}}(P)$ can be used to express the intent. Note that when optimizing objectives are present, a synthesizer needs to find not only a solution, but also a proof showing that the solution is optimal. Such additional requirements motivated us to study the problem of proving unrealizability, which we will discuss later in §1.2.

Quantitative Semantic Objectives

Similar to quantitative syntactic objectives, quantitative semantic objectives are also used to prefer some solutions over others. Quantitative semantic objectives specify the expected quantitative semantic characterizations of solutions in program-synthesis problems. In this dissertation, we focus on a particular kind of quantitative semantic objective, the resource usage bounds of programs. We take time complexity as an example to illustrate program synthesis with resource usage bounds.

$$\mathtt{prod}_{\mathtt{lin}} \;=\; \lambda x.\lambda y.\mathtt{if}\ x == 0\ \mathtt{then}\ x\ \mathtt{else}\ (\mathtt{plus}\ y\ (\mathtt{prod}_{\mathtt{lin}}\ (\mathtt{dec}\ x)\ y)) \quad (1.1)$$

The program $\mathtt{prod}_{\mathtt{lin}}$ shown in Eqn. (1.1) computes the product of two non-negative integers x and y, where `plus` (summing up two integers) and `dec` (decreasing an integer by one) are auxiliary functions. This implementation is correct

but inefficient. Let us count each call to an auxiliary function as one step; and let $T(x)$ denote the number of steps in which the program runs with input x. The implementation in Eqn. (1.1) runs in $\Theta(x)$ steps because $T(x)$ satisfies the recurrence $T(x) = T(x-1) + 2$, implying $T(x) \in \Theta(x)$. That is $\texttt{prod}_{\texttt{lin}}$ runs in a linear number of steps with respect to x.

With the quantitative semantic objective $\texttt{prod} : \langle O(\log x) \rangle$, one can specify that the synthesized implementation of `prod` should run in $O(\log x)$ steps. Here is an example of an implementation of `prod` that runs in a logarithmic number of steps.

```
prod = λx. λy. if x == 0 then x else                                    (1.2)
       if even x then double (prod (div2 x) y)
                 else plus y (double (prod (div2 x) y))
```

There are two motivations resource-usage bounds as quantitative semantic objectives. First, users usually want efficient program implementations instead of inefficient ones. Resource bounds can rule out those inefficient solutions. Second, with the insight that the solution should be a divide-and-conquer implementation with logarithmic complexity, a resource-usage bound can be used to guide the search and rule out all implementations with linear complexity.

Proving Unrealizability of Synthesis Problems

In program synthesis, the space of all candidate programs is called the *search space*. The goal of a synthesis problem is, given the correctness specification and the search space, to find a program in the search space such that the program satisfies the specification; we say such a synthesis problem is realizable—it admits a solution. However, there are also cases where there is no program satisfying the specification in the search space; we say such problems *unrealizable*.

The `max2` problem with the grammar shown in Fig. 1.5 is an example of an unrealizable synthesis problem. We observe that all candidates in the search space are linear terms and can be written as $ax_by + c$ for some constants a, b, and c.

However, any implementation of the `max2` function should be non-linear in the sense that it should output x when the input x is greater (or equal) than y, and output y otherwise. Therefore, the synthesis problem admits no solution in the search space, and hence is unrealizable.

The ability to prove unrealizability will provide users with more information. Suppose that a user is synthesizing a complicated problem with a synthesizer that has no ability to prove unrealizability. If the problem is realizable and the user waits for the synthesizer for a long enough time, the synthesizer will eventually return a solution. However, if the problem is unrealizable, no matter how long the user waits, she does not know whether the cause is that the problem is unrealizable, or the solving time is not long enough.

In addition, the ability to prove unrealizability provides the ability to prove that a solution is optimal. To show that a solution P of some synthesis problem is optimal, it is sufficient to show that there is *no* program better than P in the search space. For any program P in the search space, let the synthesis problem $Sy_{>P}$ be the problem of finding a better program than P. The optimality of program P is implied by $Sy_{>P}$ being unrealizable.

While many solvers can now efficiently find solutions when they exist, there has been effectively no prior work on proving that a given synthesis problem is unrealizable. A key property of the previous example is that the grammar is infinite. When such a synthesis problem is realizable, any search technique that systematically explores the infinite search space of possible programs will eventually identify a solution to the synthesis problem. In contrast, proving that a problem is unrealizable requires showing that every program in the infinite search space fails to satisfy the specification. This problem is in general undecidable [CRST15]. Although we cannot hope to have an algorithm for establishing unrealizability, the challenge is to find a technique that succeeds for the kinds of problems encountered in practice.

1.3 Contributions and Outline

This dissertation aims to demonstrate that providing guarantees in program synthesis is an effective way of improving the reliability and availability of synthesizers. The remainder of this dissertation is arranged into five chapters.

Chapter 2 presents the formalization of quantitative syntactic objectives in synthesis program with search spaces described by tree grammars. The formalization is based on weighted grammars, and can be instantiated to a popular synthesis framework: syntax-guided synthesis. We also introduce a meta-algorithm for solving synthesis problems with quantitative syntactic objectives, including both constraint objectives and optimizing objectives. Note that, in our algorithm, the proof of a solution being optimal is reduced to the proof of the unrealizability of a particular synthesis problem, which motivates the study of proving unrealizability introduced in the following chapter.

In Chapter 3, we introduce two algorithms for proving the unrealizability of synthesis problems. One is a meta-algorithm reducing the problem of proving unrealizability to the program-verification problem. The second algorithm is a domain-specific algorithm for proving the unrealizability of synthesis problems within linear-integer-arithmetic. The meta-algorithm is more general but undecidable, while the second algorithm provides us with a decision procedure to decide the unrealizability of a particular family of synthesis problems.

Chapter 4 presents the formalization of a particular kind of quantitative semantic objective in synthesis problems with specifications as liquid types: asymptotic resource bounds of synthesized programs. We also introduce a type-guided algorithm for solving synthesis problems with asymptotic resource bounds.

In Chapters 5 and 6, we discuss the relationship of the techniques introduced in this dissertation to the rich variety of existing works and provide some directions for future work.

Part I

Quantitative Syntactic Objectives in Synthesis

In program synthesis, besides correctness, users may also care about some syntactic characterizations of solutions. Quantitative syntactic objectives are a constraint on the syntactic characteristics of solutions, which allow users to express such beyond-just-correctness intent. Quantitative objectives can be used to specify the number of occurrences of a specified operator, the size of programs, the likelihood of programs, and their composition.

Quantitative objectives include constraint objectives and optimizing objectives. Constraint objectives ask the solutions of synthesis problems to have weights satisfying given constraints. Optimizing quantitative objectives ask the solutions of synthesis problems to be optimal with respect to some specified metric. Note that when optimizing objectives are present, a synthesizer needs to find not only a solution but also a proof that shows that the solution is optimal.

In Chapter 2, we introduce the formalization of quantitative syntactic objectives based on tree grammars and the meta-algorithm of solving synthesis problems with quantitative syntactic objectives. The algorithm can solve both constraint objectives and optimizing objectives. The proof of a solution being optimal in solving optimizing objectives is reduced to the proof of unrealizability in our algorithm. So we also study the problem of proving unrealizability in this part.

In Chapter 3, we introduce a meta-algorithm for proving unrealizability of synthesis problems, and a domain-specific algorithm for proving unrealizability of synthesis problems based on tree grammars.

In this part, we focus on syntax-guided synthesis problems (SyGuS)— a popular framework that unifies a large family of synthesis problems. However, the algorithms we introduce can be generalized to synthesis problems with search spaces described by tree grammars beyond SyGuS.

Chapter 2

Syntax-Guided Synthesis with Quantitative Syntactic Objectives

2.1 Introduction

The goal of program synthesis is to find a program in some search space that meets a specification—e.g., a set of examples or a logical formula. Recently, a large family of synthesis problems has been unified into a framework called syntax-guided synthesis (SyGuS). A SyGuS problem is specified by a context-free grammar describing the search space of programs, and a logical formula describing the specification. Many synthesizers now support this format [ABJ+13] and annually compete in synthesis competitions [AFSSL16b]. Thanks to these competitions, these solvers are now quite mature and are finding wide application [HD17].

While the logical specification mechanism provided by SyGuS is powerful, it can only capture the functional requirements of the synthesis problem—e.g., the program should perform correctly on a given set of input/output examples. When multiple possible programs can satisfy the specification, SyGuS *does not* provide a way to prefer one to the other—e.g., one cannot ask a solver to return the program with the fewest if-statements. As a consequence, existing synthesis tools do not provide guarantees about what solution is returned if multiple ones exist.

While a few synthesizers have attempted to include some form of specification to express this kind of quantitative intents [BTGC16, SG15, NDAFH17, KRKK17], these approaches are domain-specific, do not apply to SyGuS problems, and do not provide a simple and flexible specification mechanism. The lack of a formal treatment of quantitative requirements stands in the way of designing synthesizers that can take advantage quantitative of objectives to perform more efficient forms of synthesis.

In this chapter, we propose QSyGuS, a unifying framework for describing syntax-guided synthesis problems with quantitative objectives over the syntax of the synthesized programs—e.g., find the most likely program with respect to a given probability distribution—and present an algorithm for solving synthesis problems expressed in this framework. We focus on syntactic objectives because they are the most common ones in practical applications of program synthesis. For example, in programming by examples, it is desirable to produce small programs with fewer constants because these programs are more likely to generalize to examples outside of the specification [Gul16]. QSyGuS extends SyGuS in two ways. First, in QSyGuS the search space is represented using weighted grammars, which augment context-free grammars with the ability to assign weights to programs. Second, QSyGuS allows the user to specify constraints over the weight of the program, including optimization objectives—e.g., find the program with the fewest if-statements and with the lowest depth.

QSyGuS is a natural, general, and flexible formalism and is grounded in the well-studied theory of weighted grammars. We leverage this theory and design an algorithm for solving QSyGuS problems using closure properties of weighted grammars. Given a QSyGuS problem, our algorithm generates a SyGuS problem that can be delegated to existing SyGuS solvers. The algorithm then iteratively refines the solution returned by the SyGuS solver to find an optimal one by further generating new SyGuS instances using weighted grammar operations. We implement our algorithm in a tool, QuaSi, and evaluate it on 26 quantitative extensions of existing SyGuS benchmarks. QuaSi can synthesize optimal solutions in 15/26 benchmarks with times comparable to those needed to find a solution that does

not need to satisfy any quantitative objective.

In this chapter, we introduce

- QSyGuS, a formal framework grounded in the theory of weighted grammars that can describe syntax-guided synthesis problems with quantitative objectives over the syntax of the synthesized programs. (§ 2.3)
- An algorithm for solving QSyGuS problems that leverages closure properties of weighted grammars and existing SyGuS solvers. (§ 2.4)
- QuaSi, a tool for specifying and solving QSyGuS problems that interfaces with existing SyGuS solvers and a comprehensive evaluation of QuaSi, which shows that QuaSi can efficiently solve QSyGuS problems over different types of weights, including additive weights, probabilities, and combinations of multiple weights. (§ 2.5)

2.2 Illustrative Example

In this section, we illustrate the main components of our framework using an example. We start with a Syntax-Guided Synthesis (SyGuS) problem in which no quantitative objective is provided. We recall that the goal of a SyGuS problem is to synthesize a function f of a given type that is accepted by a context-free grammar G, and such that $\forall x.\phi(f, x)$ holds (for a given Boolean constraint ϕ).

The following SyGuS problem asks to synthesize a function that is accepted by the following grammar and that computes the max of two numbers. The semantic

$$\begin{aligned} \text{Start} &::= \text{Start} + \text{Start} \mid \text{if}(\text{BExpr}) \text{ then Start else Start} \mid x \mid y \mid 0 \mid 1 \\ \text{BExpr} &::= \text{Start} > \text{Start} \mid \neg\text{BExpr} \mid \text{BExpr} \wedge \text{BExpr} \end{aligned}$$

Figure 2.1: Grammar of conditional linear terms.

constraint is given by the following formula.

$$\psi(f) \stackrel{\text{def}}{=} \forall x, y. f(x, y) \geqslant x \wedge f(x, y) \geqslant y \wedge (f(x, y) = x \vee f(x, y) = y)$$

The following three programs are semantically equivalent, but syntactically different solutions.

$$\begin{aligned} max_1(x, y) &= \text{if}(x > y) \text{ then } x \text{ else } y \\ max_2(x, y) &= \text{if}(x > y) \text{ then } (x + 0) \text{ else } (y + 0) \\ max_3(x, y) &= \text{if}(x > y) \text{ then } x \text{ else } (\text{if}(y > x) \text{ then } y \text{ else } x) \end{aligned}$$

All solutions are correct, but the user might, for example, prefer the smallest one. However, SyGuS does not provide ways to specify this quantitative intent.

Adding weights. In our formalism, QSyGuS, we augment context-free grammars to assign weights to programs in the search space. Concretely, we adopt weighted grammars [DKV09], a well-studied formalism with many desirable properties. In a weighted grammar, each production is assigned a weight. For example, the weighted grammar shown in Figure 2.2 extends the one from the previous SyGuS example to assign to each program p a pair of weights (w_1, w_2) where w_1 is the number of if-statements and w_2 is the number of plus operators in p. In this case, the weights are pairs of integers and the weight of a grammar derivation is the pairwise sum of all the weights of the productions involved in the derivation—e.g., the sum of (w_1, w_2) and (w_1', w_2') is $(w_1 + w_1', w_2 + w_2')$. In the figure, we write $/(w_1, w_2)$ to assign weight (w_1', w_2') to a production. We omit the weight for productions with cost $(0, 0)$. The functions max_1, max_2 and max_3 have weights $(1, 0)$, $(1, 2)$, and $(2, 0)$ respectively.

Adding and solving quantitative objectives. Once we have a way to assign weights to programs, QSyGuS allows the user to specify quantitative objectives over the weights of the productions—e.g., only allow solutions with fewer than 4 if-statements. In our example, we could require the solution to be minimal with respect to the number of if-statements, i.e., minimize the first component of the paired weight. With these constraints, both max_1 and max_2 would be considered optimal solutions because

$$
\begin{array}{rcl}
\text{Start} & ::= & \text{Start} + \text{Start}/(\mathbf{0},\mathbf{1}) \\
& \mid & \text{if}(\text{BExpr}) \text{ then Start else Start}/(\mathbf{1},\mathbf{0}) \\
& \mid & x \mid y \mid 0 \mid 1
\end{array}
\qquad
\begin{array}{rcl}
\text{BExpr} & ::= & \text{Start} > \text{Start} \\
& \mid & \neg\text{BExpr} \\
& \mid & \text{BExpr} \wedge \text{BExpr}
\end{array}
$$

Figure 2.2: Weighted grammar that assigns weight $(w_1, w_2) \in \mathsf{Nat} \times \mathsf{Nat}$ to a program where w_1 is the number of if-statements and w_2 is the number of plus-statements.

there exists no solution with 0 if-statements. If we require the solution to also be minimal with respect to the second component of the paired weight, max_1 will be *a possible* optimal solution.

Our tool QuaSi can automatically discover solutions in both of these cases. Let's consider the last minimization objective. In this case, QuaSi first uses existing SyGuS solvers to synthesize an initial solution using the non-weighted version of the grammar. Let's say that the returned solution is, for example, max_3 of weight $(2, 0)$. QuaSi uses this solution to build a new SyGuS instance that only accepts programs with at most one if-statement. Solving this SyGuS problem can, for example, result in the program max_2 of weight $(1, 2)$, which will require our solver to build yet another SyGuS instance. This approach is repeated and if it terminates, an optimal program is found.

2.3 SyGuS with Quantitative Objectives

In this section, we introduce our framework for defining syntax-guided synthesis problems with quantitative objectives over the syntax of the synthesized programs. We first provide preliminary definitions for notions such as semirings and weighted tree grammars, and then use these notions to augment SyGuS problems with quantitative objectives.

Weights over Semirings

We now define the universe of weights we will assign to programs. In general, weights are defined using monoids—i.e., sets equipped with an addition operator—but when a grammar is nondeterministic—i.e., it can produce the same term using multiple derivations—the same term might be assigned multiple weights. Hence, we choose to use semirings. Since we also care about optimization objectives, we assume all our semirings are equipped with a partial order.

Definition 2.1 (Semiring)**.** *A* (*ordered*) semiring *is a pair* $(\boldsymbol{S}, \preceq)$ *where*

1. $\boldsymbol{S} = (S, \oplus, \otimes, 0, 1)$ *is an algebra consisting of a commutative monoid* $(S, \oplus, 0)$ *and a monoid* $(S, \otimes, 1)$ *such that* $\otimes$ *distributes over* $\oplus$, $0 \neq 1$, *and, for every* $x \in S$, $x \otimes 0 = 0$,

2. $\preceq \ \subset S \times S$ *is a partial order over* S.

We often use the word semiring to refer to just the algebra $\mathbf{S}$.

Example 2.2. *In this dissertation, we focus on semirings with the following algebras.*

Boolean $\mathit{Bool} = (\mathbb{B}, \vee, \wedge, 0, 1)$. *This semiring only contains the values* `true` *and* `false` *and is used to represent non-quantitative problems.*

Tropical $\mathit{Trop} = (\mathbb{Z} \cup \{\infty\}, \min, +, \infty, 0)$. *This semiring is the most common one and is used to assign additive weights—e.g., term sizes and term depth.*

Probabilistic [1] $\mathit{Prob} = (\mathbb{R}, +, \cdot, 0, 1)$. *This semiring is used to assign probabilities to terms in a grammar.*

In our framework, we allow synthesis problems to have multiple objectives. Hence, we define a product operation to compose semirings. Intuitively, the following operation composes algebras of semirings to create a pair and applies the

[1] When the grammar of a synthesis problem is assigned cost only in the range $[0, 1]$ and is unambiguous, the weight of terms produced by the grammar is also in the range $[0, 1]$. And hence this semiring can be used to represented probabilities of programs.

operation of each algebra to the corresponding projections of the pair. Similarly, two orders can be composed to create an order over pairs of elements. We propose two such compositions, one that assigns equal weights to the two orders (Pareto) and one that prefers one order over the other (Sorted).

Definition 2.3 (Products)**.** *Given* $\boldsymbol{S}_1 = (S_1, \oplus_1, \otimes_1, 0_1, 1_1)$ *and* $\boldsymbol{S}_2 = (S_2, \oplus_2, \otimes_2, 0_2, 1_2)$*, the* product algebra *is the tuple* $\boldsymbol{S}_1 \times_S \boldsymbol{S}_2 = (S_1 \times S_2, \oplus, \otimes, (0_1, 0_2), (1_1, 1_2))$ *such that for every* $x_1, x_2 \in S_1$ *and* $y_1, y_2 \in S_2$*, we have* $(x_1, y_1) \oplus (x_2, y_2) \stackrel{\text{def}}{=} (x_1 \oplus_1 x_2, y_1 \oplus_2 y_2)$ *and* $(x_1, y_1) \otimes (x_2, y_2) \stackrel{\text{def}}{=} (x_1 \otimes_1 x_2, y_1 \otimes_2 y_2)$*.*

Given two partial orders $\preceq_1 \subset S_1 \times S_1$ *and* $\preceq_2 \subset S_2 \times S_2$*, the* Pareto product *of the two orders is defined as the partial order* $\preceq_p = \textsc{par}(\preceq_1, \preceq_2) \subseteq (S_1 \times S_2) \times (S_1 \times S_2)$ *such that, for every* $x_1, x_2 \in S_1$ *and* $y_1, y_2 \in S_2$*, we have* $(x_1, y_1) \preceq_p (x_2, y_2)$ *iff* $x_1 \preceq_1 x_2$ *and* $y_1 \preceq_2 y_2$*.*

Given two partial orders $\preceq_1 \subset S_1 \times S_1$ *and* $\preceq_2 \subset S_2 \times S_2$*, the* Sorted product *of the two orders is defined as the partial order* $\preceq_s = \textsc{sort}(\preceq_1, \preceq_2) \subseteq (S_1 \times S_2) \times (S_1 \times S_2)$ *such that, for every* $x_1, x_2 \in S_1$ *and* $y_1, y_2 \in S_2$*, we have* $(x_1, y_1) \preceq_s (x_2, y_2)$ *iff* $x_1 \preceq_1 x_2$ *or* $(x_1 = x_2$ *and* $y_1 \preceq_2 y_2)$*.*

Example 2.4. *The weights in the grammar in Figure 2.2 are from the product semiring* $\mathit{Trop} \times_S \mathit{Trop}$*. When using the Pareto partial orders, we have, for example,* $(1,0) \preceq (2,0)$ *and* $(1,0) \preceq (1,2)$*, but* $(1,2)$ *is incomparable to* $(2,0)$*. When using the Sorted product, we have, for example,* $(1,0) \preceq (1,2) \preceq (2,0)$*.*

Weighted Tree Grammars

Since SyGuS defines search spaces using context-free grammars, we propose to extend this formalism with weights to assign costs to terms in the grammar. We focus our attention on a restricted class of context-free grammars called regular tree grammars—i.e., grammars generating regular tree languages—because, to our knowledge, the benchmarks appearing in the SyGuS competition [AFSSL16a] and in practical applications of SyGuS operate over tree grammars. Moreover, it

was recently shown that SyGuS problems that are undecidable for context-free grammars become decidable with weighted tree grammars [CRST15].

Trees. A *ranked alphabet* is a tuple $(\Sigma, \mathrm{rk}_\Sigma)$ where Σ is a finite set of symbols and $\mathrm{rk}_\Sigma : \Sigma \to \mathbb{N}$ associates a rank to each symbol. For every $m \geqslant 0$, the set of all symbols in Σ with rank m is denoted by $\Sigma^{(m)}$. In our examples, a ranked alphabet is specified by showing the set Σ and attaching the respective rank to every symbol as a superscript—e.g., $\Sigma = \{+^{(2)}, c^{(0)}\}$. We use T_Σ to denote the set of all (ranked) trees over Σ—i.e., T_Σ is the smallest set such that (i) $\Sigma^{(0)} \subseteq T_\Sigma$, (ii) if $\sigma \in \Sigma^{(k)}$ and $t_1, \ldots, t_k \in T_\Sigma$, then $\sigma(t_1, \cdots, t_k) \in T_\Sigma$. In the following, we assume a fixed ranked alphabet $(\Sigma, \mathrm{rk}_\Sigma)$.

Weighted Tree Grammars. Tree grammars are similar to word grammars but they generate ranked trees instead of words. Weighted tree grammars augment tree grammars by assigning to each tree a weight from a semiring. They do so by associating weights to productions in the grammar. Weighted grammars can, for example, compute the height of a tree, the number of occurrences of some node in the tree, or the probability of a tree with respect to some distribution. In the following, we assume a fixed semiring $(\mathbf{S}, \preceq)$ where $\mathbf{S} = (S, \oplus, \otimes, 0, 1)$.

Definition 2.5 (Weighted Tree Grammar)**.** *A* weighted tree grammar (*WTG*) *is a tuple* $G = (N, Z, P, \mu)$*, where* N *is a set of non-terminal symbols with arity 0,* Z *is an axiom with* $Z \in N$*,* P *is a set of production rules of the form* $A \to \beta$ *where* $A \in N$ *is a non-terminal and* β *is a tree of* $T(\Sigma \cup N)$*, and* $\mu : P \to S$ *is a function assigning to each production a weight from the semiring.*

We can now define the semantics of a WTG as a function $w_G : T_\Sigma \mapsto S$, which assigns a weight to each tree. Intuitively, the weight of a tree is $\oplus$-sum of the weight of every possible derivation of that tree in the grammar, and the weight of a derivation is the $\otimes$-product of the weights of the productions appearing in the derivation. We use $MS(\beta) = \langle X_1, \ldots, X_k \rangle$ to denote the multi-set of all nonterminals appearing in β and $\beta[t_1/X_1, \ldots, t_k/X_k]$ to denote the result of simultaneously substituting each X_i with t_i in β. Given a derivation $p = A \to \beta$ such that $MS(\beta) = \langle X_1, \ldots, X_k \rangle$, we assume that p is a symbol of arity k. A derivation d starting at non-terminal

X is a tree of productions $d \in T(P)$ representing one possible way to derive a tree starting from X. The derivation has to be such that: (i) the root of d is a production of the form $X \to \beta$, (ii) for every node $p = A \to \beta$ in d, if $MS(\beta) = \langle X_1, \ldots, X_k \rangle$, then, for every $1 \leqslant i \leqslant k$, the i-th child of p is a production $X_i \to \beta_i$. Given a derivation d with root $p = X \to \beta$, such that $MS(\beta) = \langle X_1, \ldots, X_k \rangle$ and p has children subtrees $d_1, \ldots, d_k$, the tree generated by d is recursively defined as $tree(d) = \beta[tree(d_1)/X_1, \ldots, tree(d_k)/X_k]$. We use $\textsc{der}(X, t)$ to denote the set of all derivations d starting at X, such that $tree(d) = t$. The weight $\textsc{dw}(d)$ of a derivation d is the $\otimes$-product of the weights of the productions appearing in the derivation. Finally, the weight of a tree t is the $\oplus$-sum of the weights of all the derivations of t from the initial nonterminal $w_G(t) = \bigoplus_{d \in \textsc{der}(Z,t)} \textsc{dw}(d)$. A weighted tree grammar is *unambiguous* iff, for every $t \in T_\Sigma$, there exists at most one derivation—i.e., $|\textsc{der}(Z, t)| \leqslant 1$.

Weighted tree grammars generalize weighted tree automata. In particular, a *weighted tree automaton* (WTA) is a WTG in which every production is of the form $A \to \sigma(T_1, \ldots, T_n)$, where $A \in N$, each $T_i \in N$, and $\sigma \in \Sigma^{(n)}$. Finally, a *tree automaton* (TA) is a WTA over the Boolean semiring—i.e., the TA accepts all trees with some derivations yielding $\mathtt{true}$. Similarly, a *tree grammar* (TG) is a WTG over the Boolean semiring. Given a TA (resp. TG) G, we use $L(G)$ to denote the set of trees accepted by G—i.e., $L(G) = \{t \mid w_G(t) = \mathtt{true}\}$.

Example 2.6. *The weighted grammar in Fig. 2.2 operates over the semiring* $\mathit{Trop} \times \mathit{Trop}$, $N = \{\mathit{Start}, \mathit{BExpr}\}$, $Z = \mathit{Start}$, P *contains 9 productions, and* μ *assigns non-zero weights to two of the productions.*

Aside from being a natural formalism for assigning weights to trees, TGs and WTGs enjoy properties that make them a good choice for our model. First, WTGs (resp. TGs) are equi-expressive to WTAs (resp. TAs) and have logical characterizations [CDG+07, DV06, DKV09]. Due to this reason, tree grammars are closed under Boolean operations and enjoy decidable equivalence [CDG+07]. Second, WTGs enjoy many closure and decidability properties—e.g., given two WTGs G_1 and G_2, we can compute the grammars $G_1 \oplus G_2$ and $G_1 \otimes G_2$ such that, for every f,

$w_{G_1 \oplus G_2}(f) = w_{G_1}(f) \oplus w_{G_2}(f)$ and $w_{G_1 \otimes G_2}(f) = w_{G_1}(f) \otimes w_{G_2}(f)$. This operation is convenient for building grammars over product semirings.

QSyGuS

In this section, we formally define QSyGuS, which extends SyGuS with quantitative objectives. In SyGuS a problem is specified with respect to a background theory T—e.g., linear arithmetic—and the goal is to synthesize a function f that satisfies two constraints provided by the user. The first constraint describes a *functional semantic property* that f should satisfy and is given as a predicate $\psi(f) \stackrel{\text{def}}{=} \forall x.\phi(f, x)$. The second constraint limits the *search space* S of f and is given as a set of expressions specified by a context-free grammar G defining a subset of all the terms in T. A solution to the SyGuS problem is an expression e in S such that the formula $\psi(e)$ is valid.

We augment such a framework in two ways. First, we replace context free grammars with WTGs, which we use to assign weights (from a given semiring) to terms. Second, we augment the problem formulation with constraints over the weight of the synthesized program—i.e., only consider programs of weight greater than 2—and optimization objectives over the same weight—i.e., find the solution of minimal weight. Weight constraints range over the grammar

$$WC := WC \wedge WC \mid WC \vee WC \mid \neg WC \mid w \preceq s \mid s \preceq w \mid w \prec s \mid s \prec w,$$

where w is a special variable and s is an element of the semiring under consideration. Given a constraint $\omega \in WC$, we write $\omega(t)$ to denote the term obtained by replacing w with t in ω.

Definition 2.7 (QSyGuS)**.** *A QSyGuS problem is a tuple* $(T, (S, \preceq), \psi(f), G, \omega, \textit{OPT})$ *where:*

- T *is a background theory.*
- $(S, \preceq)$ *is an ordered semiring defining the set of weights and their operations.*

- G *is a weighted tree grammar with weights over the semiring* $\mathbf{S}$ *and that only contains terms in* T*—i.e.,* $L(G) \subseteq T$.
- $\psi(f) \stackrel{\text{def}}{=} \forall x.\phi(f, x)$ *is a Boolean formula constraining the semantic behavior of the synthesized program* f.
- $\omega \in WC$ *is a set of constraints over the weight* w *of the synthesized program.*
- $\textsc{opt}$ *is a Boolean denoting whether the solution has to have minimal weight with respect to* $\preceq$.

A solution to the QSyGuS problem is a term e *such that* $e \in L(G)$, $\psi(e)$ *is true, and* $\omega(w_G(e))$ *is true. If* $\textsc{opt}$ *is true, we also require that there is no* g *that satisfies the previous conditions such that* $\omega(w_G(g)) \prec \omega(w_G(e))$.

A SyGuS problem is a QSyGuS problem without weight constraints—i.e., $\omega \equiv \texttt{true}$ and $\textsc{opt} = \texttt{false}$. We denote such problems just as triples $(T, \psi(f), G)$.

Example 2.8. *Consider the QSyGuS problem described in Section. 2.2. We already described all the components except* ω *and* $\textsc{opt}$ *earlier in this section. In this example,* $\omega = \texttt{true}$ *and* $\textsc{opt} = \texttt{true}$ *because we want to synthesize the solution with minimal weight.*

2.4 Solving QSyGuS Problems via Grammar Reduction

In this section, we present an algorithm for solving QSyGuS problems (Algorithm 1), which works as follows. First, given a QSyGuS problem, we construct (under certain assumptions) a SyGuS problem for which the solution is guaranteed to satisfy the weight constraints ω (line 2) and use existing SyGuS solvers to find a solution to such a problem (line 3). If the QSyGuS problem requires minimization, our algorithm produces a new SyGuS instance to search for a solution that is better than the previously found one and tries to solve it (lines 6-7). This procedure is repeated until an optimal solution is found (line 8).

Function: QSyGuS-solve$(T, \mathbf{S}, \psi, G, \omega, \text{opt})$
$G' \leftarrow$ ReduceGrammar(G, ω) // extract grammar satisfying ω;
$f^* \leftarrow$ SyGuS(T, ψ, G') // solve corresponding SyGuS problem;
if $\text{opt} = \texttt{false}$ **then**
  return f^*
end
while *true* **do**
  $G' \leftarrow$ ReduceGrammar$(G', w \prec w_G(f^*))$;
  $f \leftarrow$ SyGuS(T, ψ, G') // Try to find better solution;
  if $f = \bot$ **then**
    return f^* // Return the optimal solution
  end
  $f^* \leftarrow f$;
end

Algorithm 1: QSyGuS synthesis algorithm

From QSyGuS to SyGuS

The first step of our algorithm is to construct a SyGuS problem characterizing exactly all the solutions of the QSyGuS problem that satisfy the weight constraints. Given a QSyGuS problem $P = (T, (\mathbf{S}, \preceq), \psi(f), G, \omega, \text{opt})$, we construct a SyGuS problem $P' = (T, \psi(f), G')$ such that a function g is a solution to the SyGuS problem P' iff g is a solution of $P = (T, (\mathbf{S}, \preceq), \psi(f), G, \omega, \texttt{false})$, where the optimization constraint has been dropped. We denote the grammar-reduction operation as $G' \leftarrow$ ReduceGrammar(G, ω).

Base case. First we show how to solve the problem when ω is an atomic formula—i.e. of the form $w \preceq s$, $s \preceq w$, $w \prec s$, or $s \prec w$. We start by showing how to solve the problem for $w \preceq s$ as the construction is identical for the other constraints.

Concretely, we are given a WTG $G = (N, Z, P, \mu)$ and we want to construct a TG $G_{\preceq s} = (N', Z', P')$ such that $t \in L(G_{\preceq s})$ iff $w_G(t) \preceq s$. In general, it is not possible to perform this construction for arbitrary semirings and grammars. We first present our algorithm and then describe sufficient conditions under which we can ensure termination and correctness.

The idea behind our construction is to introduce new nonterminals in the grammar $G_{\preceq s}$ to keep track of the weight of the trees that can be produced from those

nonterminals. For example, a nonterminal pair (X, s') will derive all trees derivable from X using a single derivation of weight s'. Therefore, the set of nonterminals N' is a subset of $N \times S$ (plus an initial nonterminal Z'), where S is the universe of the WTG's semiring. We construct our set of nonterminals N' starting from the leaf productions of G and then recursively explore other productions. At the same time we generate the set of productions P'. Formally, N' and P' are the smallest sets such that the following conditions hold.

1. $Z' \in N'$ (the initial nonterminal).

2. For every production $p \in P$ such that $p = (A \to \beta)$ and $\beta \in T_\Sigma$—i.e., p is a leaf—and $\mu(p) \preceq s$, then $(A, \mu(p)) \in N'$ and $((A, \mu(p)) \to \beta) \in P'$. If $A = Z$, then $Z' \to (A, \mu(p)) \in P'$.

3. For every production $p \in P$ such that $p = (A \to \beta)$, $MS(\beta) = \langle X_1, \ldots, X_k \rangle$, $(X_1, s_1), \ldots, (X_k, s_k) \in N'$ (for some values $s_i \in S$), and $\mu(p) \otimes s_1 \otimes \ldots \otimes s_k = s'$, $s' \preceq s$, then $(A, s') \in N'$, and $((A, s') \to \beta[(X_1, s_1)/X_1, \ldots, (X_k, s_k)/X_k]) \in P'$. If $A = Z$, then $Z' \to (A, s') \in P'$.

Example 2.9. *We illustrate our construction using the grammar in Figure 2.2 . Assume the weight constraint is $w \preceq (1, 0)$ and the partial order is built using a Pareto product—i.e., we accept terms with 1 or less if-statements and no plus-statements. Our construction yields the following grammar.*

$$
\begin{aligned}
Z' &::= (\textit{Start,1,0}) \mid (\textit{Start,0,0}) \\
(\textit{Start,1,0}) &::= \textit{if}((\textit{BExpr,0,0}))\ \textit{then}\ (\textit{Start,0,0})\ \textit{else}\ (\textit{Start,0,0}) \mid x \mid y \mid 0 \mid 1 \\
(\textit{Start,0,0}) &::= x \mid y \mid 0 \mid 1 \\
(\textit{BExpr,0,0}) &::= (\textit{Start,0,0}) > (\textit{Start,0,0}) \mid \neg(\textit{BExpr,0,0}) \mid (\textit{BExpr,0,0}) \wedge (\textit{BExpr,0,0})
\end{aligned}
$$

The construction of $G_{\preceq s}$ only terminates for certain semirings and grammars, and only guarantees that individual derivations yield the correct cost—i.e., it does not account for the $\oplus$-sum of multiple derivations.

Example 2.10. *The following WTG over* Prob *is ambiguous and, if we apply the grammar reduction algorithm for* $\omega := w \preceq 0.6$*, the resulting grammar will be empty. However, the tree* $1 + 1$ *has weight* $0.9 \preceq 0.6$ $(0.9 \succcurlyeq 0.6)$.

$$\begin{aligned} \mathit{Start} ::= {} & \mathit{Start} + \mathit{Start}/0.5 \\ & \mid x \mid 0 \mid 1 \mid \mathit{Expr} \end{aligned} \qquad \begin{aligned} \mathit{Expr} ::= {} & \mathit{Expr} + \mathit{Expr}/0.4 \\ & \mid x \mid 0 \mid 1 \end{aligned}$$

We now identify sufficient conditions under which the construction of $G_{\preceq s}$ terminates and is sound. In particular, we start by restricting our attention to unambiguous WTGs, which are the common ones in practice. We use $\textsc{weights}(G) = \{s \mid p \in P \wedge \mu(p) = s\}$ to denote the set of weights used by G and $M_{\mathbf{S},G} = (S', \otimes, 1)$ to denote the submonoid of $\mathbf{S}$ generated by $\textsc{weights}(G)$—i.e., the set of all weights we can generate using $\otimes$ and $\textsc{weights}(G)$.

Theorem 2.11. *Given an unambiguous WTG* G *over a semiring* $\mathbf{S}$ *such that* $M_{S,G} = (S', \otimes, 1)$*, and a weight* $s \in S$*, the construction of* $G_{\preceq s}$ *terminates if the set* $\{s' \mid s' \preceq s \wedge w \in S'\}$ *is finite. Moreover, if the set of weights* $\textsc{weights}(G)$ *is monotonically increasing with respect to* $\preceq$*—i.e. for every* $s \in S$ *and* $s' \in \textsc{weights}(G)$*,* $s \preceq s \otimes s'$*—then* $L(G_{\preceq s})$ *contains exactly every tree* t *such that* $w_G(t) \preceq s$*.*

Proof. We first show that each step of the algorithm terminates. Steps 1 and 2 terminate since the grammar G is finite. Step 3 only produces nonterminals that belongs to $N \times \{s' \mid s' \preceq s \wedge w \in S'\}$ which is also finite.

We now prove soundness. It is straightforward to prove the following claim by induction: for every nonterminal $(A, s) \in N'$, tree $t \in T_\Sigma$, we have that $d \in \textsc{der}((A, s'), t)$ iff there exists a derivation $d' \in \textsc{der}(A, t)$ such that $\textsc{dw}(d') = s'$ and $s' \preceq s$. Because G is unambiguous, every tree has at most one derivation. Therefore $\textsc{dw}(d') = w_G(t)$ and $w_G(t) \preceq s$.

□

The theorem above also holds for other atomic constraints $w \prec s$, $s \preceq w$, or $s \prec w$ (for these last two, the direction of the monotonicity is reversed). Moreover, in certain cases, even if the construction may not terminate for, let's say $s \preceq w$, it might terminate for the negated constraint $w \prec s$. In such a case, we can use the

closure properties of regular tree grammars/automata to construct the reduced grammar for $s \preceq w$ as $G_{\preceq w} = \textsc{intersect}(G, \textsc{complement}(G_{\succ w}))$. The same idea can be applied to all atomic constraints.

In practice, the restriction of Theorem 2.11 holds for grammars that operate over the Boolean and probabilistic semirings, and the tropical semiring only with positive weights. Theorem 2.11 never holds when **S** is the tropical semiring and the grammar contains negative weights. In general, one cannot construct the constrained grammar in this case. However, it is easy to modify our algorithm to work with grammars that do not contain loops—i.e., derivations from a nonterminal to a tree containing the same nonterminal—with negative weights.

Intuitively, when the grammar contains no negative loops, we can find a constant SH such that any intermediate derivation with weight greater than $s + SH$ will never result in tree with weight smaller than s. We use this idea to modify the construction of $G^{\mathsf{Trop}}_{\leqslant s}$—i.e., $G_{\leqslant s}$ for Trop—as follows. First, this constant is bounded by ck^{n+1} where c is the absolute value of the smallest negative weight in the grammar, k is the largest number of nonterminals appearing in one grammar production, and $n = |N|$ is the number of nonterminals. Second, in steps 2 and 3 of the construction, a new nonterminal and the corresponding productions are produced if $\mu(p) \leqslant s + |SH|$ (previously $\mu(p) \leqslant s$). However, if $A = Z$ in steps 2 and 3, we add a new production $Z' \to (A, s')$ only if $s' \preceq s$.

We now show when this construction terminates and return correct values. Since the tropical semiring combines multiple runs using the min operator, we can *drop* the requirement that the grammar has to be unambiguous.

Theorem 2.12. *Given a WTG* G *over* Trop *and a weight* $s \in \mathbb{Z}$*, the construction of* $G^{\mathsf{Trop}}_{\leqslant s}$ *terminates if* G *contains no loop with cumulative negative weight. Moreover,* $G^{\mathsf{Trop}}_{\leqslant s}$ *contains exactly every tree* t *such that* $w_G(t) \leqslant s$*.*

Proof. First, we show that any tree with weight $\leqslant s$ must be accepted by $G^{\mathsf{Trop}}_{\leqslant s}$. We do so by showing that if a tree t is not accepted by $G^{\mathsf{Trop}}_{\leqslant s}$—i.e., t has some subtree β with weight greater than $s + SH$—the weight of t must be greater than s. Note that the modified algorithm can track weights $\leqslant s + |SH|$ in the intermediate nonterminals

but still accept only trees with weight $\leqslant s$. According to the definition, the weight of t is the sum of all rules used to derived t, that is, $w_G(t) = w_G(\beta) + w_G(t[B/\beta])$ where $t[B/\beta] \in T_{\Sigma\cup\{B\}}$ is the result of substituting the node corresponding to β with B. Then if $t[B/\beta]$ contains loops, we can eliminate all loops from it to get a tree $t'[B/\beta]$ such that $w_G(t[B/\beta]) \geqslant w_G(t'[B/\beta])$ because loops have non-negative weights. If $t[B/\beta]$ contains no loop, its weight $w_G(t[B/\beta]) \geqslant -ck^{n+1} = -SH$ since the size of $t[B/\beta]$ is no more than k^{n+1} (the height of $t[B/\beta]$ is no more than n since there is no loop in it) and each production used to derive $t[B/\beta]$ has weight greater than $-c$. Therefore, using the fact that $w_G(t[B/\beta]) \geqslant -SH$, we have that the weight of t is $w_G(\beta) + w_G(t[B/\beta]) > s + SH + w_G(t[B/\beta]) \geqslant s$.

Now, we show that the algorithm terminates. We observe that there is only a finite number of trees without loops so the set of their weights is also finite, namely the minimum weight of any tree without loops is w^*. On the other hand, for any tree t containing loops, the weight w of t must be greater or equal to the weight of some tree without loops—i.e., $w > w^*$. This is because we can eliminate loops, whose weights are non-negative, from t and the resulting tree has a weight greater than or equal to the weight of t. So the weights of nonterminal produced by our constructions all fall in the range $[w^*, s + SH]$, which is finite. Finally, the construction only needs to consider a finite number of nonterminals, and will always terminate.

□

Composing semirings. We next discuss how Theorem 2.11 relates to product semirings. Given a grammar $G = (N, Z, P, \mu)$ over a semiring $\mathbf{S}_1 \times_{\mathbf{S}} \mathbf{S}_2$, we use $G^{\mathbf{S}_i}$ to denote the grammar (N, Z, P, μ_i) in which the weight function outputs the corresponding projected weight—i.e., if $\mu(p) = (s_1, s_2)$, then $\mu_i(p) = s_i$.

Let's first consider the case where the product semiring uses a Pareto partial order. In this case, if Theorem 2.11 holds for each grammar $G^{\mathbf{S}_i}$ and $w_i \preceq_i s_i$, then it holds for G and $(w_1, w_2) \preceq_p (s_1, s_2)$. However, the other direction is not true. Theorem 3 proves this intuition and states that, in some sense, solving Pareto partial orders is easier than solving the individual partial orders.

Theorem 3. *Given an unambiguous WTG* G *over the semiring* $\boldsymbol{S} = \boldsymbol{S}_1 \times_S \boldsymbol{S}_2$ *with Pareto partial order* $\preceq_p = \textsc{par}(\preceq_1, \preceq_2)$ *and a weight* $s = (s_1, s_2) \in S$*, if the constructions* $G^{\mathbf{S}_1}_{\preceq_1 s_1}$ *and* $G^{\mathbf{S}_2}_{\preceq_2 s_2}$ *terminate, then the construction of* $G_{\preceq s}$ *terminates.*

Proof. We first show by induction that if a nonterminal (X, w_1, w_2) is produced in the construction of $G^{\mathbf{S}}_{\preceq s}$, the nonterminals (X, w_1) and (X, w_2) must be produced in the construction of $G^{\mathbf{S}_1}_{\preceq s_1}$ and $G^{\mathbf{S}_1}_{\preceq s_1}$ respectively. For the base case, we consider the nonterminals (X, w_1, w_2) produced in step 2 with production p. The condition $\mu(p) \preceq s$ in step 2 implies that $\mu_1(p) \preceq s_1$ and $\mu_2(p) \preceq s_2$, which means that (X, w_1) and (X, w_2) are also produced in the construction of the corresponding grammar. Then, for every nonterminal (X, w_1, w_2) produced in step 3 with rule p and $\{(X_i, w_i^{(1)}, w_i^{(2)})\}_i \subseteq N'$, where $(w_1, w_2) := \mu(p) \otimes \bigotimes_i (w_i^{(1)}, w_i^{(2)}) \preceq (s_1, s_2)$, according to the induction hypothesis, nonterminals in $\{(X_i, w_1^i)\}_i$ and $\{(X_i, w_2^i)\}_i$ are already produced in the grammars $G^{\mathbf{S}_1}_{\preceq s_1}$ and $G^{\mathbf{S}_2}_{\preceq s_2}$. Therefore, we can apply step 3 with p and nonterminals $G^{\mathbf{S}_1}_{\preceq s_1}$ (or $G^{\mathbf{S}_2}_{\preceq s_2}$) to produce a new nonterminal (X, w_1) (or (X, w_2)). Note that $w_1 = \mu(p) \otimes \bigotimes_i w_i^{(1)} \preceq s_1$, and $w_2 = \mu(p) \otimes \bigotimes_i w_i^{(2)} \preceq s_2$,.

Since both of the constructions of $G^{\mathbf{S}_1}_{\preceq_1 s_1}$ and $G^{\mathbf{S}_2}_{\preceq_2 s_2}$ terminate, the number of nonterminals they produce, namely n_1 and n_2, must be finite. We have shown that the number of nonterminals produced in $G^{\mathbf{S}}_{\preceq s}$ is less than $n_1 \times n_2$, which is also finite. Finally, the construction of $G^{\mathbf{S}}_{\preceq s}$ terminates.

□

When we move to a sorted partial order, we cannot get an analogous theorem: if Theorem 2.11 holds for each grammar $G^{\mathbf{S}_i}$ and $w_i \preceq_i s_i$, then it does not necessary hold for G and $(w_1, w_2) \preceq_s (s_1, s_2)$. In particular, if the semiring $\mathbf{S}_2$ is infinite and there exists an $s_1' \prec s_1$, there will be infinitely many elements $(s_1', \_) \prec (s_1, s_2)$. Using this observation, we devise a modified algorithm for reducing grammars with sorted objectives. First, we compute the grammars $G^{\mathbf{S}_1}_{\prec_1 s_1}$, $G^{\mathbf{S}_1}_{=s_1}$, and $G^{\mathbf{S}_2}_{\prec_2 s_2}$. Second, we use WTG closure properties to compute $G_{\preceq_s}(s_1, s_2)$ as the union of $G^{\mathbf{S}_1}_{\prec_1 s_1}$ and $\textsc{intersect}(G^{\mathbf{S}_1}_{=s_1}, G^{\mathbf{S}_2}_{\prec_2 s_2})$.

General formulas. We can now inductively construct the grammar accepting only terms satisfying all constraints in ω. We again use the fact that tree grammars are

closed under Boolean operations to compute intersections and unions, and correctly characterize all conjunctions and unions appearing in the formula.

Finding an Optimal Solution

If our QSyGuS problem does not require minimization—i.e., $\text{OPT} = \texttt{false}$—the technique presented in Section 2.4 can be used to generate an equivalent SyGuS problem $P' = (T, \psi(f), G')$, which can be solved using off-the-shelf SyGuS solvers. In this section, we show how to extend this technique to handle minimization objectives. Our idea is to use SyGuS solvers to find a non-optimal solution for P' and then iteratively refine our grammar G' to search for a better solution. This loop is illustrated in Algorithm 1 (lines 5-9). Given the initial solution f^* to P' such that $w_G(f^*) = s$, we can construct a new grammar $G_{\prec s}$ and look for a solution with lower weight. If the SyGuS solver we use is sound—i.e., if it can find a solution if it exists—and complete—i.e., it can detect if a solution does not exist—Algorithm 1 terminates with an optimal solution.

In general, the above conditions are too strict and in practice this implies that the algorithm will often not terminate. However, if the SyGuS solver is sound, Algorithm 1 will eventually find the optimal solution, but it will not be able to prove that no smaller one exists. In our experiments, we will show that this approach can yield better solutions than those given by vanilla SyGuS solvers, even when Algorithm 1 does not terminate.

2.5 Implementation and Evaluation

First, we extended the SyGuS format with new syntax for expressing QSyGuS problems. Our format supports all semirings presented in Section 2.3, as well as additional ones. The format also allows creating tuples of semirings using the product operation described in Section 2.3. We augment the original SyGuS syntax to support weights on grammar productions. Weight constraints are added using an SMT-like syntax.

Second, we implemented Algorithm 1 in a tool called QuaSi. QuaSi already interfaces with three SyGuS solvers: CVC4 [BCD+11], ESolver [AFSSL16b], and EUSolver [ARU17b]. QuaSi supports all the semirings allowed in our format and implements a library for tree automata/grammars and weighted tree automata/-grammars operations, as well as several optimizations we did not discuss in the dissertation. In particular, QuaSi often uses simple grammar-reduction techniques to simplify the generated grammars, remove unnecessary productions, and consolidate equivalent ones.

We evaluate QuaSi through the following questions (experiments performed on an Intel Core i7 4.00GHz CPU with 32GB/RAM).

Q1 Can QuaSi solve quantitative variants of real SyGuS benchmarks? (§ 2.5)

Q2 What is the overhead of synthesizing optimal solutions? (§ 2.5)

Q3 How do multiple iterations of Alg. 1 affect the solution's weight? (§ 2.5)

Q4 Can QuaSi solve QSyGuS problems with multiple objectives? (§ 2.5)

Benchmarks. We perform our evaluation on 26 quantitative extensions of existing SyGuS competition benchmarks taken from 4 SyGuS benchmark tracks [AFSSL16b]: Hackers Delight, Integers, ICFP and Bitvector. 18 of our benchmarks only use a minimization objective over a single semiring (Table 2.1), while 8 use a minimization objective (Pareto or Sorted) over a product semiring (Table 2.2). We select SyGuS benchmarks using the following criteria: (*i*) the benchmark can be solved by either CVC4 [BCD+11] or ESolver [AFSSL16b], and (*ii*) the solution is not optimal according to some reasonable metric—e.g., size or number of if statements. The following are the full description of benchmarks presented in Table 2.1.

`max_ite(a,b)` These benchmarks extend the SyGuS benchmark `max2` by restricting the number of if statements between `a` and `b` and then require minimizing the total size of the solution. These benchmarks operate over the semiring $\mathsf{Trop} \times_{\mathsf{S}} \mathsf{Trop}$, but only impose one minimization objective.

`parity_not` minimizes the number of `not` operators in corresponding SyGuS benchmark.

Table 2.1: Performance of QuaSi. **T**ime shows the sequence of times taken to solve individual iterations of Alg. 1. **L**argest is the size of the largest SyGuS sub-problem. **G**rammar Size is the number of rules in the original grammar.

	Problem	**C**VC4		**E**Solver		**G**rammar
		Time[sec]	**L**argest	**T**ime[sec]	**L**argest	**S**ize
Trop	`max_ite(2,3)`	0.1+0.1	42	0.1	42	13
	`max_ite(2,15)`	0.1+0.1	239	0.3	239	13
	`max_ite(3,15)`	0.1+0.1+0.1	238	OOM	238	13
	`max_ite(10,15)`	0.5+0.5+0.9	226	OOM	226	13
	`parity_not`	0.1+TO	301	26.9+TO	43	6
	`max3_ite`	0.1+TO	31	OOM	-	14
	`array_search_3`	0.1+TO	135	TO	-	15
	`array_search_5`	0.1+TO	108	TO	-	16
	`hackers_5`	0.1+0.1	27	0.1+0.1+0.1	35	13
	`hackers_7`	0.1+0.3	35	0.1+0.1+0.2	41	13
	`hackers_17`	0.1+0.7	41	2.8+3.0+1.0	62	13
	`hackers_19`	0.2+TO	174	TO	-	13
	`icfp_7`	0.2+TO	146	TO	-	11
	`LinExpr_eq1ex`	0.7+TO	1717	TO	-	14
Prob	`hackers_2_prob`	0.6+4.1+0.1	95	0.8+0.1+0.2	154	13
	`hackers_5_prob`	0.1+0.9+0.1	96	0.1+0.2+0.1	154	13
	`hackers_7_prob`	0.1+TO	162	0.1+0.1+0.2	212	13
	`hackers_17_prob`	0.1+TO	187	3.4+6.5+OOM	291	13

`max3_ite` minimizes the number of `ite` operators in corresponding SyGuS benchmark.

The remainder of the Tropbenchmarks minimizes the size of the solution in the corresponding SyGuS benchmark.

`hackers_a_prob` Probabilistic extensions of corresponding SyGuS benchmarks (Prob semiring). The probability scheme we use assigns probability $\frac{1}{8}$ to shift operators, probability $\frac{1}{4}$ to arithmetic operators and $\frac{1}{2}$ to logical operators. The goal is to find the most probable solution.

Effectiveness of QSyGuS Solver

We evaluate the effectiveness of QuaSi on the 18 single-minimization-objective benchmarks. For each benchmark, we run QuaSi using either CVC4 or ESolver as

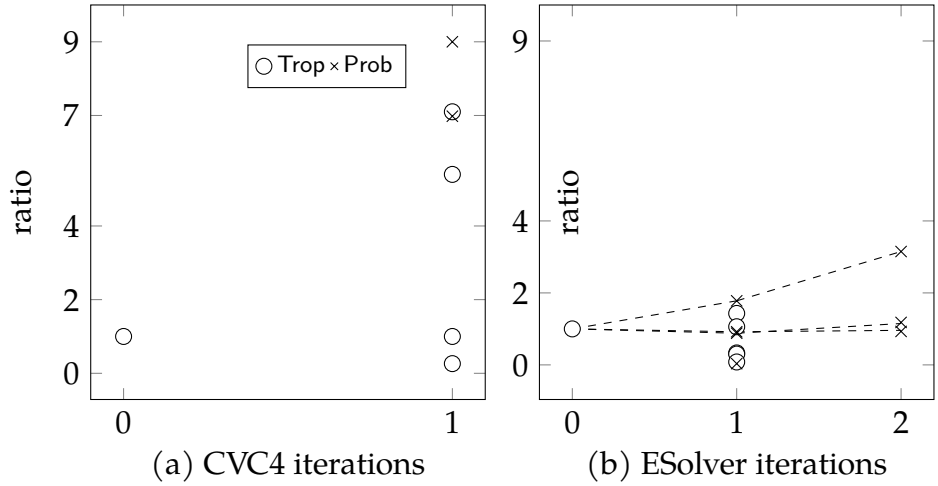
the backend SyGuS solver (we also evaluated QuaSi using EUSolver [ARU17b], but, due to its poor performance, we do not report the results). The results are shown in Table 2.1. The timeout for each iteration of Alg. 1 is 10 minutes.

With CVC4, QuaSi terminates with an optimal solution for 9/18 benchmarks, taking less than 5 seconds (avg: 0.7s) to solve each sub-problem. In 3 of these cases, the initial solution is already optimal and the second iteration is used to prove optimality. With ESolver, QuaSi terminates with an optimal solution for 8/18 benchmarks, taking less than 7 seconds (avg: 0.9s) to solve each sub-problem. In 2 cases, it can find a better solution than the original one, but it cannot prove that the solution is optimal. Overall, by combining solvers, QuaSi can find a better solution than the original SyGuS solution given by one of the two solvers for 9/18 benchmarks. QuaSi cannot improve the initial solution of the linear-integer-arithmetic benchmarks (`array_search` and `LinExpr_eq1ex`).

Both solvers timeout on large grammars. The grammars in Table 2.1 are 1 to 2 order of magnitude larger than those in existing SyGuS benchmarks (avg: 224 vs 13 rules) and existing solvers have not yet been optimized for this parameter. In some cases, the solver times out for intermediate grammars that do not contain a solution, but that generate infinitely many terms. In general, existing SyGuS solvers cannot prove unsatisfiability for these types of problems. To answer **Q1**, QuaSi can **solve quantitative variants of 10/18 real SyGuS benchmarks**.

Solving Time for Different Iterations

In this section, we evaluate the time required by each iteration of Alg. 1. Figure 2.3 shows the ratio of time taken by each iteration with respect to the initial non-quantitative SyGuS solving time. Some of the iterations shown in Figure 2.1 do not appear in Figure 2.3 since they resulted in no solution—i.e., the initial solution was minimal. CVC4 is typically slower in subsequent iterations and can take up to 10 times the original solving time, while later iterations of ESolver have comparable runtime to the initial run, and are often faster. These numbers are largely due to how the two solvers work: CVC4 is optimized to solve problems where the grammar



(a) CVC4 iterations (b) ESolver iterations

Figure 2.3: Solving time across iterations

imposes no restrictions on the structure of the solution, while ESolver performs enumerative search and takes advantage of more restrictive grammars.

One interesting point is the `parity_not` benchmark. ESolver takes 26.9s to find an initial solution. But, with a weight constraint $w < 11$, an solution can be found in 2.2s. CVC4 can find the initial solution with weight 11 in 0.1s but cannot solve the next iteration. We tried using different solvers in different iterations of our algorithm and, in fact, found that, if we use CVC4 to find an initial solution and then ESolver in subsequent iterations with restricted grammars, we can fully solve this benchmark in a total of 2.3s, which is much better than the time taken by a single solver. To answer **Q2**, with appropriate choices of solvers **the overhead of synthesizing optimal solutions is minimal**.

Solution Weight across Iterations

In this section, we present how the weight of the synthesized solutions changes across each iteration of Alg. 1. Figure 2.4 shows the percentage of weight of solutions synthesized at each iteration with respect to the weight of the initial SyGuS solution. The result shows that we can improve the solutions of CVC4 by 15-25% in one

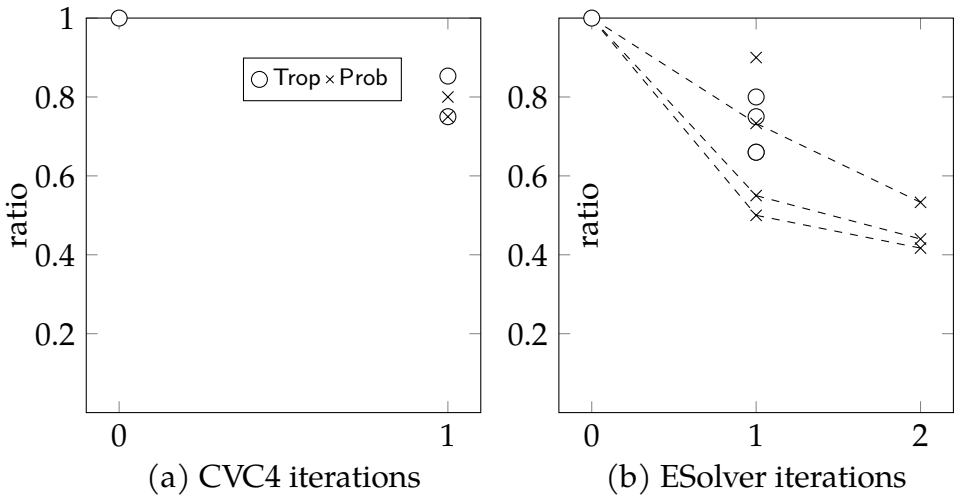


Figure 2.4: Solution weight across iterations.

iteration, and the solutions of ESolver by 20-50% when taking one iteration and 50-60% when taking two. The Prob benchmarks, which require two iterations, can be improved more when using ESolver because ESolver tends to synthesize small terms whose probability may also be small. To answer **Q3**, QuaSi can **improve the weights of SyGuS solutions by 20-60**%.

Multi-Objective Optimization

In this section, we evaluate the effectiveness of QuaSi on the 8 benchmarks involving two minimization objectives. The benchmarks consists of two families, 4 for sorted optimization and 4 for Pareto optimization. The sorted-optimization benchmarks ask to minimize first the number of occurrences of a specified operator (`bvand` in `hacks` and `ite` in `array_search`) and then the size of the solution. The Pareto-optimization benchmarks have the same objectives as sorted optimizations but here we are synthesizing a Pareto-optimal solution instead of sorted-optimal one. The results are shown in Table 2.2. We do not present the results using CVC4 because it cannot solve any of the benchmarks.

The `array_search` times out since it is already hard on a single objective. For

Table 2.2: Performance of QuaSi on multi-objective benchmarks. **Weight** denotes the sequence of weights explored during minimization.

	Problem	**T**ime[sec]	**W**eight	**L**argest	**S**ize
Trop × Trop	`array_search_sorted`	TO	-	-	15
	`hackers_5_sorted`	0.1+0.1+01	$(0,3) \to (0,2)$	31	13
	`hackers_7_sorted`	0.1+0.3+0.1	$(1,4) \to (0,5) \to (0,3)$	72	13
	`hackers_17_sorted`	0.1+156.1+TO	$(2,5) \to (1,4) \to (0,6)$	97	13
	`array_search_pareto`	TO	-	-	15
	`hackers_5_pareto`	0.1+0.1+01	$(0,3) \to (0,2)$	31	13
	`hackers_7_pareto`	0.1+0.3+0.1	$(1,4) \to (1,3) \to (0,3)$	74	13
	`hackers_17_pareto`	0.1+9.1+0.1	$(2,5) \to (2,4) \to (1,4)$	54	13

`hackers_5` benchmarks, the initial solution is already optimized for the first objective, so the problem degenerates to the single-objective optimization problem. For `hackers_7` and `hackers_17`, Table **??** shows the weights of the intermediate solutions we can see that Pareto and Sorted optimizations yield different solutions. To answer **Q4**, QuaSi can **solve problems with multiple objectives** when the same problems are feasible with a single objective.

2.6 Summary

We presented QSyGuS, a general framework for defining and solving SyGuS problems in the presence of quantitative objectives over the syntax of the programs. QSyGuS is (*i*) *natural*: requires minimal modification to the SyGuS format, (*ii*) *general*: it supports complex but practical types of weights, (*iii*) *formal*: it is grounded in the theory of weighted tree grammars, (*iv*) *effective*: our tool QuaSi can solve quantitative variations of existing SyGuS benchmarks with little overhead. In the future, we plan to extend our framework to handle probabilistic objectives and quantitative objectives over the semantics of the program—e.g., synthesize programs that satisfy most of the specification.

Chapter 3

Proving Unrealizability of Syntax-Guided Synthesis Problems

3.1 Introduction

In the previous chapter, we described how to add quantitative syntactic objectives to syntax-guided synthesis problems (SyGuS), which is a large family of synthesis problems specified by a regular-tree grammar that describes the search space of programs, and a logical formula that constitutes the behavioral specification. We also presented an algorithm for solving optimizing objectives by reducing the problems of proving solutions being optimal to problems of proving unrealizability of synthesis problems (Section 2.4). So, in this chapter, we study the problem of proving unrealizability of SyGuS problems.

Recall the SyGuS problem shown in Section 2.2 to synthesize a function f that computes the maximum of two variables x and y, denoted by $(\psi_{\texttt{max2}}(f, x, y), G_1)$. The goal is to create e_f—an expression-tree for f—where e_f is in the language of the following regular-tree grammar G_1:

$$\begin{aligned} \text{Start} &::= \text{Plus}(\text{Start}, \text{Start}) \mid \text{IfThenElse}(\text{BExpr}, \text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1 \\ \text{BExpr} &::= \text{GreaterThan}(\text{Start}, \text{Start}) \mid \text{Not}(\text{BExpr}) \mid \text{And}(\text{BExpr}, \text{BExpr}) \end{aligned}$$

and $\forall x, y.\psi_{\texttt{max2}}([\![e_f]\!], x, y)$ is valid, where $[\![e_f]\!]$ denotes the meaning of e_f, and

$$\psi_{\texttt{max2}}(f, x, y) := f(x, y) \geqslant x \wedge f(x, y) \geqslant y \wedge (f(x, y) = x \vee f(x, y) = y).$$

SyGuS solvers can easily find a solution, such as

$$e := \texttt{if } x > y \texttt{ then } x \texttt{ else } y.$$

Note that the grammar G_1 is the same as the grammar shown in Fig. 2.1 but with all operators written as functions.

Although many solvers can now find solutions efficiently to many SyGuS problems, there has been effectively no work on the much harder task of proving that a given SyGuS problem is *unrealizable*—i.e., it does not admit a solution. For example, consider the SyGuS problem $(\psi_{\texttt{max2}}(f, x, y), G_2)$, where G_2 is the more restricted grammar with if-then-else operators and conditions stripped out:

$$\text{Start} ::= \text{Plus}(\text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1$$

This SyGuS problem does *not* have a solution, because no expression generated by G_2 meets the specification.[1] However, to the best of our knowledge, current SyGuS solvers cannot prove that such a SyGuS problem is unrealizable.[2]

A key property of the previous example is that the grammar is infinite. When such a SyGuS problem is realizable, any search technique that systematically explores the infinite search space of possible programs will eventually identify a solution to the synthesis problem. In contrast, proving that a problem is unrealizable requires showing that *every* program in the *infinite* search space *fails to satisfy* the specification. This problem is in general undecidable [CRST15]. Although we

[1] Grammar G_2 only generates terms that are equivalent to some linear function of x and y; however, the maximum function cannot be described by a linear function.

[2] The synthesis problem $(\psi_{\texttt{max2}}(f, x, y), G_2)$ arises from a QSyGuS problem in which the goal is to produce an expression that (i) satisfies the specification $\psi_{\texttt{max2}}(f, x, y)$, and (ii) uses the smallest possible number of if-then-else operators. Existing SyGuS solvers can easily produce a solution that uses one if-then-else operator, but cannot prove that no better solution exists—i.e., $(\psi_{\texttt{max2}}(f, x, y), G_2)$ is unrealizable.

cannot hope to have an algorithm for establishing unrealizability, the challenge is to find a technique that succeeds for the kinds of problems encountered in practice. Existing synthesizers can detect the absence of a solution in certain cases (e.g., because the grammar is finite, or is infinite but can only generate a finite number of functionally distinct programs). However, in practice, as our experiments show, this ability is limited—no existing solver was able to show unrealizability for any of the examples considered in this chapter.

In this chapter, we present two techniques for proving that a possibly infinite SyGuS problem is unrealizable.

Proving Unrealizability via a Reachability Problem

The first technique (§3.3) is reducing the problems of proving unrealizability to reachability problems. The technique builds on two ideas.

1. We observe that unrealizability can often be proven using *finitely many input examples*. In §3.3, we show how the example discussed above can be proven to be unrealizable using four input examples—$(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$.

2. We devise a way to encode a SyGuS problem $(\psi(f,\bar{x}), G)$ over a finite set of examples E as a *reachability problem in a recursive program* $P[G,E]$. In particular, the program that we construct has an assertion that holds if and only the given SyGuS problem is unrealizable. Consequently, *unrealizability* can be proven by establishing that the assertion always holds. This property can often be established by a conventional program-analysis tool.

The encoding mentioned in item 2 is non-trivial for three reasons. The following list explains each issue, and sketches how they are addressed

1) *Infinitely many terms.* We need to model the infinitely many terms generated by the grammar of a given synthesis problem $(\psi(f,\bar{x}), G)$.

To address this issue, we use non-determinism and recursion, and give an encoding $P[G,E]$ such that (i) each non-deterministic path p in the program $P[G,E]$

corresponds to a possible expression e_p that G can generate, and (ii) for each expression e that G can generate, there is a path p_e in $P[G, E]$. (There is an isomorphism between paths and the expression-trees of G.)

2) Nondeterminism. We need the computation performed along each path p in $P[G, E]$ to mimic the execution of expression e_p. Because the program uses nondeterminism, we need to make sure that, for a given path p in the program $P[G, E]$, computational steps are carried out that mimic the evaluation of e_p for *each* of the finitely many example inputs in E.

We address this issue by threading the expression-evaluation computations associated with each example in E through the *same* non-deterministic choices.

3) Complex Specifications. We need to handle specifications that allow for nested calls of the programs being synthesized.

For instance, consider the specification $f(f(x)) = x$. To handle this specification, we introduce a new variable y and rewrite the specification as $f(x) = y \wedge f(y) = x$. Because y is now also used as an input to f, we will thread both the computations of x and y through the non-deterministic recursive program.

3.2 Background

In this chapter, we focus on *typed* regular tree grammars, in which each nonterminal and each symbol is associated with a type. There is a finite set of types $\{\tau_1, \ldots, \tau_k\}$. Associated with each symbol $\sigma^{(i)} \in \Sigma^{(i)}$, there is a type assignment $\alpha_{\sigma^{(i)}} = (\tau_0, \tau_1, \ldots, \tau_i)$, where τ_0 is called the *left-hand-side type* and $\tau_1, \ldots, \tau_i$ are called the *right-hand-side types*. Tree grammars are similar to word grammars, but generate trees over a ranked alphabet instead of words.

Definition 3.1 (Regular Tree Grammar)**.** *A* **typed regular tree grammar** (*RTG*) *is a tuple* $G = (N, \Sigma, S, \alpha, \delta)$*, where* N *is a finite set of non-terminal symbols of arity 0;* Σ *is a ranked alphabet;* $S \in N$ *is an initial non-terminal;* α *is a type assignment that gives types for members of* $\Sigma \cup N$*; and* δ *is a finite set of productions of the form* $A_0 \to \sigma^{(i)}(A_1, \ldots, A_i)$*,*

where for $1 \leqslant j \leqslant i$, *each* $A_j \in N$ *is a non-terminal such that if* $\alpha(\sigma^{(i)}) = (\tau_0, \tau_1, \ldots, \tau_i)$ *then* $\alpha(A_j) = \tau_j$.

In a SyGuS problem, each variable, such as x and y in the example RTGs in §3.1, is treated as an arity-0 symbol—i.e., $x^{(0)}$ and $y^{(0)}$.

Given a tree $t \in T_{\Sigma \cup N}$, applying a production $r = A \to \beta$ to t produces the tree t' resulting from replacing the left-most occurrence of A in t with the right-hand side β. A tree $t \in T_\Sigma$ is generated by the grammar G—denoted by $t \in L(G)$—iff it can be obtained by applying a sequence of productions $r_1 \cdots r_n$ to the tree whose root is the initial non-terminal S.

Syntax-Guided Synthesis. A SyGuS problem is specified with respect to a background theory T—e.g., linear arithmetic—and the goal is to synthesize a function f that satisfies two constraints provided by the user. The first constraint, $\psi(f, \bar{x})$, describes a *semantic property* that f should satisfy. The second constraint limits the *search space* S of f, and is given as a set of expressions specified by an RTG G that defines a subset of all terms in T.

Definition 3.2 (SyGuS)**.** *A SyGuS problem over a background theory* T *is a pair* $sy = (\psi(f, \bar{x}), G)$ *where* G *is a regular tree grammar that only contains terms in* T*—i.e.,* $L(G) \subseteq T$*—and* $\psi(f, \bar{x})$ *is a Boolean formula constraining the semantic behavior of the synthesized program* f.

A SyGuS problem is **realizable** *if there exists a expression* $e \in L(G)$ *such that* $\forall \bar{x}.\psi([\![e]\!], \bar{x})$ *is true. Otherwise we say that the problem is* **unrealizable**.

Theorem 3.3 (Undecidability [CRST15])**.** *Given a SyGuS problem* sy*, it is undecidable to check whether* sy *is realizable.*

Counterexample-Guided Inductive Synthesis. The Counterexample-Guided Inductive Synthesis (CEGIS) algorithm is a popular approach to solving synthesis problems. Instead of directly looking for an expression that satisfies the specification φ on *all* possible inputs, the CEGIS algorithm uses a synthesizer S that can find expressions that are correct on a *finite* set of examples E. If S finds a solution

that is correct on all elements of E, CEGIS uses a verifier V to check whether the discovered solution is also correct for all possible inputs to the problem. If not, a counterexample obtained from V is added to the set of examples, and the process repeats. More formally, CEGIS starts with an empty set of examples E and repeats the following steps:

1. Call the synthesizer S to find an expression e such that $\psi^E([\![e]\!], \bar{x}) \stackrel{\text{def}}{=} \forall \bar{x} \in E.\psi([\![e]\!], \bar{x})$ holds and go to step 2; return *unrealizable* if no expression exists.

2. Call the verifier V to find a model c for the formula $\neg\psi([\![e]\!], \bar{x})$, and add c to the counterexample set E; return e as a valid solution if no model is found.

Because SyGuS problems are only defined over first-order decidable theories, any SMT solver can be used as the verifier V to check whether the formula $\neg\psi([\![e]\!], \bar{x})$ is satisfiable. On the other hand, providing a synthesizer S to find solutions such that $\forall \bar{x} \in E.\psi([\![e]\!], \bar{x})$ holds is a much harder problem because e is a second-order term drawn from an infinite search space. In fact, checking whether such an e exists is an undecidable problem.

CEGIS and Unrealizability

The CEGIS algorithm is sound but incomplete for proving unrealizability. Given a SyGuS problem $sy = (\psi(f, \bar{x}), G)$ and a finite set of inputs E, we denote by $sy^E := (\psi^E(f, \bar{x}), G)$ the corresponding SyGuS problem that only requires the function f to be correct on the examples in E.

Lemma 3.4 (Soundness)**.** *If sy^E is unrealizable then sy is unrealizable.*

Proof. For every expression e and input $\bar{c}$ we have that $\psi([\![e]\!], \bar{c}) \Rightarrow \psi_E([\![e]\!], \bar{c})$ and by contraposition $\neg\psi^E([\![e]\!], \bar{c}) \Rightarrow \neg\psi([\![e]\!], \bar{c})$. Hence, the lemma holds. □

Even when given a perfect synthesizer S—i.e., one that can solve a problem sy^E for every possible set E—there are SyGuS problems for which the CEGIS algorithm is not powerful enough to prove unrealizability.

Lemma 3.5 (Incompleteness)**.** *There exists an unrealizable SyGuS problem* sy *such that for every finite set of examples* E *the problem* sy^E *is realizable.*

Proof. Let $sy_{eq} = (\psi_{eq}(f, x), G_{eq})$ be the SyGuS problem over the theory of linear-integer arithmetic such that $\psi_{eq}(f, x) \stackrel{\text{def}}{=} f(x) = x$; that is, $\psi_{eq}(f, x)$ is a predicate denoting that f should implement the identity function. Let G_{eq} be the following grammar:

$$\begin{aligned} \text{Start} &::= \text{Plus}(\text{Start}, \text{Start}) \mid \text{IfThenElse}(\text{BExpr}, \text{Start}, \text{Start}) \mid 0 \mid 1 \\ \text{BExpr} &::= \text{Equals}(x, \text{Start}) \end{aligned}$$

The problem sy_{eq} is unrealizable. Because the grammar does not contain the production Start $\rightarrow x$, every expression $e = L(G_{eq})$ can only produce a finite number of constant outputs. However, for every set of examples $E = \{n_1, \ldots, n_k\}$ the following expression $e_E \in L(G_{eq})$ is a valid solution to sy^E (i.e., $\psi^E_{eq}([\![e^E]\!], x)$ holds):

$$\begin{aligned} &\text{IfThenElse}(\text{Equals}(x, T(n_1)), T(n_1), \\ &\quad \text{IfThenElse}(\text{Equals}(x, T(n_2)), T(n_2), \ldots, T(n_k) \ldots) \end{aligned}$$

where $T(n)$ is the expression-tree corresponding to $0 + \underbrace{1 + \ldots + 1}_{n}$. Hence, the CEGIS algorithm will never terminate for sy_{eq}. □

Despite this negative result, we will show that a CEGIS algorithm can prove unrealizability for many SyGuS instances (§3.8).

3.3 Unrealizability as Verification

Illustrative Example

In this section, we illustrate the main components of our framework for establishing the unrealizability of a SyGuS problem.

Consider the SyGuS problem to synthesize a function f that computes the maximum of two variables x and y, denoted by $(\psi_{\texttt{max2}}(f, x, y), G_1)$. The goal is to create

e_f—an expression-tree for f—where e_f is in the language of the following regular-tree grammar G_1:

$$\begin{aligned} \text{Start} &::= \text{Plus}(\text{Start}, \text{Start}) \mid \text{IfThenElse}(\text{BExpr}, \text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1 \\ \text{BExpr} &::= \text{GreaterThan}(\text{Start}, \text{Start}) \mid \text{Not}(\text{BExpr}) \mid \text{And}(\text{BExpr}, \text{BExpr}) \end{aligned}$$

and $\forall x, y.\psi_{\texttt{max2}}([\![e_f]\!], x, y)$ is valid, where $[\![e_f]\!]$ denotes the meaning of e_f, and

$$\psi_{\texttt{max2}}(f, x, y) := f(x, y) \geqslant x \wedge f(x, y) \geqslant y \wedge (f(x, y) = x \vee f(x, y) = y).$$

SyGuS solvers can easily find a solution, such as

$$e := \text{IfThenElse}(\text{GreaterThan}(x, y), x, y).$$

Although many solvers can now find solutions efficiently to many SyGuS problems, there has been effectively no work on the much harder task of proving that a given SyGuS problem is *unrealizable*—i.e., it does not admit a solution. For example, consider the SyGuS problem $(\psi_{\texttt{max2}}(f, x, y), G_2)$, where G_2 is the more restricted grammar with if-then-else operators and conditions stripped out:

$$\text{Start} ::= \text{Plus}(\text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1$$

This SyGuS problem does *not* have a solution, because no expression generated by G_2 meets the specification.[3] However, to the best of our knowledge, current SyGuS solvers cannot prove that such a SyGuS problem is unrealizable. As an example, we use the problem $(\psi_{\texttt{max2}}(f, x, y), G_2)$ discussed in §3.1, and show how unrealizability can be proven using four input examples: $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$.

Our method can be seen as a variant of Counter-Example-Guided Inductive Synthesis (CEGIS), in which the goal is to create a program P in which a certain assertion always holds. Until such a program is created, each round of the algorithm

[3]Grammar G_2 generates all linear functions of x and y, and hence generates an infinite number of functionally distinct programs; however, the maximum function cannot be described by a linear function.

```
1  int I_0;
2  void Start(int x_0,int y_0){
3    if(nd()){  // Encodes ``Start ::= Plus(Start, Start)''
4      Start(x_0, y_0);
5      int tempL_0 = I_0;
6      Start(x_0, y_0);
7      int tempR_0 = I_0;
8      I_0 = tempL_0 + tempR_0;
9    }
10   else if(nd()) I_0 = x_0;  // Encodes ``Start ::= x''
11   else if(nd()) I_0 = y_0;  // Encodes ``Start ::= y''
12   else if(nd()) I_0 = 1;    // Encodes ``Start ::= 1''
13   else          I_0 = 0;    // Encodes ``Start ::= 0''
14 }
15
16 bool spec(int x, int y, int f){
17   return (f>=x && f>=y && (f==x || f==y))
18 }
19
20 void main(){
21   int x_0 = 0; int y_0 = 1;   // Input example (0,1)
22   Start(x_0,y_0);
23   assert(!spec(x_0,y_0,I_0));
24 }
```

Figure 3.1: Program $P[G_2, E_1]$ created during the course of proving the unrealizability of $(\psi_{\texttt{max2}}(f, x, y), G_2)$ using the set of input examples $E_1 = \{(0, 1)\}$.

returns a counter-example, from which we extract an additional input example for the original SyGuS problem. On the i^{th} round, the current set of input examples E_i is used, together with the grammar—in this case G_2—and the specification of the desired behavior—$\psi_{\texttt{max2}}(f, x, y)$, to create a candidate program $P[G_2, E_i]$. The program $P[G_2, E_i]$ contains an assertion, and a standard program analyzer is used to check whether the assertion always holds.

Suppose that for the SyGuS problem $(\psi_{\texttt{max2}}(f, x, y), G_2)$ we start with just the one example input $(0, 1)$—i.e., $E_1 = \{(0, 1)\}$. Fig. 3.1 shows the initial program $P[G_2, E_1]$ that our method creates. The function `spec` implements the predicate $\psi_{\texttt{max2}}(f, x, y)$. (All of the programs $\{P[G_2, E_i]\}$ use the same function `spec`.) The initialization statements "`int x_0 = 0; int y_0 = 1;`" at line (21) in procedure `main` corre-

spond to the input example $(0, 1)$. The recursive procedure `Start` encodes the productions of grammar G_2. `Start` is non-deterministic; it contains four calls to an external function `nd()`, which returns a non-deterministically chosen Boolean value. The calls to `nd()` can be understood as controlling whether or not a production is selected from G_2 during a top-down, left-to-right generation of an expression-tree: lines (3)–(8) correspond to "Start ::= Plus(Start, Start)," and lines (10), (11), (12), and (13) correspond to "Start ::= x," "Start ::= y," "Start ::= 1," and "Start ::= 0," respectively. The code in the five cases in the body of `Start` encodes the semantics of the respective production of G_2; in particular, the statements that are executed along any execution path of $P[G_2, E_1]$ implement the *bottom-up evaluation of some expression-tree that can be generated by* G_2. For instance, consider the path that visits statements in the following order (for brevity, some statement numbers have been elided):

$$21 \;\; 22 \;\; (_{\texttt{Start}} \;\; 3 \;\; 4 \;\; (_{\texttt{Start}} \;\; 10 \;\;)_{\texttt{Start}} \;\; 6 \;\; (_{\texttt{Start}} \;\; 12 \;\;)_{\texttt{Start}} \;\; 8 \;\;)_{\texttt{Start}} \;\; 23, \qquad (3.1)$$

where $(_{\texttt{Start}}$ and $)_{\texttt{Start}}$ indicate entry to, and return from, procedure `Start`, respectively. Path (3.1) corresponds to the top-down, left-to-right generation of the expression-tree `Plus(x,1)`, interleaved with the tree's bottom-up evaluation.

Note that with path (3.1), when control returns to `main`, variable `I_0` has the value 1, and thus the assertion at line (23) fails.

A sound program analyzer will discover that some such path exists in the program, and will return the sequence of non-deterministic choices required to follow one such path. Suppose that the analyzer chooses to report path (3.1); the sequence of choices would be t, f, t, f, f, f, t, which can be decoded to create the expression-tree `Plus(x,1)`. At this point, we have a candidate definition for f: $f = x + 1$. This formula can be checked using an SMT solver to see whether it satisfies the behavioral specification $\psi_{\texttt{max2}}(f, x, y)$. In this case, the SMT solver returns "false." One counter-example that it could return is $(0, 0)$.

At this point, program $P[G_2, E_2]$ would be constructed using both of the example inputs $(0, 1)$ and $(0, 0)$. Rather than describe $P[G_2, E_2]$, we will describe the final

```
int I_0, I_1, I_2, I_3;
void Start(int x_0,int y_0,...,int x_3,int y_3){
  if(nd()){  // Encodes ``Start ::= Plus(Start, Start)''
    Start(x_0, y_0, x_1, y_1, x_2, y_2, x_3, y_3);
    int tempL_0 = I_0; int tempL_1 = I_1;
    int tempL_2 = I_2; int tempL_3 = I_3;
    Start(x_0, y_0, x_1, y_1, x_2, y_2, x_3, y_3);
    int tempR_0 = I_0; int tempR_1 = I_1;
    int tempR_2 = I_2; int tempR_3 = I_3;
    I_0 = tempL_0 + tempR_0;
    I_1 = tempL_1 + tempR_1;
    I_2 = tempL_2 + tempR_2;
    I_3 = tempL_3 + tempR_3;}
  else if(nd()) {  // Encodes ``Start ::= x''
    I_0 = x_0; I_1 = x_1; I_2 = x_2; I_3 = x_3;}
  else if(nd()) {  // Encodes ``Start ::= y''
    I_0 = y_0; I_1 = y_1; I_2 = y_2; I_3 = y_3;}
  else if(nd()) {  // Encodes ``Start ::= 1''
    I_0 = 1;   I_1 = 1;   I_2 = 1;   I_3 = 1;}
  else {           // Encodes ``Start ::= 0''
    I_0 = 0;   I_1 = 0;   I_2 = 0;   I_3 = 0;}
}

bool spec(int x, int y, int f){
  return (f>=x && f>=y && (f==x || f==y))
}

void main(){
  int x_0 = 0; int y_0 = 1;  // Input example (0,1)
  int x_1 = 0; int y_1 = 0;  // Input example (0,0)
  int x_2 = 1; int y_2 = 1;  // Input example (1,1)
  int x_3 = 1; int y_3 = 0;  // Input example (1,0)
  Start(x_0,y_0,x_1,y_1,x_2,y_2,x_3,y_3);
  assert(   !spec(x_0,y_0,I_0) || !spec(x_1,y_1,I_1)
         || !spec(x_2,y_2,I_2) || !spec(x_3,y_3,I_3));
}
```

Figure 3.2: Program $P[G_2, E_4]$ created during the course of proving the unrealizability of $(\psi_{\texttt{max2}}(f, x, y), G_2)$ using the set of input examples $E_4 = \{(0,0), (0,1), (1,0), (1,1)\}$.

program constructed, $P[G_2, E_4]$ (see Fig. 3.2).

As can be seen from the comments in the two programs, program $P[G_2, E_4]$ has

the same basic structure as $P[G_2, E_1]$.

- `main` begins with initialization statements for the four example inputs.
- `Start` has five cases that correspond to the five productions of G_2.

The main difference is that because the encoding of G_2 in `Start` uses non-determinism, we need to make sure that along *each* path p in $P[G_2, E_4]$, each of the example inputs is used to evaluate the *same* expression-tree. We address this issue by threading the expression-evaluation computations associated with each of the example inputs through the *same* non-deterministic choices. That is, each of the five "production cases" in `Start` has four encodings of the production's semantics—one for each of the four expression evaluations. By this means, the statements that are executed along path p perform *four simultaneous bottom-up evaluations* of the expression-tree from G_2 that corresponds to p.

Programs $P[G_2, E_2]$ and $P[G_2, E_3]$ are similar to $P[G_2, E_4]$, but their paths carry out two and three simultaneous bottom-up evaluations, respectively. The actions taken during rounds 2 and 3 to generate a new counter-example—and hence a new example input—are similar to what was described for round 1. On round 4, however, the program analyzer will determine that the assertion on lines (34)–(35) always holds, which means that there is no path through $P[G_2, E_4]$ for which the behavioral specification holds for all of the input examples. This property means that there is no expression-tree that satisfies the specification—i.e., the SyGuS problem $(\psi_{\texttt{max2}}(f, x, y), G_2)$ is unrealizable.

Our implementation uses the program-analysis tool SeaHorn [GKN15] as the assertion checker. In the case of $P[G_2, E_4]$, SeaHorn takes only 0.5 seconds to establish that the assertion in $P[G_2, E_4]$ always holds.

From Unrealizability to Unreachability

In this section, we show how a SyGuS problem for finitely many examples can be reduced to a reachability problem in a non-deterministic, recursive program in an imperative programming language.

Reachability Problems

A *program* P takes an initial state I as input and outputs a final state O, i.e., $[\![P]\!](I) = O$ where $[\![\cdot]\!]$ denotes the semantic function of the programming language. As illustrated in §3.3, we allow a program to contain calls to an external function `nd()`, which returns a non-deterministically chosen Boolean value. When program P contains calls to `nd()`, we use $\hat{P}$ to denote the program that is the same as P except that $\hat{P}$ takes an additional integer input `n`, and each call `nd()` is replaced by a call to a local function `nextbit()` defined as follows:

```
bool nextbit(){bool b = n%2; n=n>>1; return b;}.
```

In other words, the integer parameter `n` of $\hat{P}[n]$ formalizes all of the non-deterministic choices made by P in calls to `nd()`.

For the programs $P[G, E]$ used in our unrealizability algorithm, the only calls to `nd()` are ones that control whether or not a production is selected from grammar G during a top-down, left-to-right generation of an expression-tree. Given `n`, we can decode it to identify which expression-tree `n` represents.

Example 3.6. *Consider again the SyGuS problem* $(\psi_{max2}(f, x, y), G_2)$ *discussed in §3.3. In the discussion of the initial program* $P[G_2, E_1]$ *(Fig. 3.1), we hypothesized that the program analyzer chose to report path (3.1) in* P*, for which the sequence of non-deterministic choices is* t, f, t, f, f, f, t*. That sequence means that for* $\hat{P}[n]$*, the value of* `n` *is* 1000101 *(base* 2*)* *(or* 69 *(base* 10*))*. *The* 1*s, from low-order to high-order position, represent choices of production instances in a top-down, left-to-right generation of an expression-tree. (The* 0*s represent rejected possible choices.) The rightmost* 1 *in* `n` *corresponds to the choice in line (3) of "*`Start ::= Plus(Start, Start)`*"; the* 1 *in the third-from-rightmost position corresponds to the choice in line (10) of "*`Start ::= x`*" as the left child of the* `Plus` *node; and the* 1 *in the leftmost position corresponds to the choice in line (12) of "*`Start ::= 1`*" as the right child. By this means, we learn that the behavioral specification* $\psi_{max2}(f, x, y)$ *holds for the example set* $E_1 = \{(0, 1)\}$ *for* $f \mapsto$ `Plus(x,1)`*.* □

Definition 3.7 (Reachability Problem)**.** *Given a program* $\hat{P}[n]$*, containing assertion statements and a non-deterministic integer input* n*, we use* re_P *to denote the corresponding reachability problem. The reachability problem* re_P *is* ***satisfiable*** *if there exists a value* n *that, when bound to* n*, falsifies any of the assertions in* $\hat{P}[n]$*. The problem is* ***unsatisfiable*** *otherwise.*

Reduction to Reachability

The main component of our framework is an encoding *enc* that given a SyGuS problem $sy^E = (\psi^E(f, x), G)$ over a set of examples $E = \{c_1, \ldots, c_k\}$, outputs a program $P[G, E]$ such that sy^E is **realizable** if and only if $re_{enc(sy,E)}$ is **satisfiable**. In this section, we define all the components of $P[G, E]$, and state the correctness properties of our reduction.

Remark: In this section, we assume that in the specification $\psi(f, x)$ every occurrence of f has x as input parameter. We show how to overcome this restriction in §3.3. In the following, we assume that the input x has type τ_I, where τ_I could be a complex type—e.g., a tuple type.

Program construction. Recall that the grammar G is a tuple $(N, \Sigma, S, \alpha, \delta)$. First, for each non-terminal $A \in N$, the program $P[G, E]$ contains k global variables $\{g\_1\_A, \ldots, g\_k\_A\}$ of type $\alpha(A)$ that are used to express the values resulting from evaluating expressions generated from non-terminal A on the k examples. Second, for each non-terminal $A \in N$, the program $P[G, E]$ contains a function

$$\texttt{void funcA}(\tau_I\ \texttt{v1}, \ldots, \tau_I\ \texttt{vk})\{\ \textit{bodyA}\ \}$$

We denote by $\delta(A) = \{r_1, \ldots, r_m\}$ the set of production rules of the form $A \to \beta$ in δ. The body $\texttt{body}A$ of `funcA` has the following structure:

$$\begin{aligned}
&\texttt{if(nd())}\ \{\texttt{En}_\delta(r_1)\}\\
&\texttt{else if(nd())}\ \{\texttt{En}_\delta(r_2)\}\\
&\ldots\\
&\texttt{else}\ \{\texttt{En}_\delta(r_m)\}
\end{aligned}$$

The encoding $\mathtt{En}_\delta(r)$ of a production $r = A_0 \to b^{(j)}(A_1, \cdots, A_j)$ is defined as follows (τ_i denotes the type of the term A_i):

```
funcA1(v1,…,vk);
τ1 child_1_1 = g_1_A1;...;τ1 child_1_k = g_k_Aj;
...
funcAj(v1,…,vk);
τj child_j_1 = g_1_A1;...;τj child_j_k = g_k_Aj;
g_1_A0 = enc_b^1(child_1_1,...,child_1_k)
...
g_k_A0 = enc_b^k(child_j_1,...,child_j_k)
```

Note that if $b^{(j)}$ is of arity 0—i.e., if $j = 0$—the construction yields k assignments of the form $\mathtt{g\_m\_A0} = enc_b^m()$.

The function enc_b^m interprets the semantics of b on the m^{th} input example. We take Linear Integer Arithmetic as an example to illustrate how enc_b^m works.

$$
\begin{array}{rlrl}
enc^m_{0^{(0)}} & := \mathtt{0} & enc^m_{1^{(0)}} & := \mathtt{1} \\
enc^m_{x^{(0)}} & := \mathtt{vi} & enc^m_{\mathrm{Equals}^{(2)}}(L,R) & := (L\mathtt{=}R) \\
enc^m_{\mathrm{Plus}^{(2)}}(L,R) & := L\mathtt{+}R & enc^m_{\mathrm{Minus}^{(2)}}(L,R) & := L\mathtt{-}R \\
\multicolumn{4}{c}{enc^m_{\mathrm{IfThenElse}^{(3)}}(B,L,R) := \mathtt{if}(B)\ L\ \mathtt{else}\ R}
\end{array}
$$

We now turn to the correctness of the construction. First, we formalize the relationship between expression-trees in $L(G)$, the semantics of $P[G, E]$, and the number n. Given an expression-tree e, we assume that each node q in e is annotated with the production that has produced that node. Recall that $\delta(A) = \{r_1, \ldots, r_m\}$ is the set of productions with head A (where the subscripts are indexes in some arbitrary, but fixed order). Concretely, for every node q, we assume there is a function $pr(q) = (A, i)$, which associates q with a pair that indicates that non-terminal A produced n using the production r_i (i.e., r_i is the i^{th} production whose left-hand-side non-terminal is A).

We now define how we can extract a number $\#(e)$ for which the program $\hat{P}[\#(e)]$

will exhibit the same semantics as that of the expression-tree e. First, for every node q in e such that $\mathrm{pr}(q) = (A, i)$, we define the following number:

$$\#_{nd}(q) = \begin{cases} 1\underbrace{0\cdots0}_{i-1} & \text{if } i < |\delta(A)| \\ \underbrace{0\cdots0}_{i-1} & \text{if } i = |\delta(A)|. \end{cases}$$

The number $\#_{nd}(q)$ indicates what suffix of the value of n will cause `funcA` to trigger the code corresponding to production r_i. Let $q_1 \cdots q_m$ be the sequence of nodes visited during a pre-order traversal of expression-tree e. The number corresponding to e, denoted by $\#(e)$, is defined as the bit-vector $\#_{nd}(q_m) \cdots \#_{nd}(q_1)$.

Finally, we add the entry-point of the program, which calls the function `funcS` corresponding to the initial non-terminal S, and contains the assertion that encodes our reachability problem on all the input examples $E = \{c_1, \ldots, c_k\}$.

```
void main(){
  τ_I x1 = c_1; ··· ;τ_I xk = c_k;
  funcS(x1, ..., xk);
  assert ⋁_{1⩽i⩽k} ¬ψ(f, c_i)[g_i_S/f(x)]; // At least one c_i fails }
```

Correctness. We first need to show that the function $\#(\cdot)$ captures the correct language of expression-trees. Given a non-terminal A, a value n, and input values $i_1, \ldots, i_k$, we use $[\![\texttt{funcA}[n]]\!](i_1, \ldots, i_k) = (o_1, \ldots o_k)$ to denote the values of the variables $\{\texttt{g\_1\_A}, \ldots, \texttt{g\_k\_A}\}$ at the end of the execution of $\texttt{funcA}[n]$ with the initial value of $n = n$ and input values $x_1, \ldots, x_k$. Given a non-terminal A, we write $L(G, A)$ to denote the set of terms that can be derived starting with A.

Lemma 3.8. *Let A be a non-terminal, $e \in L(G, A)$ an expression, and $\{i_1, \ldots, i_k\}$ an input set. Then, $([\![e]\!](i_1), \ldots, [\![e]\!](i_k)) = [\![\mathit{funcA}[\#(e)]]\!](i_1, \ldots, i_k)$.*

Proof. The proof is by structural induction on e. Let q denote the root of e, and $A \to \sigma^{(j)}(A_1, \ldots, A_j)$ denote the production instance at q. Note that $\#(e) =$

$\#(e_j)\cdots\#(e_1)\#_{nd}(q)$.

Suppose that $e = q$ is a leaf node; that is, the tree at q is an instance of a production of the form $A \to q^{(0)}$. Because $\#(e) = \#_{nd}(q^{(0)})$, for every input set $\{i_1,\ldots,i_k\}$, `funcA`$[\#(e)]$ selects the branch in `funcA` that captures the semantics of $A \to q^{(0)}$. In that code, e is evaluated on the k values $\{i_1,\ldots,i_k\}$. Therefore, $([\![e]\!](i_1),\ldots,[\![e]\!](i_k)) = [\![\texttt{funcA}[\#(e)]]\!](i_1,\ldots,i_k)$ holds.

Inductive step: Let $e = \sigma^{(j)}(e_1,\ldots,e_j)$, where the property to be shown is assumed to hold for each of the e_l. For each e_l, let q_l be the root of e_l.

The procedure `funcA`$[\#(e)]$ uses $\#_{nd}(q)$ to select the branch B in `funcA` that captures the semantics of the production $A \to \sigma^{(j)}(A_1,\ldots,A_j)$. For every input set $\{i_1,\ldots,i_k\}$, the induction hypothesis ensures that the following property holds: for $1 \leqslant l \leqslant j$, $([\![e_l]\!](i_1),\ldots,[\![e_l]\!](i_k)) = [\![\texttt{funcA}l[\#(e_l)]]\!](i_1,\ldots,i_k)$. Therefore, each call to a procedure `funcA`l in B computes the k intermediate answers that correspond to the evaluation of e_l on the k values $\{i_1,\ldots,i_k\}$. The code in B that follows the final call to `funcA`j uses the collections of intermediate results to finish k computations of the semantics of $A \to \sigma^{(j)}(A_1,\ldots,A_j)$. Therefore, $([\![e]\!](i_1),\ldots,[\![e]\!](i_k)) = [\![\texttt{funcA}[\#(e)]]\!](i_1,\ldots,i_k)$ holds. □

Each procedure `funcA`$[n](i_1,\ldots,i_k)$ that we construct has an explicit dependence on variable `n`, where `n` controls the non-deterministic choices made by the `funcA` and procedures called by `funcA`. As a consequence, when relating numbers and expression-trees, there are two additional issues to contend with:

Non-termination. Some numbers can cause `funcA`$[n]$ to fail to terminate. For instance, if the case for "`Start ::= Plus(Start, Start)`" in program $P[G_2, E_1]$ from Fig. 3.1 were moved from the first branch (lines (3)–(8)) to the final else case (line (13)), the number $n = 0 = \ldots0000000$ (base 2) would cause `Start` to never terminate, due to repeated selections of `Plus` nodes. However, note that the only assert statement in the program is placed at the end of the main procedure. Now, consider a value of `n` such that $re_{enc(sy,E)}$ is satisfiable. Defn. 3.7 implies that the flow of control will reach and falsify the assertion,

which implies that $\texttt{funcA}[n]$ terminates. [4]

Shared suffixes of sufficient length. In Ex. 3.6, we showed how for program $P[G_2, E_1]$ (Fig. 3.1) the number $n = 1000101$ (base 2) corresponds to the top-down, left-to-right generation of `Plus(x,1)`. That derivation consumed exactly seven bits; thus, any number that, written in base 2, shares the suffix 1000101—e.g., 11010101000101—will also generate `Plus(x,1)`.

The issue of shared suffixes is addressed in the following lemma:

Lemma 3.9. *For every non-terminal* A *and number* n *such that* $[\![\mathit{funcA}[n]]\!](i_1, \ldots, i_k) \neq \perp$ (*i.e.,* funcA *terminates when the non-deterministic choices are controlled by* n)*, there exists a minimal* n' *that is a* (*base* 2) *suffix of* n *for which* (*i*) *there is an* $e \in L(G)$ *such that* $\#(e) = n'$*, and* (*ii*) *for every input* $\{i_1, \ldots, i_k\}$*, we have* $[\![\mathit{funcA}[n]]\!](i_1, \ldots, i_k) = [\![\mathit{funcA}[n']]\!](i_1, \ldots, i_k)$.

Proof. Assume that the computation $[\![\texttt{funcA}[n]]\!](i_1, \ldots, i_k)$ terminates. Let $b_1, \ldots, b_j$ be the finite sequence of bits drawn by `nd()` throughout the computation.
*P*roof of (i): Let e be the expression-tree generated top-down, left-to-right using the sequence $b_1, \ldots, b_j$. Let n' be the binary number $b_j \cdots b_1$. Because $\#(e)$ is the concatenation, in right-to-left order, of the sequence of $\#(\cdot)$ values for the nodes of e visited during a pre-order traversal, $\#(e) = n'$.
*P*roof of (ii): Property (ii) holds because n and n' agree on the (base 2) suffix $b_j \cdots b_1$, and exactly j bits are used during the executions of both $\texttt{funcA}[n](i_1, \ldots, i_k)$ and $\texttt{funcA}[n'](i_1, \ldots, i_k)$—which also shows that $n' = b_j \cdots b_1$ (base 2) is minimal. □

We are now ready to state the correctness property of our construction.

Theorem 3.10. *Given a* SyGuS *problem* $sy^E = (\psi_E(f, x), G)$ *over a finite set of examples* E*, the problem* sy^E *is* ***realizable*** *iff* $re_{enc(sy,E)}$ *is* ***satisfiable****.*

[4] If the SyGuS problem deals with the synthesis of programs for a language that can express non-terminating programs, that would be an additional source of non-termination, different from that discussed in item **Non-termination**. That issue does not arise for LIA SyGuS problems. Dealing with the more general kind of non-termination is postponed for future work.

Proof. $\Rightarrow$ *direction*: Assume that $\mathtt{sy}^{\mathtt{E}}$ is **realizable**. Then there exists an expression $e \in L(G) = L(G, S)$ such that $\forall x \in E.\psi([\![e]\!], x)$. By Lemma 3.8, for every $\{i_1, \ldots, i_k\}$, $([\![e]\!](i_1), \ldots, [\![e]\!](i_k)) = [\![\mathtt{funcA}[\#(e)]]\!](i_1, \ldots, i_k)$. Hence, the assertion in program $enc(\mathtt{sy}, E)$ is false and the reachability problem $\mathtt{re}_{enc(\mathtt{sy},E)}$ is **satisfiable**.

$\Leftarrow$ *direction*: Assume that $\mathtt{re}_{enc(\mathtt{sy},E)}$ is **satisfiable**. Then there exists a value of n that makes the assertion in program $enc(\mathtt{sy}, E)$ false (i.e., the specification holds for all inputs $c_i \in E$). By Lemma 3.9, there exists a minimal n' for which the program has equivalent semantics (in particular, the assertion in $enc(\mathtt{sy}, E)$ is still false), and there exists an expression $e \in L(G)$ such that $\#(e) = n'$. Hence, e is a solution to SyGuS problem $\mathtt{sy}^{\mathtt{E}}$; i.e., $\mathtt{sy}^{\mathtt{E}}$ is **realizable**. □

Encoding in the Presence of Nested Function-Invocations

In §3.3, we presented a simplified encoding that relied on the specification $\psi(f, x)$ to only involve function invocations of the form $f(x)$, where x represents the input parameter of the function to be synthesized. In this section, we show with a simple example how such a restriction can be overcome.

Consider the following semantic specification that involves multiple invocations of the function f on different arguments, as well as nested function calls:

$$\psi_1(f, x) \stackrel{\text{def}}{=} f(f(x)) = f(x + x).$$

By introducing new input variables and performing the proper refactoring, we can rewrite ψ_1 as the following specification, where f is always called on a single input variable:

$$\psi_2(f, x, y_1, y_2, y_3, y_4) \stackrel{\text{def}}{=} \left[\begin{array}{cc} & f(x) = y_1 \wedge f(y_1) = y_2 \\ \wedge & x + x = y_3 \wedge f(y_3) = y_4 \end{array}\right] \rightarrow y_2 = y_4.$$

It is now easy to adapt our encoding to operate over this new specification. First, the program $P[G, E]$ will now operate over input examples of the form $c = \{w_1, \ldots, w_k\}$, where each example c is a tuple corresponding to the values

of variables $\{x, y_1, y_2, y_3, y_4\}$. Second, the program will need to compute the values of all possible calls of f on the various input parameters. Hence, for every expression $f(z)$ in ψ_2, non-terminal A, and example w_i, the program $P[G, E]$ will have a global variable `z_i_A` computing the value of the expression generated by A parametrized by z, with respect to the values in input example w_i.

For instance, assume that the input grammar has a production $A \to \pi_1$ that generates an access on the first parameter of the function to be synthesized, and assume that we currently only have one input example. The corresponding code for the production would be

```
funcA (int v_x, int v_y1, int v_y2, int v_y3, int v_y4) {
  if(nd()) {
    x_1_A = v_x;   // Computing f(x)
    y1_1_A = v_y1; // Computing f(y1)
    y3_1_A = v_y3; // Computing f(y3)
  }
  ...
```

In summary, thanks to the ability to execute a finite number of inputs in lock-step, our encoding can handle specifications that contain nested function-invocations.

Overcoming a Quirk of SeaHorn

Because SeaHorn is unsound for satisfiability, it can report that some expression-tree satisfies behavioral specification ψ, when in fact no such expression-tree exists. In effect, SeaHorn overapproximates the set of reachable states, and erroneously concludes that the assertion in $\mathrm{re}_{enc(sy,E)}$ can be falsified (i.e., all example inputs satisfy ψ). We encountered this situation in our experiments; for some unknown reason, when the following two productions were included in the grammar, SeaHorn would report that $\mathrm{re}_{enc(sy,E)}$ was ***satisfiable*** in cases when it should have

reported ***unsatisfiable***:

$$\text{BExpr} ::= \text{Not}(\text{BExpr}) \mid \text{And}(\text{BExpr}, \text{BExpr}) \tag{3.2}$$

For the examples on which this happened, we found that we could delete these two productions, which resulted in a grammar of equivalent expressiveness. That is, because the grammar still contained the IfThenElse operator, for all expressions e_1, e_2, e_3, and e_4, the expression $\text{IfThenElse}(\text{Not}(e_1), e_2, e_3)$ is equivalent to $\text{IfThenElse}(e_1, e_3, e_2)$, and the expression $\text{IfThenElse}(\text{And}(e_1, e_2), e_3, e_4)$ is equivalent to $\text{IfThenElse}(e_1, \text{IfThenElse}(e_2, e_3, e_4), e_4)$. When we ran the same SyGuS problem sy with productions (3.2) removed from the grammar, SeaHorn reported that $re_{enc(sy,E)}$ was unsatisfiable. Because SeaHorn is sound for unsatisfiability, the latter is the correct answer, and demonstrates that SyGuS problem sy (in both modified and unmodified form) is unrealizable.

Because the expressibility of the grammar is unchanged with and without productions (3.2), these examples demonstrate that the effect is caused by some overapproximation made by SeaHorn, triggered by productions (3.2) and the encoding described in §3.3.

3.4 Unrealizability as Grammar Flow Analysis

Illustrative Example

SyGuS problems in LIA. Consider the SyGuS problem in which the goal is to create a term e_f whose meaning is $e_f(x) := 2x + 2$, but where e_f is in the language of the following regular tree grammar G_1:[5]

$$\textit{Start} ::= \text{Plus}(\text{Var}(x), \text{Var}(x), \text{Var}(x), \textit{Start}) \mid \text{Num}(0) \tag{3.3}$$

[5] For readability, we allow grammars to contain n-ary Plus symbols and trees. In the next sections, we will write the grammar G_1 as follows:

$\textit{Start} ::= \text{Plus}(\textit{S1}, \textit{Start}) \mid \text{Num}(0)$ $\quad$ $\textit{S1} ::= \text{Plus}(\textit{S2}, \text{Var}(x))$

$\textit{S2} ::= \text{Plus}(\textit{S3}, \text{Var}(x))$ $\quad$ $\textit{S3} ::= \text{Var}(x)$.

This problem is unrealizable because every term in the grammar G_1 is of—in essence—the form $3kx$ (with $k \geqslant 0$).

A typical synthesizer tries to solve this problem using a counterexample-guided inductive synthesis (CEGIS) strategy that searches for a program consistent with a finite set of examples E. Here, let's assume that the initial input example in E is i_1, which has x set to 1—i.e $i_1(x) = 1$. For this example, the input i_1 corresponds to the output $o_1 = 4$.

In this particular case, there exists no term in the grammar G_1 that is consistent with the example i_1. To prove that this grammar does not contain a term that is consistent with the specification on the example i_1, we compute for each nonterminal A a value $n_{1,E}(A)$ [6] that describes the set of values any term derived from A can produce when evaluated on i_1—i.e., $\gamma(n_{1,E}(A)) \supseteq \{[\![e]\!](i_1) \mid e \in L_{G_1}(A)\}$, where, as usual in abstract interpretation, γ denotes the concretization function. As we show in §3.4, for $n_{1,E}(A)$ to be an overapproximation of the set of output values that any term derived from A can produce for the current set of examples E, it should satisfy the following equation:

$$n_{1,E}(\mathit{Start}) = [\![\mathrm{Plus}]\!]^\#_E([\![\mathrm{Var}(x)]\!]^\#_E, [\![\mathrm{Var}(x)]\!]^\#_E, [\![\mathrm{Var}(x)]\!]^\#_E, n_{1,E}(\mathit{Start})) \oplus [\![\mathrm{Num}(0)]\!]^\#_E. \tag{3.4}$$

For every term e, the notation $[\![e]\!]^\#_E$ denotes an abstract semantics of e—i.e., $[\![e]\!]^\#_E$ overapproximates the set of values e can produce when evaluated on the examples in E—and $\oplus$ denotes the *join* operator, which overapproximates $\cup$.

In this example, we represent each $n_{1,E}(A)$ using a *semi-linear set*—i.e., a set of terms $\{l_1, \ldots, l_n\}$, where each l_i is a term of the form $c + \lambda_1 c_1 + \cdots + \lambda_k c_k$ (called a *linear set*), the values $\lambda_i \in \mathbb{N}$ are parameters, and the values $c_j \in \mathbb{Z}$ are fixed coefficients. We then replace each $[\![e]\!]^\#_E$ with a corresponding semi-linear-set interpretation. For example, $[\![\mathrm{Var}(x)]\!]^\#_E$ is the vector of inputs E projected onto the x coordinate—i.e., $[\![\mathrm{Var}(x)]\!]^\#_E = \{i_1(x)\} = \{1\}$. We rewrite $[\![\mathrm{Plus}]\!]^\#_E$ as $\otimes$, with $x \otimes y$

[6] This section uses a simplified notation for readability. In §3.4 the term $n_{1,E}(A)$ is written $n_{\mathcal{G}_{1E}}$ where $\mathcal{G}_1$ is used to denote a GFA problem.

being the semi-linear set representing $\{a + b \mid a \in x, b \in y\}$

We rewrite Eqn. (3.4) to use semi-linear sets:

$$n_{1,E}(\mathit{Start}) = (\{1\} \otimes \{1\} \otimes \{1\} \otimes n_{1,E}(\mathit{Start})) \oplus \{0\}, \tag{3.5}$$

where $x \oplus y$ is the semi-linear set representing $\{a \mid a \in x \vee a \in y\}$. These operations can be performed precisely.

In this example, an *exact* solution to this set of equations is the semi-linear set $n_{1,E}(\mathit{Start}) = \{0 + \lambda 3\}$, which describes the set of all possible values produced by any term in grammar G_1 for the set of examples $E = \langle i_1 \rangle$. In particular, such a solution can be computed automatically [EKL10].[7] This SyGuS problem does *not* have a solution, because none of the values in $n_{1,E}(\mathit{Start})$ meets the specification on the given input example, i.e., the following formula is not satisfiable:

$$\exists \lambda.[i_1 = 1 \wedge o_1 = 0 + \lambda 3 \wedge \lambda \geqslant 0] \wedge o_1 = 2i_1 + 2. \tag{3.6}$$

SyGuS problems in CLIA. For grammars with a more complex background theory, such as CLIA (LIA with conditionals), it may be more complicated to compute an overapproximation of the possible outputs of any term in the grammar. For example, consider the SyGuS problem where once again the goal is to synthesize a term whose meaning is $e_f(x) := 2x + 2$, but now in the more expressive CLIA grammar G_2:

$$\begin{aligned}
\mathit{Start} ::= {} & \mathrm{IfThenElse}(\mathrm{BExp}, \mathrm{Exp3}, \mathit{Start}) \mid \mathrm{Exp2} \mid \mathrm{Exp3} \\
\mathrm{BExp} ::= {} & \mathrm{LessThan}(\mathrm{Var}(x), \mathrm{Num}(2)) \\
\mid {} & \mathrm{LessThan}(\mathrm{Num}(0), \mathit{Start}) \mid \mathrm{And}(\mathrm{BExp}, \mathrm{BExp}) \\
\mathrm{Exp2} ::= {} & \mathrm{Plus}(\mathrm{Var}(x), \mathrm{Var}(x), \mathrm{Exp2}) \mid \mathrm{Num}(0) \\
\mathrm{Exp3} ::= {} & \mathrm{Plus}(\mathrm{Var}(x), \mathrm{Var}(x), \mathrm{Var}(x), \mathrm{Exp3}) \mid \mathrm{Num}(0)
\end{aligned} \tag{3.7}$$

[7] Some intuition can be gained by thinking of Eqn. (3.5) as being similar to a context-free grammar of the form $X := aX \mid b$, which has the regular-language solution a^*b. Similarly, Eqn. (3.5) has the solution $\{3\}^{\circledast} \otimes \{0\}$. Here $\circledast$ is the iterated addition of the (trivial) semi-linear set $\{3\}$, so the overall solution is $\{0 + 3\lambda \mid \lambda \in \mathbb{N}\}$.

Consider again the input example i_1=1 with output o_1=4. The candidate term Plus(Var(x), Var(x), Plus(Var(x), Var(x), Num(0))) in this grammar is correct on the input i_1. A SyGuS solver that enumerates all terms in the grammar will find this term, test it on the given specification, see that it is not correct on all inputs, and produce a counterexample. In this case, suppose that the counterexample is i_2 where $i_2(x)$=2 with the corresponding output o_2=6. There is no term in G_2 that is consistent with both of these examples, and we will prove this fact like we did before, that is, by solving the following set of equations:[8]

$$\begin{aligned}
n_{2,E}(\mathit{Start}) = {} & [\![\mathrm{IfThenElse}]\!]^{\#}_{E}(n_{2,E}(\mathrm{BExp}), n_{2,E}(\mathrm{Exp3}), \\
& n_{2,E}(\mathit{Start})) \oplus n_{2,E}(\mathrm{Exp2}) \oplus n_{2,E}(\mathrm{Exp3}) \\
n_{2,E}(\mathrm{BExp}) = {} & [\![\mathrm{LessThan}]\!]^{\#}_{E}([\![\mathrm{Var}(x)]\!]^{\#}_{E}, [\![\mathrm{Num}(2)]\!]^{\#}_{E}) \\
\oplus {} & [\![\mathrm{LessThan}]\!]^{\#}_{E}([\![\mathrm{Num}(0)]\!]^{\#}_{E}, n_{2,E}(\mathit{Start})) \\
\oplus {} & [\![\mathrm{And}]\!]^{\#}_{E}(n_{2,E}(\mathrm{BExp}), n_{2,E}(\mathrm{BExp})) \\
n_{2,E}(\mathrm{Exp2}) = {} & [\![\mathrm{Plus}]\!]^{\#}_{E}([\![\mathrm{Var}(x)]\!]^{\#}_{E}, [\![\mathrm{Var}(x)]\!]^{\#}_{E}, n_{2,E}(\mathrm{Exp2})) \\
\oplus {} & [\![\mathrm{Num}(0)]\!]^{\#}_{E} \\
n_{2,E}(\mathrm{Exp3}) = {} & [\![\mathrm{Plus}]\!]^{\#}_{E}([\![\mathrm{Var}(x)]\!]^{\#}_{E}, [\![\mathrm{Var}(x)]\!]^{\#}_{E}, [\![\mathrm{Var}(x)]\!]^{\#}_{E}, \\
& n_{2,E}(\mathrm{Exp3})) \oplus [\![\mathrm{Num}(0)]\!]^{\#}_{E}
\end{aligned} \tag{3.8}$$

Because we want to track the possible values each term can have for *both* examples, we need a domain that summarizes vectors of values. Luckily, semi-linear sets can easily be extended to vectors—i.e., each l_i in a semi-linear set sl is a linear set of the form $\{\vec{v}_0 + \lambda_1\vec{v}_1 + \cdots + \lambda_k\vec{v}_k \mid \lambda_i \in \mathbb{N}\}$ (with $\vec{v}_j \in \mathbb{Z}^k$). Second, because some nonterminals are Boolean-valued and some are integer-valued, we need different representations of the possible outputs of each nonterminal. We will use semi-linear sets for $n_{2,E}(\mathit{Start})$, $n_{2,E}(\mathrm{Exp2})$ and $n_{2,E}(\mathrm{Exp3})$, and a set of Boolean vectors for $n_{2,E}(\mathrm{BExp})$—e.g., $n_{2,E}(\mathrm{BExp})$ could be a set $\{(t, f), (t, t)\}$, which denotes that a Boolean expression generated by BExp can be true for i_1 and false for i_2, or true for

[8]Note that the $\oplus$ symbol is overloaded. On the right-hand side of $n_{2,E}(\mathit{BExp})$, $\oplus$ is an operation on an abstract Boolean value, whereas the $\oplus$ on the right-hand-side of the other equations is an operation on semi-linear sets. Both operations denote set union, and are handled in a uniform way by operating over a multi-sorted domain of Booleans and semi-linear sets.

both. We can now instantiate all constant terminals and variable terminals with their abstraction, e.g., $[\![\mathrm{Var}(x)]\!]^{\#}_{E}$ with $\{(1,2)\}$ and $[\![\mathrm{Num}(0)]\!]^{\#}_{E}$ with $\{(0,0)\}$.

We then start solving part of our equations by observing that Exp2 and Exp3 are only recursive in themselves. Therefore, we can compute their summaries independently, obtaining $n_{2,E}(\mathrm{Exp2}) = \{(0,0) + \lambda(2,4)\}, n_{2,E}(\mathrm{Exp3}) = \{(0,0) + \lambda(3,6)\}$. We can now replace all instances of $n_{2,E}(\mathrm{Exp2})$ and $n_{2,E}(\mathrm{Exp3})$, and obtain the following set of equations:

$$\begin{aligned} n_{2,E}(\mathit{Start}) = {} & [\![\mathrm{IfThenElse}]\!]^{\#}_{E}(n_{2,E}(\mathrm{BExp}), \{(0,0) + \lambda(3,6)\}, \\ & n_{2,E}(\mathit{Start})) \oplus \{(0,0) + \lambda(2,4)\} \\ & \oplus \{(0,0) + \lambda(3,6)\} \\ n_{2,E}(\mathrm{BExp}) = {} & \{(t,f)\} \oplus [\![\mathrm{LessThan}]\!]^{\#}_{E}(\{(0,0)\}, n_{2,E}(\mathit{Start})) \\ & \oplus [\![\mathrm{And}]\!]^{\#}_{E}(n_{2,E}(\mathrm{BExp}), n_{2,E}(\mathrm{BExp})) \end{aligned} \tag{3.9}$$

We now have to face the problem of solving equations over $n_{2,E}(\mathrm{BExp})$ and $n_{2,E}(\mathit{Start})$, which represent different types of values and are mutually recursive. Because the domain of $n_{2,E}(\mathrm{BExp})$ is finite (it has at most $2^{|E|}$ elements), we can solve the equations iteratively until we reach a fixed point for both variables. In particular, we initialize all variables to the empty set and evaluate right-hand sides, so $n^{0}_{2,E}(\mathrm{BExp}) = \{(t,f)\}$ (the superscript denotes the iteration the algorithm is in). We can replace $n_{2,E}(\mathrm{BExp})$ with the value of $n^{0}_{2,E}(\mathrm{BExp})$ in the equation for $n^{1}_{2,E}(\mathit{Start})$ as follows:

$$\begin{aligned} n^{1}_{2,E}(\mathit{Start}) = {} & [\![\mathrm{IfThenElse}]\!]^{\#}_{E}(\{(t,f)\}, \{(0,0) + \lambda(3,6)\}, \\ & n^{1}_{2,E}(\mathit{Start})) \oplus \{(0,0) + \lambda(2,4)\} \\ & \oplus \{(0,0) + \lambda(3,6)\} \end{aligned} \tag{3.10}$$

At this point, we face a new problem: we need to express the abstract semantics of IfThenElse using the semi-linear set operators $\oplus$ and $\otimes$. In particular, we would like to produce a semi-linear set in which, for each vector, some components come from the semi-linear set for the then-branch (i.e., values corresponding to inputs for which the IfThenElse guard was true), and some components come from the

semi-linear set for the else-branch (i.e., values corresponding to inputs for which the IfThenElse guard was false). We overcome this problem by rewriting the above equations as follows:

$$\begin{aligned} \mathfrak{n}^1_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{t},\mathsf{t})}) &= \{(0,0) + \lambda(3,0)\} \otimes \mathfrak{n}^1_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{f},\mathsf{t})}) \\ &\oplus \{(0,0) + \lambda(2,4)\} \oplus \{(0,0) + \lambda(3,6)\} \\ \mathfrak{n}^1_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{f},\mathsf{t})}) &= \{(0,0) + \lambda(0,0)\} \otimes \mathfrak{n}^1_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{f},\mathsf{t})}) \\ &\oplus \{(0,0) + \lambda(0,4)\} \oplus \{(0,0) + \lambda(0,6)\} \end{aligned} \tag{3.11}$$

Intuitively, $\mathfrak{n}^1_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{f},\mathsf{t})})$ is the abstraction obtained by only executing the expressions generated by *Start* on the second example and leaving the output of the first example as 0 to represent the fact that only the example $\mathfrak{i}_2$ followed the else branch of the IfThenElse statement. Similarly, the semi-linear set $\{(0,0) + \lambda(3,0)\}$ zeroes out the second component of the semi-linear set appearing in the then branch. The value of $\mathfrak{n}^1_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{t},\mathsf{t})})$ (which is also the value of $\mathfrak{n}^1_{2,\mathsf{E}}(\mathit{Start})$), is then computed by summing ($\otimes$) together the then and else values. This set of equations is now in the form that we can solve automatically—i.e., it only involves the operations $\oplus$ and $\otimes$ over semi-linear sets—and thus we can compute the value of $\mathfrak{n}^1_{2,\mathsf{E}}(\mathit{Start})$. We now plug that value into the equation for BExp and compute the value of $\mathfrak{n}^1_{2,\mathsf{E}}(\mathtt{BExp})$,

$$\begin{aligned} \mathfrak{n}^1_{2,\mathsf{E}}(\mathtt{BExp}) = \{(\mathsf{t},\mathsf{f})\} &\oplus [\![\mathrm{LessThan}]\!]^{\#}_{\mathsf{E}}(\{(0,0)\}, \mathfrak{n}^1_{2,\mathsf{E}}(\mathit{Start})) \\ &\oplus [\![\mathrm{And}]\!]^{\#}_{\mathsf{E}}(\mathfrak{n}^1_{2,\mathsf{E}}(\mathtt{BExp}), \mathfrak{n}^1_{2,\mathsf{E}}(\mathtt{BExp})) \end{aligned} \tag{3.12}$$

Because $\mathfrak{n}^1_{2,\mathsf{E}}(\mathtt{BExp})$ has a finite domain, equations over such a domain can be solved iteratively, in this case yielding the fixed-point value $\mathfrak{n}^1_{2,\mathsf{E}}(\mathtt{BExp}) = \{(\mathsf{t},\mathsf{f}), (\mathsf{t},\mathsf{t}), (\mathsf{f},\mathsf{f})\}$. We now plug this solution into the equation for *Start* and compute the value of $\mathfrak{n}^2_{2,\mathsf{E}}(\mathit{Start})$ similarly to how we computed that of $\mathfrak{n}^1_{2,\mathsf{E}}(\mathit{Start})$. We then use $\mathfrak{n}^2_{2,\mathsf{E}}(\mathit{Start})$ to compute $\mathfrak{n}^2_{2,\mathsf{E}}(\mathtt{BExp})$ and discover that $\mathfrak{n}^2_{2,\mathsf{E}}(\mathtt{BExp}) = \mathfrak{n}^1_{2,\mathsf{E}}(\mathtt{BExp})$. Because we have reached a fixed point, we have found the set of possible values the grammar can output on our set of examples, i.e., the abstraction $\mathfrak{n}^1_{2,\mathsf{E}}(\mathtt{Start})$ captures all possible values the grammar G_2 can output on E. By plugging such values in the original formula similarly to what we did in Eqn. (3.6) we get that no output set

satisfies the formula on the given input examples, and therefore this SyGuS problem is unrealizable.

Grammar Flow Analysis

GFA is a formalism used for equipping the language of a grammar with a semantics in which the meaning of a tree is a value from a (complete) *combine semilattice*.

Definition 3.11 (Combine Semilattice)**.** *A* combine semilattice *is an algebraic structure* $\mathcal{D} = (D, \oplus)$*, where* $\oplus : D \times D \to D$ *is a binary operation on* D *(called "*combine*") that is commutative, associative, and idempotent.*[9]

Commutativity: *For all* $d_1, d_2 \in D$, $d_1 \oplus d_2 = d_2 \oplus d_1$.

Associativity: *For all* $d_1, d_2, d_3 \in D$, $d_1 \oplus (d_2 \oplus d_3) = (d_1 \oplus d_2) \oplus d_1$.

Idempotence: *For all* $d \in D$, $d \oplus d = d$.

A partial order, denoted by $\sqsubseteq$*, is induced on the elements of* $\mathcal{D}$ *as follows: for all* $d_1, d_2 \in D$, $d_1 \sqsubseteq d_2$ *iff* $d_1 \oplus d_2 = d_2$*. A combine semilattice is* complete *if it is closed under infinite combines.*

Definition 3.12. [*GFA*] [*MW91*, *Ram96*] *Let* $\mathcal{D} = (D, \oplus)$ *be a complete combine semilattice. Recall that in a regular-tree grammar* $G = (N, \Sigma, S, \delta)$*,* δ *is a set of productions of the form*

$$X_0 \to g(X_1, \ldots, X_k), \qquad \text{with } g \in \Sigma.$$

In a GFA problem $\mathcal{G} = (G, \mathcal{D})$*, each production is associated with a* production function $[\![\cdot]\!]^\#$ *that provides an interpretation of* g*—i.e.,* $[\![g]\!]^\# : D^k \to D$.[10] $[\![\cdot]\!]^\#$ *is extended*

[9] We have chosen to use the neutral term "combine," rather than meet or join, due to varying nomenclature in the literature. In our applications, if the semilattice is oriented according to the conventions of the abstract-interpretation literature, a combine-semilattice is a join-semilattice; if it is oriented according to the conventions of the dataflow-analysis literature, it is a meet-semilattice.

[10] The definition above is a simplified version of GFA. In the usual definition, interpretations are given via productions rather than alphabet symbols. That approach is somewhat more expressive because if two productions use the same symbol, e.g., $X_0 \to g(X_1, X_2)$ and $X_3 \to g(X_4, X_5)$, the production functions for the two productions are allowed to be different. We use the simplified definition because we do not need this ability.

to trees in $L(G)$ *in the usual way, by thinking of each tree* $e \in L(G)$ *as a term over the operations* $[\![g]\!]^\#$*. Term* e *denotes a composition of functions, and corresponds to a unique value in* D*, which we call* $[\![e]\!]^\#_{\mathcal{G}}$ *(or simply* $[\![e]\!]^\#$ *when* $\mathcal{G}$ *is understood).*

Let $L_G(X)$ *denote the trees derivable from a nonterminal* X*. The* grammar-flow-analysis problem *is to overapproximate, for each nonterminal* X*, the* combine-over-all-derivations *value* $m_{\mathcal{G}}(X)$ *defined as follows:*

$$m_{\mathcal{G}}(X) = \oplus_{e \in L_G(X)} [\![e]\!]^\#_{\mathcal{G}}.$$

We can also associate G *with a system of mutually recursive equations, where each equation has the form*

$$n_{\mathcal{G}}(X_0) =_{X_0 \to g(X_1,\ldots,X_k) \in \delta} [\![g]\!]^\#(n_{\mathcal{G}}(X_1), \ldots, n_{\mathcal{G}}(X_k)). \tag{3.13}$$

We use $n_{\mathcal{G}}(X)$ *to denote the value of nonterminal* X *in the* least fixed-point solution *of* G*'s equations.*

In essence, GFA is about two ways of folding the semantics of terms onto nonterminals:

Derivation-tree based: $m_{\mathcal{G}}(X)$ defines the semantics of a term in a compositional fashion, and folds all terms in $L_G(X)$ onto nonterminal X by combining ($\oplus$) their values.

Equational: $n_{\mathcal{G}}(X)$ obtains a value for X by using the values of "neighboring" nonterminals—i.e., nonterminals that appear on the right-hand side of productions of X.

Furthermore, GFA ensures that for all X, $m_{\mathcal{G}}(X) \sqsubseteq n_{\mathcal{G}}(X)$.

The relevance of GFA for showing unrealizability is that whenever an RTG G is recursive, $L(G)$ is an infinite set of trees. Thus, in general, there is not a clear method to compute the combine-over-all-derivations value $m_{\mathcal{G}}(X) = \bigoplus_{e \in L(G)} [\![e]\!]^\#_{\mathcal{G}}$. However, we can employ fixed-point finding procedures to compute $n_{\mathcal{G}}(X)$. Because $m_{\mathcal{G}}(X) \sqsubseteq n_{\mathcal{G}}(X)$, our computed value will be a safe overapproximation.

However, in some cases we have a stronger relationship between $m_{\mathcal{G}}(X)$ and $n_{\mathcal{G}}(X)$. A production function $[\![g]\!]^{\#}$ is *infinitely distributive* in a given argument position if

$$[\![g]\!]^{\#}(\ldots, \oplus j \in J x_j, \ldots) = \oplus_{j \in J} [\![g]\!]^{\#}(\ldots, x_j, \ldots)$$

where J is a finite or infinite index set.

Theorem 3.13. [*MW91*, *Ram96*] *If every production function* $[\![g]\!]^{\#}$, $g \in \Sigma$, *is infinitely distributive in each argument position, then for all nonterminals* X, $m_{\mathcal{G}}(X) = n_{\mathcal{G}}(X)$.[11]

This theorem is key to our decision procedures for LIA and CLIA grammars, because the domain of semi-linear sets has this property (§3.5).

Connecting GFA to Unrealizability

In this section, we show how GFA can be used to check whether a SyGuS problem with finitely many examples E is unrealizable. Intuitively, we use GFA to overapproximate the set of values the expressions generated by the grammar can yield when evaluated on a certain set of input examples E.

Definition 3.14. *Let* $sy^E = (\psi^E, G)$ *be a SyGuS problem with example set* E, *regular-tree grammar* $G = (N, \Sigma, S, \delta)$, *and background theory* T. *Let* $[\![\cdot]\!]_E$ *be the semantics of trees in* $L_G(X)$ *obtained via* T, *when* $\mu_E(\cdot)$ *is used to interpret occurrences of terminals of* G *that represent arguments to the function to be synthesized in the SyGuS problem.*

Let $\mathcal{D} = (D, \oplus)$ *be a complete combine semilattice for which there is a concretization function* $\gamma \colon D \to Val^{|E|}$, *where Val is the type of the output values produced by the function to be synthesized in the SyGuS problem. Let* $\mathcal{G}_E = (G, \mathcal{D})$ *be a GFA problem that uses* $\mu_E(\cdot)$ *to interpret occurrences of terminals of* G *that represent arguments to the function to be synthesized. Then*

[11] Thm. 3.13 generalizes other similar theorems [KU77, SP81] about the coincidence of the valuations obtained from a path-based semantics (generalized in GFA to the derivation-tree-based semantics $\{m_{\mathcal{G}}(X) \mid X \in N\}$) and an equational semantics ($\{n_{\mathcal{G}}(X) \mid X \in N\}$) when dataflow functions distribute over the combine operator.

1. $\mathcal{G}_E$ *is a* sound abstraction *of the semantics of* $L_G(X)$ *if*

$$\gamma(m_{\mathcal{G}_E}(X)) \supseteq \{[\![e]\!]_E \mid e \in L_G(X)\}.$$

2. $\mathcal{G}_E$ *is an* exact abstraction *of the semantics of* $L_G(X)$ *if*

$$\gamma(m_{\mathcal{G}_E}(X)) = \{[\![e]\!]_E \mid e \in L_G(X)\}.$$

By using such abstractions, including the one described in §3.4 based on semi-linear sets (see §3.5 and §3.6), the results obtained by solving a GFA problem can imply that a SyGuS problem with finitely many examples E is unrealizable.

The idea is that, given a SyGuS problem $sy^E = (\psi^E, G)$ with example set E, regular-tree grammar $G = (N, \Sigma, S, \delta)$, and background theory T, we can (i) solve the GFA problem $\mathcal{G}_E = (G, \mathcal{D})$ with some complete domain semilattice $\mathcal{D} = (D, \oplus)$ to obtain an overapproximation of $\gamma(m_{\mathcal{G}_E}(S))$, and then (ii) check if the approximation is disjoint from the specification, i.e., the predicate $\vec{o} \in \gamma(m_{\mathcal{G}_E}(S)) \wedge \bigwedge_{i_j \in E} \psi(\vec{o}_j, i_j)$ is unsatisfiable.

Checking that the previous predicate holds can be operationalized with the use of *symbolic concretization* [RSY04] and an SMT solver. We view an abstract domain $\mathcal{D}$ as (implicitly) a logic fragment $\mathcal{L}_{\mathcal{D}}$ of some general-purpose logic $\mathcal{L}$, and each abstract value as (implicitly) representing a formula in $\mathcal{L}_{\mathcal{D}}$. The connection between $\mathcal{D}$ and $\mathcal{L}_{\mathcal{D}}$ can be made explicit: we say that $\widehat{\gamma}$ is a *symbolic-concretization operation* for $\mathcal{D}$ if $\widehat{\gamma}(\cdot, \vec{o}) : \mathcal{D} \to \mathcal{L}_{\mathcal{D}}$ maps each $a \in \mathcal{D}$ to a formula with free variables $\vec{o}$, such that $[\![\widehat{\gamma}(a, \vec{o})]\!]_{\mathcal{L}} = \gamma(a)$. If $\widehat{\gamma}$ exists, we say that $\mathcal{L}$ *supports symbolic concretization for* $\mathcal{D}$.

Theorem 3.15. *Let* $sy^E = (\psi^E, G)$ *be a SyGuS problem with example set* E*, regular-tree grammar* $G = (N, \Sigma, S, \delta)$*, and background theory* T*. Let* $\mathcal{D} = (D, \oplus)$ *be a complete combine semilattice, and* $\mathcal{G}_E = (G, \mathcal{D})$ *be a grammar-flow-analysis problem over regular-tree grammar* G*. Assume the theory* T *supports symbolic concretization of* $\mathcal{D}$*. Let* $\mathcal{P}$ *be the*

property

$$\mathcal{P} \stackrel{\text{def}}{=} \widehat{\gamma}(n_{\mathcal{G}_E}(S), \vec{o}) \wedge \bigwedge_{i_j \in E} \psi(\vec{o}_j, i_j).$$

1. *Suppose that* $\mathcal{G}_E$ *is a* ***sound*** *abstraction of the semantics of* $L(G)$ *with respect to background theory* T*. Then* sy^E *is unrealizable* ***if*** $\mathcal{P}$ *is unsatisfiable.*

2. *Suppose that* $\mathcal{G}_E$ *is an* ***exact*** *abstraction of the semantics of* $L(G)$ *with respect to background theory* T*. Then* sy^E *is unrealizable* ***if and only if*** $\mathcal{P}$ *is unsatisfiable.*

Proof. Suppose $\widehat{\gamma}(n_{\mathcal{G}_E}(S), \vec{o}) \wedge \bigwedge_{i_j \in E} \psi(o_j, i_j)$ is unsatisfiable. By definition of symbolic concretization this means $\nexists \vec{o} \in [\![\widehat{\gamma}(n_{\mathcal{G}_E}(S), \vec{o})]\!]_{\mathcal{L}}$ such that $\bigwedge_{i_j \in E} \psi(o_j, i_j)$. Equivalently

$$\forall \vec{o} \in \gamma(n_{\mathcal{G}_E}(S)) \,.\, \bigvee_{i_j \in E} \neg\psi(o_j, i_j).$$

Since $m_{\mathcal{G}_E}(X) \sqsubseteq n_{\mathcal{G}_E}(X)$, the above implies

$$\forall \vec{o} \in \gamma(m_{\mathcal{G}_E}(S)) \,.\, \bigvee_{i_j \in E} \neg\psi(o_j, i_j).$$

Since $\mathcal{G}_E$ is a sound abstraction we have

$$\forall \vec{o} \in \{[\![e]\!]_E \mid e \in L_G(S)\} \,.\, \bigvee_{i_j \in E} \neg\psi(o_j, i_j).$$

This property means that for every possible output vector of start symbol S there is one coordinate that violates the specification. Thus, the problem is unrealizable.

Furthermore, if $\mathcal{G}_E$ is an exact abstraction, and infinitely distributive, the above properties are all equivalent. Thus, the above chain of reasoning also goes in the reverse direction. □

Algorithm for Showing Unrealizability

Algorithm 2 summarizes our strategy for showing unrealizability.

Function: CheckUnrealizable(G, ψ, E)
Input : Grammar G, specification ψ, set of examples E
$\mathcal{G}_E \leftarrow (G, \mathcal{D})$ // GFA problem from G and E (Def. 3.14) ;
$s \leftarrow n_{\mathcal{G}_E}(\mathit{Start})$ // Compute solution to the GFA problem;
if $\widehat{\gamma}(s, \vec{o}) \wedge \bigwedge_{i_j \in E} \psi(o_j, i_j)$ *is unsatisfiable* **then**
  return Unrealizable
end
return $\begin{cases} \text{Realizable}, & \mathcal{G}_E \text{ is an exact abstraction} \\ \text{Unknown}, & \text{otherwise} \end{cases}$

Algorithm 2: Checking whether sy^E is unrealizable

Example 3.16. *Recall the SyGuS problem, from §3.4, of synthesizing a function* $e_f(x) = 2x + 2$ *using the grammar from Eqn.* (3.3)*. Suppose that we call Algorithm 2 with the example set* $E = \{1\}$*, and use the abstract domain of semi-linear sets. Algorithm 2 first creates a GFA problem* $\mathcal{G}_E$*, which is shown as the recursive equation system given as Eqn.* (3.5)*. The solution of the GFA problem then gets assigned to* s *at line* (2)*. In this example,* s *is the semi-linear set* $\{0 + \lambda 3\}$*. This set can be symbolically concretized as the set of models of* $\exists \lambda \geqslant 0.o_1 = 0 + \lambda 3$*. Then, on line* (2) *the LIA formula* $\exists \lambda \geqslant 0.o_1 = 0 + \lambda 3 \wedge o_1 = 2i_1 + 2 \wedge i_1 = 1$ *is passed to an SMT solver, which will return unsat.*

GFA in Practice. So far we have been vague about how GFA problems are computationally solved. In general, there is no universal method. The performance and precision of a method depends on the choice of abstract domain $\mathcal{D}$.

Kleene iteration. Traditionally one would employ Kleene iteration to find a least fixed-point, $n_{\mathcal{G}_E}(X)$. However, Kleene iteration is only guaranteed to converge to a least fixed-point if the domain $\mathcal{D}$ satisfies the finite-ascending-chain condition. For example, the domain of predicate abstraction has this property, and therefore Algorithm 2 could be instantiated with Kleene iteration and predicate abstraction to attempt to show unrealizabilty, for arbitrary SyGuS problems. However, in the rest of this chapter, we are focused on SyGuS problems using integer arithmetic, which does have infinite ascending chains. Thus, while predicate abstraction, and other domains without infinite ascending chains, can provide a **sound** abstraction

of LIA problems, they can never provide an **exact** abstraction. Alternatively, we could still use Kleene iteration on a domain with infinite ascending chains if we provide a *widening* operator, to ensure convergence [CH78]. The issue with this strategy is that we are not guaranteed to achieve a *least* fixed-point. Such a method would still be sound, but necessarily incomplete.

Constrained Horn clauses. Another incomplete, but general, method would employ the use of the domain of constrained Horn clauses, $(\Phi, \vee)$. The set Φ contains all first-order predicates over some theory. The order of predicates is given by $P_1(\vec{v}) \leqslant P_2(\vec{v})$ iff $P_1(\vec{v}) \rightarrow P_2(\vec{v})$, for all models $\vec{v}$. The production functions $[\![\cdot]\!]^\#$ of this GFA problem get translated to constraints on the predicates. The advantage of using $(\Phi, \vee)$ is that the resulting GFA problem is a Horn-clause program, which we can then pass to an off-the-shelf, incomplete Horn-clause solver, such as the one implemented in Z3 [DMB08]. In this case, Algorithm 2 would be slightly modified. Horn-clause solvers do not provide an abstract description of the nonterminals. Instead they determine satisfiabilty of a set of Horn clauses with respect to a particular query. Therefore, in this case Algorithm 2 would use the formula in line (2) as the Horn-clause query, instead of having a separate SMT check.

Example 3.17. *The GFA problem in Eqn. (3.4) can be encoded using the following constrained Horn clause:*

$$\forall v, v'.\ \mathit{Start}(v) \leftarrow (v = 1 + 1 + 1 + v' \wedge \mathit{Start}(v')) \vee v = 0 \tag{3.14}$$

A Horn-clause solver can prove that the LIA SyGuS problem from §3.4 is unrealizable by showing that the following formula is unsatisfiable: Eqn. (3.14) $\wedge\, \mathit{Start}(o_1) \wedge o_1 = 2i_1 + 2$.

Newton's Method. In the next two sections, we provide specialized *complete* methods to solve GFA problems over LIA and CLIA grammars using Newton's method [EKL10]. Our custom methods are limited to the case of LIA and CLIA grammars, but we show that the resulting solution is exact. No prior method has this property for LIA and CLIA grammars. Consequently, our methods guarantee that not only does the check on line (2) imply unrealizability on a set of exam-

ples if the solver returns unsat, but also realizability if the solver returns sat. The latter property is important because it ensures that the current set of examples is insufficient to prove unrealizability, and we must generate more.

3.5 Proving Unrealizability of LIA SyGuS Problems with Examples

In this section, we instantiate the framework underlying Algorithm 2 to obtain a *decision procedure* for (un)realizability of SyGuS problems in *linear integer arithmetic* (LIA), where the specification is given by examples. First, we review the conditions for applying Newton's method for finding the least fixed-point of a GFA problem over a commutative, idempotent, ω-continuous semiring (§3.5). We then show that the domain of semi-linear sets can be formulated as such a problem. This approach provides a method to compute $n_{\mathcal{G}_E}(\mathit{Start})$ for LIA SyGuS problems. We then show that the domain of semi-linear sets is *exact* and *infinitely distributive* (§3.5). Finally, we show that semi-linear sets admit symbolic concretization (§3.5). Thus, by Thm. 3.15, we obtain a decision procedure for checking (un)realizability.

Solving Equations using Newton's Method

We provide background definitions on semirings and Newton's method for solving equations over certain semirings.

Definition 3.18. *A* semiring $\boldsymbol{S} = (D, \oplus, \otimes, \underline{0}, \underline{1})$ *consists of a set of* elements D *equipped with two binary operations:* combine ($\oplus$) *and* extend ($\otimes$). $\oplus$ *and* $\otimes$ *are associative, and have identity elements* $\underline{0}$ *and* $\underline{1}$*, respectively.* $\oplus$ *is commutative, and* $\otimes$ *distributes over* $\oplus$*. For every* $x \in D$, $x \otimes \underline{0} = \underline{0} = \underline{0} \otimes x$.

A semiring is commutative *if for all* $a, b \in D$, $a \otimes b = b \otimes a$.

An ω-continuous semiring *is a semiring with the following additional properties:*

1. *The relation* $\sqsubseteq \stackrel{\text{def}}{=} \{(a, b) \in D \otimes D \mid \exists d \,.\, a \oplus d = b\}$ *is a partial order.*

2. *Every* ω*-chain* $(a_i)_{i\in\mathbb{N}}$ (*i.e., for all* $i \in \mathbb{N}$ $a_i \sqsubseteq a_{i+1}$) *has a supremum* $\sup_{i\in\mathbb{N}} a_i$ *with respect to* $\sqsubseteq$.

3. *Given an arbitrary sequence* $(c_i)_{i\in\mathbb{N}}$, *define*

$$\oplus_{i\in\mathbb{N}} c_i \stackrel{\text{def}}{=} \sup\{c_0 \oplus c_1 \oplus \ldots \oplus c_i \mid i \in \mathbb{N}\}.$$

The supremum exists by (2) *above. Then, for every sequence* $(a_i)_{i\in\mathbb{N}}$, *for every* $b \in \boldsymbol{S}$, *and every partition* $(I_j)_{j\in J}$ *of* $\mathbb{N}$, *the following properties all hold:*

$$b \otimes (\oplus_{i\in\mathbb{N}} a_i) = \oplus_{i\in\mathbb{N}} (b \otimes a_i) \qquad (\oplus_{i\in\mathbb{N}} a_i) \otimes b = \oplus_{i\in\mathbb{N}} (a_i \otimes b)$$
$$\oplus_{j\in J} \left(\oplus_{i\in I_j} a_i\right) = \oplus_{i\in\mathbb{N}} a_i$$

The notation a^i *denotes the* i^{th} *term in the sequence in which* $a^0 = \underline{1}$ *and* $a^{i+1} = a^i \otimes a$.

An ω*-continuous semiring has a* Kleene-star operator $^{\circledast}\colon D \to D$ *defined as follows:* $a^{\circledast} = \oplus_{i\in\mathbb{N}} a^i$.

A semiring is idempotent *if for all* $a \in D$, $a \oplus a = a$. *In an idempotent semiring, the order on elements is defined by* $a \sqsubseteq b$ *iff* $a \oplus b = b$.

Recently, Esparza et al. [EKL10] developed an iterative method, called *Newtonian Program Analysis* (NPA), which solves a set of semiring equations by an iterative computation. The technique does not operate on the equations themselves, but on an augmented set of expressions created using a notion of a formal derivative of the expressions on the equation system's right-hand sides.

Lemma 3.19. [*Newton's Method* [*EKL10*]] *For a system of equations in* N *variables over a commutative, idempotent,* ω*-continuous semiring, NPA reaches the least fixed point after at most* $|N|$ *iterations.*

Lem. 3.19 is a powerful result because it applies even in cases when the semiring has infinite ascending chains.

Removing Non-Commutative Operators

Our first step towards using GFA to generate equations that can be solved using Newton's method removes non-commutative operators from the grammar.

We define the language LIA^+,

$$T_{\text{LIA}^+} ::= \text{Plus}(T_{\text{LIA}^+}, T_{\text{LIA}^+}) \mid \text{Num}(c) \mid \text{Var}(x) \mid \text{NegVar}(x)$$

with the following semantics with respect to examples E:

$$[\![\text{Plus}]\!]_E(v_1, v_2) := v_1 + v_2 \quad (3.15)$$
$$[\![\text{Num}(c)]\!]_E := \langle c, ..., c \rangle \quad (3.16)$$
$$[\![\text{Var}(x)]\!]_E := \mu_E(x) \quad (3.17)$$
$$[\![\text{NegVar}(x)]\!]_E := -\mu_E(x) \quad (3.18)$$

We say that a regular-tree grammar is an LIA^+ grammar if all of its symbols are in the alphabet $\{\text{Plus}, \text{Num}(c), \text{Var}(x), \text{NegVar}(x)\}$.

We next show how any LIA grammar can be rewritten into an LIA^+ grammar that accepts terms that are semantically equivalent to those in the original grammar. We introduce a grammar-rewriting function h that recursively pushes negations to the leaves of the terms in an LIA grammar G, to produce an LIA^+ grammar $h(G)$ that does not contain the Minus symbol. Given an LIA grammar $G = (N, \Sigma, S^{\text{LIA}}, \delta)$, we define the rewritten grammar $h(G)$ as the tuple $(N \cup N^-, \Sigma^{\text{LIA}^+}, S, \delta^-)$ where δ^- is defined as follows. For every production $X \to \alpha \in \delta$:

- If $\alpha = \text{Plus}(X_1, X_2)$, then δ^- contains the productions $X^- \to \text{Plus}(X_1^-, X_2^-)$ and $X \to \text{Plus}(X_1, X_2)$;
- If $\alpha = \text{Minus}(X_1, X_2)$, then δ^- contains the productions $X^- \to \text{Plus}(X_1^-, X_2)$ and $X \to \text{Plus}(X_1, X_2^-)$;

- If $\alpha = \mathrm{Num}(c)$, then δ^- contains the productions $X \to \mathrm{Num}(c)$ and $X^- \to \mathrm{Num}(-c)$.
- If $\alpha = \mathrm{Var}(x)$, then δ^- contains the productions $X \to \mathrm{Var}(x)$ and $X^- \to \mathrm{NegVar}(x)$.

It is trivial to see that the grammar $h(G)$ only produces terms in LIA^+.

Example 3.20. *Consider the LIA grammar* G*:*

$$\mathit{Start} \ ::= \ \mathit{Minus}(\mathit{Start}, \mathit{Start}) \mid 1 \mid x$$

The following LIA^+ *grammar* $h(G)$ *is equivalent to* G*:*

$$\begin{aligned} \mathit{Start} &::= \mathit{Plus}(\mathit{Start}, \mathit{Start}^-) \mid \mathit{Num}(1) \mid \mathit{Var}(x) \\ \mathit{Start}^- &::= \mathit{Plus}(\mathit{Start}^-, \mathit{Start}) \mid \mathit{Num}(-1) \mid \mathit{NegVar}(x). \end{aligned}$$

The following lemma shows that the original and the rewritten grammars produce semantically equivalent terms.

Lemma 3.21. *An LIA grammar* G *is semantically equivalent to the* LIA^+ *grammar* $h(G)$*, i.e.,*

$$(\forall e \in L(G) \exists e' \in L(h(G)).[\![e]\!] = [\![e']\!]) \tag{3.19}$$

$$\wedge (\forall e' \in L(h(G)) \exists e \in L(G).[\![e]\!] = [\![e']\!])\,. \tag{3.20}$$

Proof. We start by proving the following result, which states that the terms produced by some nonterminal X in G are equivalent to terms produced by the corresponding nonterminal X in $h(G)$, and to the negation of terms produced by the corresponding negative nonterminal X^- in $h(G)$:
(i) $\big(\forall e \in L_G(X) \exists e' \in L_{h(G)}(X).[\![e]\!] = [\![e']\!]\big) \wedge \big(\forall e' \in L_{h(G)}(X) \exists e \in L_G(X).[\![e]\!] = [\![e']\!]\big)$, and
(ii) $\big(\forall e \in L_G(X) \exists e' \in L_{h(G)}(X^-).[\![e]\!] = [\![e']\!]\big) \wedge \big(\forall e' \in L_{h(G)}(X^-) \exists e \in L_G(X).[\![e]\!] = -[\![e']\!]\big)$.

We proceed by induction on e. The base cases are $e = \text{Num}(c)$ and $e = \text{Var}(x)$. According to the definition of h, there exists productions $X \to \text{Num}(c)$ (resp., $X \to \text{Var}(x)$) and $X \to \text{Num}(-c)$ (resp., $X \to \text{NegVar}(x)$) in $h(G)$. Note that $[\![\text{Num}(c)]\!] = -[\![\text{Num}(-c)]\!]$ and $[\![\text{Var}(x)]\!] = -[\![\text{NegVar}(x)]\!]$. Hence, the property holds for the base cases.

Now the induction step is

- Assume $e = \text{Plus}(e_1, e_2)$ where e_1 and e_2 are terms produced by nonterminals X_1 and X_2, respectively. According to the induction hypothesis, X_1 in $h(G)$ can produce a term e_1' equivalent to e_1 and X_2 in $h(G)$ can produce a term e_2' equivalent e_2. Therefore the nonterminal X in $h(G)$ can produce $\text{Plus}(e_1', e_2')$ whose semantics is equivalent to e. The analysis for X^- in $h(G)$ is similar.

- Assume $e = \text{Minus}(e_1, e_2)$ where e_1 and e_2 are terms produced by nonterminals X_1 and X_2, respectively. According to the induction hypothesis, X_1 in $h(G)$ can produce a term e_1' equivalent to e_1 and X_2^- in $h(G)$ can produce a term e_2' such that $[\![e_2']\!] = -[\![e_2]\!]$ Therefore the nonterminal X in $h(G)$ can produce $\text{Plus}(e_1', e_2')$ whose semantic is equivalent to e, i.e., $[\![\text{Plus}(e_1', e_2')]\!] = [\![e_1']\!] - [\![e_2]\!] = [\![\text{Minus}(e_1, e_2)]\!]$. The analysis for X^- in $h(G)$ is similar.

Finally, terms produced by *Start* in G are semantically equivalent to terms produced by *Start* in $h(G)$, and hence G is semantically equivalent to $h(G)$ □

Grammar Flow Analysis using Semi-Linear Sets

Thanks to §3.5, we can assume that the SyGuS grammar G only produces LIA^+ terms. In this section, we use grammar-flow analysis to generate equations such that the solutions to the equations assign a semi-linear set to each nonterminal X that, for the finitely many examples in E, *exactly* describes the set of possible values produced by any term in $L_G(X)$.

We start by defining the complete combine semilattice $(\mathcal{SL}, \oplus)$ of *semi-linear sets*. We then use them, together with the set of examples E, to define a specific family

of GFA problems: $\mathcal{G}_E = (G, \mathcal{SL})$, where $G = (N, \Sigma, S, \delta)$ is an LIA$^+$ grammar. For simplicity, we use notation $\mathcal{SL}$ for both the semilattice and its domain

In the terminology of abstract interpretation, $\mathcal{SL}$ is an abstract domain that we can use to represent, for every nonterminal X, the set of possible output vectors produced by evaluating each term in $L_G(X)$ on the examples in E. Moreover, the representation is *exact*; i.e., $\gamma(m_{\mathcal{G}_E}(X)) = \{[\![e]\!]_E \mid e \in L_G(X)\}$ where γ denotes the usual operation of concretization.

Definition 3.22 (Semi-linear Set)**.** *A linear set* $\langle \vec{u}, \{\vec{v}_1, \cdots, \vec{v}_n\}\rangle$ *denotes the set of integer vectors* $\{\vec{u} + \lambda_1\vec{v}_1 + \cdots + \lambda_n\vec{v}_n \mid \lambda_1, \ldots, \lambda_n \in \mathbb{N}\}$*, where* $\vec{u}, \vec{v}_1, ..., \vec{v}_n \in \mathbb{Z}^d$ *and* d *is the dimension of the linear set. A semi-linear set is a finite union* $\bigcup_i \langle \vec{u}_i, V_i\rangle$ *of linear sets, also denoted by* $\{\langle \vec{u}_i, V_i\rangle\}_i$*.*

The concretization *of a semi-linear set* $sl = \{\langle \vec{u}_i, V_i\rangle\}_i$*, denoted by* $\gamma(sl)$*, is the set of vectors*

$$\bigcup_i \{\vec{u}_i + \lambda_{1,i}\vec{v}_{1,i} + \cdots + \lambda_{n,i}\vec{v}_{n,i} \mid \lambda_{1,i}, \ldots, \lambda_{n,i} \in \mathbb{N}\}.$$

Semi-linear sets were originally used in a well-known result in formal-language theory: Parikh's theorem [Par66]. Parikh's theorem states that, given a context-free grammar G with terminals $(t_1, \ldots, t_n)$, if one looks only at the number of occurrences of each terminal symbol in each word in a context-free language, without regard to their order—i.e., each word w is represented by a vector $v_w = \langle c_1, \ldots, c_n\rangle$, which denotes that each terminal t_i appears exactly c_i times in w—the set of vectors $\{v_w \mid w \in L(G)\}$ is representable by a semi-linear set. If a grammar for an LIA SyGuS problem only uses addition (which is a commutative operation), we can represent any term in the language of the grammar by simply counting the number of times each terminal (i.e., a constant or a variable) appears in the term. Consequently, we can use a domain of values similar to the ones used in Parikh's theorem to represent the set of possible terms (or, more precisely, their semantics) as a semi-linear set.

While the details of Parikh's theorem are not relevant to this chapter, the core idea behind its proof is that grammars over commutative operators can be transformed into regular languages and therefore regular expressions. Then, to compute

the set of all possible count vectors that the grammar can produce one needs to "evaluate" the regular expressions using operators analogous to the regular-expression concatenation, union, and star. For semi-linear sets, these operators are $\otimes$, $\oplus$ and $\circledast$, defined as follows [BET03]:

$$\begin{aligned}
\{\langle \vec{u}_{1,i}, V_{1,i}\rangle\}_i \oplus \{\langle \vec{u}_{2,j}, V_{2,j}\rangle\}_j &= \{\langle \vec{u}_{1,i}, V_{1,i}\rangle\}_i \cup \{\langle \vec{u}_{2,j}, V_{2,j}\rangle\}_j \\
\{\langle \vec{u}_{1,i}, V_{1,i}\rangle\}_i \otimes \{\langle \vec{u}_{2,j}, V_{2,j}\rangle\}_j &= \bigcup_{i,j} \{\langle \vec{u}_{1,i} + \vec{u}_{2,j}, V_{1,i} \cup V_{2,j}\rangle\} \\
(\{\langle \vec{u}_i, V_i\rangle\}_i)^{\circledast} &= \{\langle \vec{0}, \bigcup_i (\{\vec{u}_i\} \cup V_i)\rangle\}
\end{aligned} \tag{3.21}$$

The semi-linear sets $\mathbf{0} \stackrel{\text{def}}{=} \emptyset$ and $\mathbf{1} \stackrel{\text{def}}{=} \{\langle \vec{0}, \emptyset\rangle\}$ are the identity elements for $\oplus$ and $\otimes$, respectively. We use $(\mathcal{SL}, \oplus)$ to denote the complete combine semilattice of semi-linear sets with the least element $\mathbf{0}$.

We define the GFA problem $\mathcal{G}_E = (G, \mathcal{SL})$ by giving the following interpretations to LIA$^+$ operators:

$$[\![\text{Plus}]\!]^\#_E(sl_1, sl_2) = sl_1 \otimes sl_2 \tag{3.22}$$

$$[\![\text{Num}(c)]\!]^\#_E = \{\langle \langle c, \cdots, c\rangle, \emptyset\rangle\} \tag{3.23}$$

$$[\![\text{Var}(x)]\!]^\#_E = \{\langle \mu_E(x), \emptyset\rangle\} \tag{3.24}$$

$$[\![\text{NegVar}(x)]\!]^\#_E = \{\langle -\mu_E(x), \emptyset\rangle\} \tag{3.25}$$

Now consider the combine-over-all-derivations value $m_{\mathcal{G}_E}(X) = \bigoplus_{e \in L_G(X)} [\![e]\!]^\#_E$ for the grammar-flow-analysis problem $\mathcal{G}_E$. For an arbitrary tree $e \in L_G(X)$, in the computation of $[\![e]\!]^\#_E$ via Eqns. (3.22)–(3.25), there is never any use of the $\oplus$ operation of $\mathcal{SL}$. Consequently, the computation of $[\![e]\!]^\#_E$ produces a semi-linear set that consists of a *single vector*—the same vector, in fact, that is produced by the computation of $[\![e]\!]_E$ shown in Ex. **??**. In particular, $\oplus$ two lines above Eqn. (3.21) preserves singleton sets, and hence for singleton sets, $\otimes$ one line above Eqn. (3.21) emulates $[\![\text{Plus}]\!]_E$. Therefore, the combine-over-all-derivations value $m_{\mathcal{G}_E}(X) = \bigoplus_{e \in L_G(X)} [\![e]\!]^\#_E$ is exactly the set of vectors $\{[\![e]\!]_E \mid e \in L_G(X)\}$. In other words,

$m_{\mathcal{G}_E}(X)$ is an *exact* abstraction of the $[\![\cdot]\!]_E$ semantics of the terms in $L_G(X)$, i.e., $\gamma(m_{\mathcal{G}_E}(X)) = \{[\![e]\!]_E \mid e \in L_G(X)\}$. Because $[\![\text{Plus}]\!]^\#_E$ is infinitely distributive over $\oplus$ ([EKL10, Defn. 2.1 and §2.3.3]), $m_{\mathcal{G}_E}(X) = n_{\mathcal{G}_E}(X)$ holds by Thm. 3.13, and thus we can compute $m_{\mathcal{G}_E}(X)$ by solving a set of equations in which, for each $X_0 \in N$, there is an equation of the form

$$n_{\mathcal{G}_E}(X_0) = \oplus_{X_0 \to g(X_1,\ldots,X_k) \in \delta} [\![g]\!]^\#_E(n_{\mathcal{G}_E}(X_1), \ldots, n_{\mathcal{G}_E}(X_k)). \tag{3.26}$$

The argument given in the previous paragraph is captured by the following lemma:

Lemma 3.23. *Given an LIA$^+$ grammar $G = (N, \Sigma, S, \delta)$, a finite set of examples E, $\mathcal{G}_E = (G, \boldsymbol{S})$ is an exact abstraction of the semantics of the languages $L_G(X)$, for all $X \in N$ (with respect to LIA and E).*

Proof. We can show that for any expression e, the abstract semantics $[\![e]\!]^\#_E$ is always a singleton set $\{[\![e]\!]_E\}$, where the element of the singleton set is exactly the semantics of e. For an arbitrary tree $e \in L_G(X)$, in the computation of $[\![e]\!]^\#_E$ via Eqns. (3.22)–(3.25), there is never any use of the $\oplus$ operation of $\mathcal{SL}$. Consequently, the computation of $[\![e]\!]^\#_E$ produces a semi-linear set that consists of a *single vector*—the same vector, in fact, that is produced by the computation of $[\![e]\!]_E$ via Eqns. (3.15)–(3.18). In particular, Eqn. (3.21) preserves singleton sets, and hence for singleton sets, Eqn. (3.21) emulates Eqn. (3.15). Therefore, the combine-over-all-derivations value $m_{\mathcal{G}_E}(X) = \bigoplus_{e \in L_G(X)} [\![e]\!]^\#_E$ is exactly the set of vectors $\{[\![e]\!]_E \mid e \in L_G(X)\}$. In other words, $m_{\mathcal{G}_E}(X)$ is an *exact* abstraction of the $[\![\cdot]\!]_E$ semantics of the terms in $L_G(X)$, i.e., $\gamma(m_{\mathcal{G}_E}(X)) = \{[\![e]\!]_E \mid e \in L_G(X)\}$.

Therefore, $\mathcal{G}_E$ is an exact abstraction of the semantics of $L_G(X)$. □

Example 3.24. *Consider again the LIA$^+$ grammar G_1 from Eqn. (3.3), written out in the expanded form given in footnote 5:*

$$\begin{array}{ll} \mathit{Start} ::= \mathit{Plus}(\mathit{S1}, \mathit{Start}) \mid \mathit{Num}(0) & \mathit{S1} ::= \mathit{Plus}(\mathit{S2}, \mathit{Var}(x)) \\ \mathit{S2} ::= \mathit{Plus}(\mathit{S3}, \mathit{Var}(x)) & \mathit{S3} ::= \mathit{Var}(x). \end{array}$$

Let E *be* $\{1,2\}$*, and thus* $\mu_E(x) = \langle 1,2\rangle$*. The equation system for the GFA problem* $\mathcal{G}_{1E}$ *is as follows:*

$$\begin{aligned}
&n_{\mathcal{G}_{1E}}(\mathit{Start}) = n_{\mathcal{G}_{1E}}(\mathit{S1}) \otimes n_{\mathcal{G}_{1E}}(\mathit{Start}) \oplus \{\langle (0,0), \emptyset\rangle\} \\
&n_{\mathcal{G}_{1E}}(\mathit{S1}) = n_{\mathcal{G}_{1E}}(\mathit{S2}) \otimes \{\langle (1,2), \emptyset\rangle\} \\
&n_{\mathcal{G}_{1E}}(\mathit{S2}) = n_{\mathcal{G}_{1E}}(\mathit{S3}) \otimes \{\langle (1,2), \emptyset\rangle\} \quad n_{\mathcal{G}_{1E}}(\mathit{S3}) = \{\langle (1,2), \emptyset\rangle\}
\end{aligned}$$

which has the solution

$$\begin{aligned}
n_{\mathcal{G}_{1E}}(\mathit{Start}) &= \{\langle (0,0), \{(3,6)\}\rangle\} \qquad n_{\mathcal{G}_{1E}}(\mathit{S2}) = \{\langle (2,4), \emptyset\rangle\} \\
n_{\mathcal{G}_{1E}}(\mathit{S1}) &= \{\langle (3,6), \emptyset\rangle\} \qquad n_{\mathcal{G}_{1E}}(\mathit{S3}) = \{\langle (1,2), \emptyset\rangle\}.
\end{aligned}$$

The concretizations of semi-linear sets in the solution are

$$\begin{aligned}
\gamma(n_{\mathcal{G}_{1E}}(\mathit{Start})) &= \{(0,0) + \lambda(3,6) \mid \lambda \in \mathbb{N}\} \\
\gamma(n_{\mathcal{G}_{1E}}(\mathit{S1})) &= \{(3,6)\} \qquad \gamma(n_{\mathcal{G}_{1E}}(\mathit{S2})) = \{(2,4)\} \\
\gamma(n_{\mathcal{G}_{1E}}(\mathit{S3})) &= \{(1,2)\}.
\end{aligned}$$

The following proposition shows that the equations generated in Eqn. (3.26) can be solved using Newton's method.

Proposition 3.25. $(\mathcal{SL}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ *is a commutative, idempotent,* ω*-continuous semiring.*

Moreover, $(\mathcal{SL}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ has infinite ascending chains; consequently, Lem. 3.19 is directly relevant to our setting. Henceforth, we use the term "semiring"—and symbol **S**—to mean a *commutative, idempotent,* ω*-continuous semiring*.

Before concluding this section, we analyze the size of the semi-linear set computed by the NPA method when solving equations generated by LIA$^+$ grammars. For a semi-linear set $sl = \{\langle \vec{u}_i, V_i\rangle_i\}$, let its *size* be $\sum_i (|V_i| + 1)$.

Given an LIA grammar, a finite set of examples E and a nonterminal $X \in N$, the semi-linear set $n_{\mathcal{G}_E}(X)$ yielded by NPA can contain exponentially many linear sets [KT10].

Checking Unrealizability

We now show how symbolic concretization for $\mathcal{SL}$ can be used to prove that no element $\vec{o}$ in $n_{\mathcal{G}}(\mathit{Start})$ satisfies the specification $\psi^{E}(\vec{o})$ of the SyGuS problem. The logic LIA supports symbolic concretization for $\mathcal{SL}$. For instance, for a linear set $\{\langle \vec{u}, \{\vec{v}_1, \ldots, \vec{v}_n\}\rangle\}$, its symbolic concretization $\widehat{\gamma}(\langle \vec{u}, \{\vec{v}_1, \ldots, \vec{v}_n\}\rangle, \vec{o})$ is defined as follows:

$$\exists \lambda_1 \in \mathbb{N}, \ldots, \lambda_n \in \mathbb{N}. (\vec{o} = \vec{u} + \lambda_1 \vec{v}_1 + \cdots + \lambda_n \vec{v}_n).$$

Thus, the symbolic concretization for a semi-linear set is:

$$\widehat{\gamma}(\{\langle \vec{u}_i, V_i\rangle\}_i, \vec{o}) \stackrel{\text{def}}{=} \bigvee_i \widehat{\gamma}(\langle \vec{u}_i, V_i\rangle, \vec{o}). \tag{3.27}$$

Note that $\vec{o}$ is shared among all disjuncts. The set of satisfying assignments to $\vec{o}$ consist of exactly the vectors in $\gamma(\{\langle \vec{u}_i, V_i\rangle\}_i)$.

Our decidability result follows directly from Thm. 3.15.

Theorem 3.26. *Given an LIA SyGuS problem* sy *and a finite set of examples* E*, it is decidable whether the SyGuS problem* sy^{E} *is realizable.*

Proof. We have shown that $n_{\mathcal{G}_E}(x)$ is an exact abstraction for LIA grammars (Lemma 3.23) and LIA supports symbolic concretization (Eqn. (3.27)). According to Thm. 3.15, $\mathcal{G}_E = (G, \mathbf{S})$ is sound and complete for proving unrealizability of LIA SyGuS problems for finitely example, and hence decidable. □

3.6 Proving Unrealizability of CLIA SyGuS Problems with Examples

In this section, we instantiate the framework from §3.4 to obtain a *decision procedure* for realizability of SyGuS problems in *conditional linear integer arithmetic* (CLIA), where the specification is given by examples. The decision procedure follows the same steps as the one for LIA in §3.5. The main difference is a technique for

solving equations generated from grammars that involve both Boolean and integer operations.

Conditional Linear Integer Arithmetic

The grammar of all CLIA terms is the following:

$$\begin{array}{rcl} T_{\mathbb{Z}} & ::= & \mathrm{IfThenElse}(T_{\mathbb{B}}, T_{\mathbb{Z}}, T_{\mathbb{Z}}) \mid \mathrm{Plus}(T_{\mathbb{Z}}, T_{\mathbb{Z}}) \\ & \mid & \mathrm{Minus}(T_{\mathbb{Z}}, T_{\mathbb{Z}}) \mid \mathrm{Num}(c) \mid \mathrm{Var}(x) \\ T_{\mathbb{B}} & ::= & \mathrm{And}(T_{\mathbb{B}}, T_{\mathbb{B}}) \mid \mathrm{Not}(T_{\mathbb{B}}) \mid \mathrm{LessThan}(T_{\mathbb{Z}}, T_{\mathbb{Z}}) \end{array}$$

where $c \in \mathbb{Z}$ is a constant and $x \in \mathcal{V}$ is a input variable to the function being synthesized. Notice that the definitions of $T_{\mathbb{Z}}$ and $T_{\mathbb{B}}$ are mutually recursive. [12] The example grammar presented in Eqn. (3.7) in §3.4 is a CLIA grammar.

We now define the semantics of CLIA terms. Given an integer vector $\vec{v} \in \mathbb{Z}^d$ and a Boolean vector $\vec{b} \in \mathbb{B}^d$, let $\textsc{proj}_{\vec{\mathbb{Z}}}(\vec{v}, b)$ be the integer vector obtained by keeping the vector elements of $\vec{v}$ corresponding to the indices for which $\vec{b}$ is true, and zeroing out all other elements:

$$\begin{array}{l} \textsc{proj}_{\vec{\mathbb{Z}}}(\langle u_1, \ldots, u_d \rangle, \langle b_1, \ldots, b_d \rangle) \\ \quad = \langle \mathrm{if}(b_1) \text{ then } u_1 \text{ else } 0, \ldots, \mathrm{if}(b_d) \text{ then } u_d \text{ else } 0 \rangle \end{array}$$

The semantics of symbols that are not in LIA is as follows:

$$\begin{array}{rcl} [\![\mathrm{IfThenElse}]\!]_E(\vec{b}, \vec{v_1}, \vec{v_2}) & = & \textsc{proj}_{\vec{\mathbb{Z}}}(\vec{v_1}, \vec{b}) + \textsc{proj}_{\vec{\mathbb{Z}}}(\vec{v_2}, \neg\vec{b}) \\ [\![\mathrm{Not}]\!]_E(\vec{b}) = \neg\vec{b} \qquad [\![\mathrm{And}]\!]_E(\vec{b_1}, \vec{b_2}) = \vec{b_1} \wedge \vec{b_2} & & \\ [\![\mathrm{LessThan}]\!]_E(\vec{v_1}, \vec{v_2}) & = & \vec{v_1} < \vec{v_2} \end{array}$$

where the operations $+$, $\wedge$, $<$, and $\neg$ are performed element-wise—e.g., $\vec{u} < \vec{v} = \langle b_1, \ldots, b_n \rangle$ such that $b_i \Leftrightarrow u_i < v_i$.

[12] For any SyGuS problem over CLIA terms, the goal will be to synthesize a term with a specific type—i.e., either $\mathbb{B}$ or $\mathbb{Z}$.

Similarly to what we did in §3.5, any CLIA grammar G can be rewritten into an equivalent CLIA$^+$ grammar $h(G)$ that does not contain any occurrences of Minus, but may contain the symbol NegVar.

The rest of the section is organized as follows. First, we present the abstract domains used to represent Boolean and integer terms (§3.6). Second, we show how to compute an exact abstraction of Boolean nonterminals in grammars without IfThenElse (§3.6). Third, we show how to solve SyGuS problems with CLIA grammars containing arbitrary operators, in particular IfThenElse and mutual recursion (§3.6).

Abstract Semantics for CLIA

We use sets of Boolean vectors as the abstract domain for Boolean nonterminals, and semi-linear sets as the abstract domain for integer nonterminals. We use b to denote a Boolean vector and *bset* to denote sets of Boolean vectors.

Given a semi-linear set $sl \in \mathcal{SL}$ and a Boolean vector $\vec{b} \in \mathbb{B}^d$, let $\textsc{proj}_{\mathcal{SL}}(sl, \vec{b})$ be the semi-linear set obtained by zeroing out for each vector in sl the elements at all index positions for which $\vec{b}$ is false:

$$\begin{aligned}
\textsc{proj}_{\mathcal{SL}}(\{\langle \vec{u}_i, \Omega_i \rangle\}_i, \vec{b}) &= \{\textsc{proj}_{\mathbf{S}}(\langle \vec{u}_i, \Omega_i \rangle, \vec{b})\}_i \\
\textsc{proj}_{\mathbf{S}}(\langle \vec{u}, \{\vec{v}_1, ..., \vec{v}_n\} \rangle, \vec{b}) &= \langle \textsc{proj}_{\vec{\mathbb{Z}}}(\vec{u}, \vec{b}), \{\textsc{proj}_{\vec{\mathbb{Z}}}(\vec{v}_i, \vec{b})\}_i \rangle
\end{aligned}$$

Next, we lift the concrete semantics to semi-linear sets and define the abstract semantics of CLIA operators that are not in LIA.

$$\begin{aligned}
[\![\text{IfThenElse}]\!]^\#_E(bset, sl_1, sl_2) &= \\
&\quad \bigoplus_{\vec{b} \in bset} \textsc{proj}_{\mathcal{SL}}(sl_1, \vec{b}) \otimes \textsc{proj}_{\mathcal{SL}}(sl_2, \neg\vec{b}) \\
[\![\text{LessThan}]\!]^\#_E(sl_1, sl_2) &= \{v_1 < v_2 \mid v_1 \in sl_1, v_2 \in sl_2\} \\
[\![\text{Not}]\!]^\#_E(bset) &= \bigcup_{\vec{b} \in bset} \{\neg\vec{b}\} \\
[\![\text{And}]\!]^\#_E(bset_1, bset_2) &= \bigcup_{\vec{b_1} \in bset_1, \vec{b_2} \in bset_2} \{\vec{b_1} \wedge \vec{b_2}\}
\end{aligned}$$

Example 3.27. *Consider a set of Boolean vectors* $bset := \{(t, f), (t, t)\}$

and two semi-linear sets $\mathrm{sl}_1 := \{\langle (1,2), \{(3,4)\}\rangle\}$ *and* $\mathrm{sl}_2 := \{\langle (5,6), \{(7,8)\}\rangle\}$. *Then* (i) $[\![\mathit{Not}]\!]^\#_{\mathsf{E}}(\mathit{bset}) = \{(f,t),(f,f)\}$, *and* (ii) $[\![\mathit{LessThan}]\!]^\#_{\mathsf{E}}(\mathrm{sl}_1, \mathrm{sl}_2) = \{(t,t),(t,f),(f,f)\}$ *since* $(1,2) < (5,6) = (t,t)$, $(1,2) + (3,4) < (5,6) = (t,f)$ *and* $(1,2) + 2(3,4) < (5,6) = (f,f)$. *Finally,*

$$
\begin{aligned}
& [\![\mathit{IfThenElse}]\!]^\#_{\mathsf{E}}(\mathit{bset}, \mathrm{sl}_1, \mathrm{sl}_2) \\
= \; & \{\langle (1,0), \{(3,0)\}\rangle\} \otimes \{\langle (0,6), \{(0,8)\}\rangle\} \\
& \oplus \{\langle (1,2), \{(3,4)\}\rangle\} \otimes \{\langle (0,0), \{(0,0)\}\rangle\} \\
= \; & \{\langle (1,6), \{(3,0),(0,8)\}\rangle, \langle (1,2), \{(3,4),(0,0)\}\rangle\}
\end{aligned}
$$

□

Operationally, the semantics of the LessThan symbol can be implemented using an SMT solver. As shown in §3.5, a semi-linear set sl can be symbolically concretized as a formula $\widehat{\gamma}(\mathrm{sl}, \vec{\alpha})$ in LIA (a decidable SMT theory). Therefore, the set $[\![\mathrm{LessThan}]\!]^\#_{\mathsf{E}}(\mathrm{sl}_1, \mathrm{sl}_2) = \mathit{bset}$ can be computed by performing $2^{|\mathsf{E}|}$ SMT queries—i.e., for every Boolean vector $\vec{b} = \langle b_1, \ldots, b_{|\mathsf{E}|}\rangle$, we have that $\vec{b} \in \mathit{bset}$ iff the following formula is satisfiable: $\widehat{\gamma}(\mathrm{sl}_1, \vec{\alpha}_1) \wedge \widehat{\gamma}(\mathrm{sl}_2, \vec{\alpha}_2) \wedge \vec{b} = \vec{\alpha_1} < \vec{\alpha_2}$.

Similarly to how we defined $[\![\cdot]\!]^\#_{\mathsf{E}}$ for multisorted terms, we overload $\oplus$ as the union of sets of Boolean vectors, and define a multisorted semilattice $\mathcal{D}_{\mathrm{CLIA}^+} := (2^{\mathbb{B}} \uplus \mathcal{SL}, \oplus)$ over sets of Boolean vectors and semi-linear sets. We use $\mathcal{G}^{\mathrm{CLIA}+}_{\mathsf{E}} := (\mathsf{G}, \mathcal{D}_{\mathrm{CLIA}^+})$ to denote the GFA problem for a CLIA^+ grammar G and finitely many examples E. $\mathcal{G}^{\mathrm{CLIA}+}_{\mathsf{E}}$ is an exact abstraction of the semantics of CLIA^+ grammars.

Lemma 3.28. *Given* CLIA^+ *grammar* $\mathsf{G} = (N, \Sigma, S, \delta)$, *finite set of examples* E, $\mathcal{G}^{\mathrm{CLIA}+}_{\mathsf{E}}$ *is an exact abstraction of the semantics of the languages* $\mathsf{L}_{\mathsf{G}}(X)$, *for all* $X \in N$ (*with respect to LIA and* E).

Proof. Using a similar argument as in §3.5, we can show that for any expression e, the abstract semantics $[\![e]\!]^\#_{\mathsf{E}}$ is always a singleton set $\{[\![e]\!]_{\mathsf{E}}\}$, where the element of the singleton set is exactly the semantics of e. Therefore, $\mathfrak{m}_{\mathcal{G}^{\mathrm{CLIA}+}_{\mathsf{E}}} = \bigoplus_{e \in \mathsf{L}_{\mathsf{G}}(X)} [\![e]\!]^\#_{\mathsf{E}}$ is exactly $\{[\![e]\!]_{\mathsf{E}} \mid e \in \mathsf{L}_{\mathsf{G}}(X)\}$. □

CLIA Equations without Mutual Recursion

A CLIA grammar G contains Boolean and integer nonterminals. A nonterminal X is a Boolean nonterminal if $[\![X]\!] \in \mathbb{B}$, and is an integer nonterminal if $[\![X]\!] \in \mathbb{Z}$. In this subsection, we assume that there exists no mutual recursion, i.e., G contains no IfThenElse productions. Under this assumption, the only operator that connects Boolean nonterminals and integer nonterminals is LessThan, and hence no Boolean nonterminal appears in the productions of an integer nonterminal. Therefore, we can proceed by first solving the equations that involve integer nonterminals, using the technique presented in §3.5, and then plugging the corresponding values into the equations that involve Boolean nonterminals.

Example 3.29. *Consider the following grammar* $\mathsf{G}_{\mathfrak{b}}$*:*

$$\begin{aligned} \mathsf{BExp} ::= {} & \mathit{LessThan}(\mathsf{X}, \mathsf{N2}) \mid \mathit{LessThan}(\mathsf{N0}, \mathsf{Exp}) \\ \mid {} & \mathit{And}(\mathsf{BExp}, \mathsf{BExp}) \\ \mathsf{Exp} ::= {} & \mathit{Plus}(\mathsf{X}, \mathsf{Exp}) \mid \mathit{Num}(0) \qquad \mathsf{X} ::= \mathit{Var}(\mathrm{x}) \\ \mathsf{N0} ::= {} & \mathit{Num}(0) \qquad \mathsf{N2} ::= \mathit{Num}(2) \end{aligned} \tag{3.28}$$

Assume that the given set of examples is $\mathsf{E} = \{1, 2\}$*. If we consider the equations generated by grammar flow analysis for this grammar, all the variables corresponding to the integer nonterminals* $\mathsf{Exp}, \mathsf{X}, \mathsf{N0}, \mathsf{N2}$ *do not depend on any of the variables for the Boolean nonterminals. Therefore, we can solve the corresponding set of equations using the techniques presented in §3.5. For each such nonterminal* X*, by plugging the value of each* $\mathfrak{n}_{\mathcal{G}_{\mathsf{E}}^{CLIA+}}(\mathsf{X})$ *in the equations corresponding to* BExpr *we get the following equation:*

$$\begin{aligned} \mathfrak{n}_{\mathcal{G}_{\mathsf{E}}^{CLIA+}}(\mathsf{BExp}) = {} & \{(t, f)\} \oplus \{(t, t), (f, f)\} \\ & \oplus [\![\mathit{And}]\!]^{\#}(\mathfrak{n}_{\mathcal{G}_{\mathsf{E}}^{CLIA+}}(\mathsf{BExp}), \mathfrak{n}_{\mathcal{G}_{\mathsf{E}}^{CLIA+}}(\mathsf{BExp})) \end{aligned} \tag{3.29}$$

where $\oplus$ *is the set union operator.* □

After this step, we are left with a set of equations $eqs_{\mathbb{B}}$ that involve only Boolean nonterminals and Boolean symbols. Concretely, for every nonterminal X in the set

of Boolean nonterminals $N_{\mathbb{B}}$, $eqs_{\mathbb{B}}$ contains an equation

$$n_{\mathcal{G}_E^{CLIA+}}(X) = \bigoplus_{X \to g(X_1,\ldots,X_k) \in \delta} [\![g]\!]_E^{\#}(n_{\mathcal{G}_E^{CLIA+}}(X_1), \ldots, n_{\mathcal{G}_E^{CLIA+}}(X_k)) \tag{3.30}$$

Because the domain of sets of Boolean vectors is finite, the least fixed point of $eqs_{\mathbb{B}}$ can be found using an algorithm SolveBool that iteratively computes finer under-approximations of $n_{\mathcal{G}_E^{CLIA+}}$ as $n^k_{\mathcal{G}_E^{CLIA+}}$—i.e., the under-approximation at iteration k—until it reaches the least fixed point, which—by Thm. 3.13—is an exact abstraction. The initial under-approximation is $n^{(0)}_{\mathcal{G}_E^{CLIA+}}(X) = \emptyset$ for all Boolean nonterminals in X. The under-approximation of each terminal X at iteration i is the following expression:

$$\begin{aligned} & n^{(i)}_{\mathcal{G}_E^{CLIA+}}(X) \\ = \; & n^{(i-1)}_{\mathcal{G}_E^{CLIA+}}(X) \oplus \\ & \bigoplus_{X \to g(X_1,\ldots,X_n) \in \delta} [\![g]\!]_E^{\#}(n^{(i-1)}_{\mathcal{G}_E^{CLIA+}}(X_1), \ldots, n^{(i-1)}_{\mathcal{G}_E^{CLIA+}}(X_n)) \mid X \in N_{\mathbb{B}}). \end{aligned}$$

Notice that $[\![g]\!]_E^{\#}$ is computable for every operator g (§3.6). This algorithm terminates in at most $2^{|E|}|N_{\mathbb{B}}|$ iterations because the set of Boolean vectors has size at most $2^{|E|}$, and each iteration adds at least one Boolean vector to one of the variables until the least fixed point is reached.

Example 3.30. *Recall Eqn. (3.29) from Ex. 3.29. In the first iteration of the iterative algorithm* $n^{(0)}_{\mathcal{G}_E^{CLIA+}}(X) = \emptyset$. *We compute* $n^{(1)}_{\mathcal{G}_E^{CLIA+}}(\mathrm{BExp})$ *as follows:*

$$\begin{aligned} n^{(1)}_{\mathcal{G}_E^{CLIA+}}(\mathrm{BExp}) \;&=\; \{(t,f)\} \oplus \{(t,t),(f,f)\} \\ &\oplus\; [\![And]\!]^{\#}(n^{(0)}_{\mathcal{G}_E^{CLIA+}}(\mathrm{BExp}), n^{(0)}_{\mathcal{G}_E^{CLIA+}}(\mathrm{BExp})) \end{aligned}$$

and obtain $n^{(1)}_{\mathcal{G}_E^{CLIA+}}(\mathrm{BExp}) = \{(t,f),(t,t),(f,f)\}$. *If we compute* $n^{(2)}_{\mathcal{G}_E^{CLIA+}}(\mathrm{BExp})$ *using the same technique, we reach a fixed point—i.e.,* $n^{(2)}_{\mathcal{G}_E^{CLIA+}}(\mathrm{BExp}) = n^{(1)}_{\mathcal{G}_E^{CLIA+}}(\mathrm{BExp})$. □

Lemma 3.31. *Given a set of equations involving only Boolean-nonterminal variables and representing the abstract semantics of k examples, the iterative algorithm* SolveBool *computes a fixed-point solution in at most $n2^k$ iterations, where n is the number of nonterminal variables.*

Proof. Note that $n^{(i-1)}_{\mathcal{G}^{\text{CLIA}+}_E}(X) \subseteq n^{(i)}_{\mathcal{G}^{\text{CLIA}+}_E}(X)$ for all i and X, and the size of a set of Boolean vector with dimension k is at most 2^k. Then the size of underapproximations is strictly increasing (otherwise the least fixed point is reached) and bounded by $n2^k$, i.e., $\sum_{X\in N} |n^{(i-1)}_{\mathcal{G}^{\text{CLIA}+}_E}(X)| < \sum_{X\in N} |n^{(i)}_{\mathcal{G}^{\text{CLIA}+}_E}(X)| \leqslant n2^k$, for all i. Therefore, the iteration number i can be at most $n2^k$. □

CLIA Equations with Mutual Recursion

We have seen how to compute exact abstractions for grammars without mutual recursion, for both integer (§3.5) and Boolean (§3.6) nonterminals. In this section, we show how to handle grammars that involve IfThenElse symbols, which introduce mutual recursion between Boolean and integer nonterminals. See Eqn. (3.9) in §3.4 for an example of equations that involve mutual recursion. To solve mutually recursive equations, we cannot simply compute the abstraction for one type and use the corresponding values to compute the abstraction for the other type, like we did in §3.6. However, we show that if we repeat such substitutions in an iterative fashion, we obtain an algorithm SolveMutual that computes an exact abstraction for a grammar with mutual recursion.

At the k-th iteration, for every nonterminal X, the algorithm computes an underapproximation $n^k_{\mathcal{G}^{\text{CLIA}+}_E}(X)$ of $n_{\mathcal{G}^{\text{CLIA}+}_E}(X)$. Initially, $n^{-1}_{\mathcal{G}^{\text{CLIA}+}_E}(X) = \mathbf{0}$ for all nonterminals X of type $\mathbb{Z}$. At iteration $k \geqslant 0$ the algorithm does the following:

Step 1 Replace each integer nonterminal Z with the value $n^{k-1}_{\mathcal{G}^{\text{CLIA}+}_E}(Z)$ from iteration $k-1$ and use the technique in §3.6 to compute $n^k_{\mathcal{G}^{\text{CLIA}+}_E}(B)$ for each Boolean nontermi-

nal B. Formally, for each Boolean nonterminal $B \in N_{\mathbb{B}}$ we have the equation:

$$n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(B) = \bigoplus_{B \to g(X_1,\ldots,X_n) \in \delta} [\![g]\!]^{\#}_{E}(n^{(i_1)}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X_1), \ldots, n^{(i_n)}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X_n)) \tag{3.31}$$

where each i_j is equal to k if $X_j \in N_{\mathbb{B}}$ and $k-1$ if $X_j \in N_{\mathbb{Z}}$.

Step 2 Replace each Boolean nonterminal B with the value $n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(B)$ from ***Step 1*** and compute $n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(Z)$ for each integer nonterminal Z (see Eqn. (3.10) in §3.4 for an example).

Formally, for each integer nonterminal $Z \in N_{\mathbb{Z}}$ we have the equation:

$$n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(Z) = \bigoplus_{Z \to g(X_1,\ldots,X_n) \in \delta} [\![g]\!]^{\#}_{E}(n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X_1), \ldots, n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X_n)) \tag{3.32}$$

where for each $X_j \in N_{\mathbb{B}}$, $n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X_j)$ is the value computed in ***Step 1***. The equations obtained at ***Step 2*** only contain integer nonterminals, but they may contain IfThenElse symbols for which the abstract semantics contains the $\text{PROJ}_{\mathbb{SL}}$ operator that is not directly supported by the equation-solving technique presented in §3.5. In the rest of this section, we present a way to transform the given set of equations into a new set of equations that faithfully describes the abstract semantics of IfThenElse symbols, using only $\otimes$ and $\oplus$ operations over semi-linear sets. The resulting equations can be solved using the technique presented in §3.5.

The iterative algorithm SolveMutual is guaranteed to terminate in $|N|2^{|E|}$ iterations.

Lemma 3.32. *Given a set of equations involving both Boolean- and integer-nonterminal variables that represent the abstract semantics of k examples, the iterative algorithm SolveMutual computes a fixed-point solution in at most $n2^k$ iterations, where n is the number of nonterminal variables.*

Proof. In each iteration, at least one of the set $n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(B)$ of Boolean vectors should be different from the set $n^{k-1}_{\mathcal{G}^{\text{CLIA}+}_{E}}(B)$ in the previous iteration. Each set of Boolean

vectors can be only updated at most $2^{|E|}$ times. Therefore there can be at most $|N|2^{|E|}$ iterations. □

$[\![\textbf{IfThenElse}]\!]^{\#}_{E}$ using Semi-Linear-Set Operations

In this section, we show how to solve equations that involve IfThenElse symbols. Recall the definition of the abstract semantics of IfThenElse symbols:

$$[\![\text{IfThenElse}]\!]^{\#}_{E}(\mathit{bset}, sl_1, sl_2) = \bigoplus_{b \in \mathit{bset}} \textsc{proj}_{\mathcal{SL}}(sl_1, b) \otimes \textsc{proj}_{\mathcal{SL}}(sl_2, \neg b)$$

In the rest of this section, we show how equations that involve the semantics of IfThenElse symbols can be rewritten into equations that involve only $\oplus$ and $\otimes$ operations, so that they can be solved using Newton's method. For every possible Boolean vector b, the new set of equations contains a new variable $n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X^b)$, so that the solution to the set of equations for this variable is $\textsc{proj}_{\mathcal{SL}}(n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X), b)$.

Let *eqs* be a set of equations over a set of integer nonterminals N. We write x/y to denote the substitution of every occurrence of x with y. We generate a set of equations $\textsc{RemIf}(\mathit{eqs}) = \mathit{eqs}'$ over the set of variables $N^{\mathbb{B}^d}$ as follows. For every equation $n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X) = \bigoplus_i \alpha_i$ in *eqs* and $b \in \mathbb{B}^d$, there exists an equation $n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X^b) = \bigoplus_i \pi_b(\alpha_i)$ in eqs', where π_b applies the following substitution in this order:

1. For every $X \in N$ and $b' \in \mathbb{B}^d$, π_b applies the substitution

$$\textsc{proj}_{\mathcal{SL}}(n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X), b')/n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X^{b \wedge b'}).$$

2. For every $X \in N$, π_b applies $n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X)/n^{k}_{\mathcal{G}^{\text{CLIA}+}_{E}}(X^b)$.

3. For any semi-linear set sl appearing in *eqs*, π_b applies the substitution $sl/\textsc{proj}_{\mathcal{SL}}(sl, b)$. Because sl is a constant, this substitution yields a constant semi-linear set.

Example 3.33. *Figure 3.3 illustrates how Eqn. (3.10) is rewritten into Eqn. (3.11). We omit equations for variables* $\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{\{f,f\}})$ *and* $\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{\{t,f\}})$ *because they do not contribute to the solving of* $\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{\{t,t\}})$*. After expanding the definition of* $[\![\mathit{IfThenElse}]\!]^{\#}$*, we apply the substitutions to obtain Eqn. (3.11). Substitution 2 is not applied because there are no variables of the form* $\mathfrak{n}^{1}_{2,\mathsf{E}}(X)$ *after applying substitution 1.*

$$
\begin{aligned}
\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}) = {} & [\![\mathrm{IfThenElse}]\!]^{\#}_{\mathsf{E}}(\{(\mathsf{t},\mathsf{f})\},\{(0,0)+\lambda(3,6)\},\\
& \mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}))\oplus\{(0,0)+\lambda(2,4)\}\oplus\{(0,0)+\lambda(3,6)\}\\
& \Downarrow \text{Generate equations for } \mathtt{Start}^{\mathsf{b}}\\
\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{t},\mathsf{t})}) = {} & \pi_{\{\mathsf{t},\mathsf{t}\}}\big([\![\mathrm{IfThenElse}]\!]^{\#}_{\mathsf{E}}(\{(\mathsf{t},\mathsf{f})\},\{(0,0)+\lambda(3,6)\},\\
& \mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}))\big)\oplus\pi_{\{\mathsf{t},\mathsf{t}\}}\big(\{(0,0)+\lambda(2,4)\}\big)\oplus\pi_{\{\mathsf{t},\mathsf{t}\}}\big(\{(0,0)+\lambda(3,6)\}\big)\\
\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{f},\mathsf{t})}) = {} & \pi_{\{\mathsf{f},\mathsf{t}\}}\big([\![\mathrm{IfThenElse}]\!]^{\#}_{\mathsf{E}}(\{(\mathsf{t},\mathsf{f})\},\{(0,0)+\lambda(3,6)\},\\
& \mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}))\big)\oplus\pi_{\{\mathsf{f},\mathsf{t}\}}\big(\{(0,0)+\lambda(2,4)\}\big)\oplus\pi_{\{\mathsf{f},\mathsf{t}\}}\big(\{(0,0)+\lambda(3,6)\}\big)\\
& \Downarrow \text{Expand definition of } [\![\mathrm{IfThenElse}]\!]^{\#}\\
\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{t},\mathsf{t})}) = {} & \pi_{\{\mathsf{t},\mathsf{t}\}}\big(\textsc{proj}_{\mathbb{SL}}(\{(0,0)+\lambda(3,6)\},\{\mathsf{f},\mathsf{t}\})\big)\\
& \otimes\ \pi_{\{\mathsf{t},\mathsf{t}\}}\big(\textsc{proj}_{\mathbb{SL}}(\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}),(\mathsf{f},\mathsf{t}))\big)\\
& \oplus\ \pi_{\{\mathsf{t},\mathsf{t}\}}\big(\{(0,0)+\lambda(2,4)\}\big)\oplus\pi_{\{\mathsf{t},\mathsf{t}\}}\big(\{(0,0)+\lambda(3,6)\}\big)\\
\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{f},\mathsf{t})}) = {} & \pi_{\{\mathsf{f},\mathsf{t}\}}\big(\textsc{proj}_{\mathbb{SL}}(\{(0,0)+\lambda(3,6)\},\{\mathsf{f},\mathsf{t}\})\big)\\
& \otimes\ \pi_{\{\mathsf{f},\mathsf{t}\}}\big(\textsc{proj}_{\mathbb{SL}}(\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}),(\mathsf{f},\mathsf{t}))\big)\\
& \oplus\ \pi_{\{\mathsf{f},\mathsf{t}\}}\big(\{(0,0)+\lambda(2,4)\}\big)\oplus\pi_{\{\mathsf{f},\mathsf{t}\}}\big(\{(0,0)+\lambda(3,6)\}\big)\\
& \Big\Downarrow \begin{array}{l}\text{Apply } \textsc{proj}_{\mathbb{SL}} \text{ to constants}\\ \text{Apply substitution 1}\end{array}\\
\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{t},\mathsf{t})}) = {} & \pi_{\{\mathsf{t},\mathsf{t}\}}\big(\{(0,0)+\lambda(3,0)\}\big)\otimes\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{t},\mathsf{t})\wedge(\mathsf{f},\mathsf{t})})\\
& \oplus\ \pi_{\{\mathsf{t},\mathsf{t}\}}\big(\{(0,0)+\lambda(2,4)\}\big)\oplus\pi_{\{\mathsf{t},\mathsf{t}\}}\big(\{(0,0)+\lambda(3,6)\}\big)\\
\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{f},\mathsf{t})}) = {} & \pi_{\{\mathsf{f},\mathsf{t}\}}\big(\{(0,0)+\lambda(3,0)\}\big)\otimes\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{f},\mathsf{t})\wedge(\mathsf{f},\mathsf{t})})\\
& \oplus\ \pi_{\{\mathsf{f},\mathsf{t}\}}\big(\{(0,0)+\lambda(2,4)\}\big)\oplus\pi_{\{\mathsf{f},\mathsf{t}\}}\big(\{(0,0)+\lambda(3,6)\}\big)\\
& \Downarrow \text{Apply substitution 3}\\
\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{t},\mathsf{t})}) = {} & \{(0,0)+\lambda(3,0)\}\otimes\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{f},\mathsf{t})})\oplus\{(0,0)+\lambda(2,4)\}\oplus\{(0,0)+\lambda(3,6)\}\\
\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{f},\mathsf{t})}) = {} & \{(0,0)+\lambda(0,0)\}\otimes\mathfrak{n}^{1}_{2,\mathsf{E}}(\mathit{Start}^{(\mathsf{f},\mathsf{t})})\oplus\{(0,0)+\lambda(0,4)\}\oplus\{(0,0)+\lambda(0,6)\}
\end{aligned}
$$

Figure 3.3: Rewriting Eqn. (3.10) into Eqn. (3.11).

Lemma 3.34. *Given a set of equations eqs involving only variables* $V := \{n_{\mathcal{G}_E^{CLIA+}}(X)\}_{X \in N}$, *the set of equations* RemIf(*eqs*) *has at most* $|V|2^{|E|}$ *variables, and an assignment* σ' *is a solution of* RemIf(*eqs*) *iff there exists a solution* σ *of eqs such that* $\sigma(n_{\mathcal{G}_E^{CLIA+}}(X)) = \sigma'(n_{\mathcal{G}_E^{CLIA+}}(X^{\vec{t}}))$ *for all* $X \in N$.

Proof. The variables in RemIf(*eqs*) are of form $n_{\mathcal{G}_E^{CLIA+}}(X^b)$ for $X \in V$ and $b \in 2^{|E|}$. Therefore there are at most $|N|2^{|E|}$ variables in RemIf(*eqs*).

To show that every solution to *eqs* is also a solution to $V^{\vec{t}} := \{n_{\mathcal{G}_E^{CLIA+}}(X^{\vec{t}})\}$ in *eqs*′, it is sufficient to prove that for all $b \in \mathbb{B}^d$, if an assignment $\sigma : V \to \mathcal{SL}$ is a solution to *eqs*, then the assignment $\sigma'(n_{\mathcal{G}_E^{CLIA+}}(X^b)) := \text{PROJ}_{\mathcal{SL}}(\sigma(n_{\mathcal{G}_E^{CLIA+}}(X)), b)$ is a solution to *eqs*′.

Actually, equations in *eqs* are of the form $X = \bigoplus_i \alpha_i$ (Eqn. (3.32)). Therefore, all we need to show is that $\text{PROJ}_{\mathcal{SL}}(\alpha_i[\sigma], b) = \pi_b(\alpha_i)[\sigma']$ for all α_i and $b \in \mathbb{B}^d$, where $[\sigma] := [\text{for each } x \in V.x/\sigma(x)]$. Note that α must be one of the following form:

- if $\alpha = \text{PROJ}_{\mathcal{SL}}(n_{\mathcal{G}_E^{CLIA+}}(X_1), b_1) \otimes \text{PROJ}_{\mathcal{SL}}(n_{\mathcal{G}_E^{CLIA+}}(X_2), b_2)$, we have

$$\begin{aligned}\text{PROJ}_{\mathcal{SL}}(\alpha[\sigma], b) &= \text{PROJ}_{\mathcal{SL}}(\sigma(n_{\mathcal{G}_E^{CLIA+}}(X_1)), b_1 \wedge b) \otimes \text{PROJ}_{\mathcal{SL}}(\sigma(n_{\mathcal{G}_E^{CLIA+}}(X_2)), b_2 \wedge b)\\ &= \sigma'(n_{\mathcal{G}_E^{CLIA+}}(X_1^{b_1 \wedge b})) \otimes \sigma'(n_{\mathcal{G}_E^{CLIA+}}(X_2^{b_2 \wedge b}))\\ &= \pi_b(\alpha_i)[\sigma']\end{aligned}$$

- if $\alpha = n_{\mathcal{G}_E^{CLIA+}}(X_1) \otimes n_{\mathcal{G}_E^{CLIA+}}(X_2)$, we have

$$\begin{aligned}\pi_b(\alpha_i)[\sigma'] &= \left(n_{\mathcal{G}_E^{CLIA+}}(X_1^b) \otimes n_{\mathcal{G}_E^{CLIA+}}(X_2^b)\right)[\sigma']\\ &= \text{PROJ}_{\mathcal{SL}}(\sigma(n_{\mathcal{G}_E^{CLIA+}}(X_1)), b) \otimes \text{PROJ}_{\mathcal{SL}}(\sigma(n_{\mathcal{G}_E^{CLIA+}}(X_2)), b)\\ &= \text{PROJ}_{\mathcal{SL}}(\alpha[\sigma], b),\end{aligned}$$

  since $\text{PROJ}_{\mathcal{SL}}$ is distributive over $\otimes$.

- if $\alpha = sl$, it is obvious that $\text{PROJ}_{\mathcal{SL}}(\alpha[\sigma], b) = \pi_b(\alpha_i)[\sigma']$.

- if $\alpha = n_{\mathcal{G}_E^{\text{CLIA}+}}(X_1)$, we have

$$\begin{aligned} \textsc{proj}_{\mathcal{SL}}(\alpha[\sigma], b) &= \textsc{proj}_{\mathcal{SL}}(\sigma(n_{\mathcal{G}_E^{\text{CLIA}+}}(X_1)), b) \\ &= \sigma'(n_{\mathcal{G}_E^{\text{CLIA}+}}(X_1^b)) = \pi_b(\alpha_i)[\sigma']. \end{aligned}$$

Therefore any solution to *eqs* is a solution to $V^{\vec{t}}$ in *eqs*′.

For the other direction, we need to show that assume σ' is a solution to *eqs*′, $\sigma(\cdot) := \sigma'(\cdot, \vec{tr})$ is a solution to *eqs*. The argument for this case is similar to the previous one. □

Checking Unrealizability

Using the symbolic-concretization technique described in §3.5 , and the complexities described throughout this section, we obtain the following decidability theorem.

Theorem 3.35. *Given a CLIA SyGuS problem* sy *and a finite set of examples* E*, it is decidable whether the SyGuS problem* sy^E *is (un)realizable.*

Proof. We have shown that $n_{\mathcal{G}_E^{\text{CLIA}+}{}_E}$ is an exact abstraction for CLIA grammars (Lemma 3.28) and the domain of semi-linear sets supports symbolic concretization. Besides, we have shown a sound and complete algorithm SolveMutual to solve $n_{\mathcal{G}_E^{\text{CLIA}+}{}_E}$. According to the Thm. 3.15, GFA is sound and complete for proving unrealizability of CLIA SyGuS problems for finitely example, and hence decidable. □

3.7 Implementation

Tool Nope

We implemented the technique of unrealizability (§3.3) as verification into a tool Nope. Nope is a tool that can return two-sided answers to unrealizability problems of the form $sy = (\psi, G)$. When it returns ***unrealizable***, no expression-tree in $L(G)$

satisfies ψ; when it returns ***realizable***, some $e \in L(G)$ satisfies ψ; Nope can also time out. Nope incorporates several existing pieces of software.

1. The (un)reachability verifier SeaHorn is applied to the reachability problems of the form $re_{enc(sy,E)}$ created during successive CEGIS rounds.

2. The SMT solver Z3 is used to check whether a generated expression-tree e satisfies ψ. If it does, Nope returns ***realizable*** (along with e); if it does not, Nope creates a new input example to add to E.

It is important to observe that SeaHorn, like most reachability verifiers, is only sound for ***un***satisfiability—i.e., if SeaHorn returns ***unsatisfiable***, the reachability problem is indeed unsatisfiable. Fortunately, SeaHorn's one-sided answers are in the correct direction for our application: to prove unrealizability, Nope only requires the reachability verifier to be sound for unsatisfiability.

There is one aspect of Nope that differs from the technique that has been presented earlier in the chapter. While SeaHorn is sound for ***un***reachability, it is not sound for reachability—i.e., it cannot soundly prove whether a synthesis problem is realizable. To address this problem, to check whether a given SyGuS problem sy^E is realizable on the finite set of examples E, Nope also calls the SyGuS solver ESolver [AFSSL16b] to synthesize an expression-tree e that satisfies sy^E.[13]

In practice, for every intermediate problem sy^E generated by the CEGIS algorithm, Nope runs the ESolver on sy^E and SeaHorn on $re_{enc(sy,E)}$ in *parallel*. If ESolver returns a solution e, SeaHorn is interrupted, and Z3 is used to check whether e satisfies ψ. Depending on the outcome, Nope either returns ***realizable*** or obtains an additional input example to add to E. If SeaHorn returns ***unsatisfiable***, Nope returns ***unrealizable***.

Modulo bugs in its constituent components, Nope is sound for both realizability and unrealizability, but because of Lemma 3.5 and the incompleteness of SeaHorn, Nope is not complete for unrealizability.

[13] We chose ESolver because on the benchmarks we considered, ESolver outperformed other SyGuS solvers (e.g., CVC4 [BCD+11]).

Tool NAY

We implemented the technique of unrealizability as grammar flow analysis (§3.4–§3.6) into a tool NAY that can return two-sided answers to unrealizability problems of the form $sy = (\psi, G)$. When it returns ***unrealizable***, no term in $L(G)$ satisfies ψ; when it returns ***realizable***, some $e \in L(G)$ satisfies ψ; NAY can also time out. NAY consists of three components: 1) a verifier (the SMT solver CVC4 [BCD+11]), which verifies the correctness of candidate solutions and produces counterexamples, 2) a synthesizer (ESolver—the enumerative solver introduced in [AFSSL16b]), which synthesizes solutions from examples, and 3) an unrealizability verifier, which proves whether the problem is unrealizable on the current set of examples.

Algorithm 1 shows NAY's CEGIS loop. Given a SYGUS problem $sy = (\psi, G)$, NAY first initialize E with a random input example with values in the range $[-50, 50]$(line (3.7)), and then, in parallel, ❶ calls ESolver to find a solution of sy^E (line (3.7)), and ❷ uses grammar flow analysis (Algorithm 2) to decide whether $sy^{E \cup E_r}$ is unrealizable (line (3.7)), where E_r is a set of randomly generated temporary examples. Randomly generated examples are used when the problem is proven to be realizable by GFA, but we do not have a candidate solution e^*—ESolver did not return yet—that can be used to issue an SMT query to possibly obtain a counterexample. During each CEGIS iteration, the following three events can happen: 1) If GFA returns unrealizable, NAY terminates and outputs unrealizable (line (3.7)). 2) If GFA returns realizable, NAY adds a temporary random example to E_r (line (3.7)), and reruns GFA with $E \cup E_r$. 3) If ESolver returns a candidate solution e^*, the problem sy^E is realizable. (ESolver never uses the temporary random examples.) Therefore, NAY kills the GFA process and then issues an SMT query to check if e^* is a solution to the SYGUS problem sy (line (3.7)): if not, NAY adds a counterexample to E (line (3.7)) and triggers the next CEGIS iteration, otherwise, NAY return e^* as a solution to the given SYGUS problem sy (line (3.7)).

NAY currently has two modes: NAY-HORN and NAY-SL.

NAY-HORN implements the constrained-Horn-clauses technique for solving equations presented in §3.4, and uses Z3's Horn-clause solver, Spacer [DMB08], to solve

the Horn clauses.

Nay-SL implements the decision procedures presented in §3.5 and §3.6 for solving LIA and CLIA problems. Nay-SL also implements two optimizations: (i) Nay-SL eagerly removes a linear set from a semi-linear set whenever it is trivially subsumed by another linear set; and (ii) Nay-SL uses the optimization presented in the following paragraph.

Algorithm 1 CEGIS with random examples

```
Function: Nay(G, ψ)
Input: Grammar G, specification ψ
i ← Random(−50, 50)    Set of examples E ← {i}  while True do
    do in parallel
        ❶ {e* ←ESolver(G, ψ, E)
        kill ❷
        if ∃i_cex.¬ψ(⟦e*⟧, i_cex) then
            E ← E ∪ {i_cex}
            continue
        end
            else
            return e* }
        end
        ❷ { E_r ← ∅
        while True do
            result ←CheckUnrealizable(G, ψ, E ∪ E_r)
            if result =Unrealizable then
                kill ❶
                return Unrealizable
            end
            i ← Random(−50, 50)
            E_r ← E_r ∪ {i}
            continue }
        end
    end
end
```

Solving GFA Equations via Stratification. The $n_{\mathcal{G}}$ equations (Eqn. (3.13)) that arise in a GFA problem are amenable to the standard optimization technique of identifying "strata" of dependences among nonterminals, and solving the equations

by finding values for nonterminals of lower "strata" first, working up to higher strata in an order that respects dependences among the equations.

This idea can be formalized in terms of the strongly connected components (SCCs) of a dependence graph, defined as follows: the nodes are the nonterminals of G; the edges represent the dependence of a left-hand-side nonterminal on a right-hand-side nonterminal. For instance, if G has the productions $X_0 \to g(X_1, X_2) \mid h(X_2, X_3)$, then the dependence graph has three edges into node X_0: $X_1 \to X_0$, $X_2 \to X_0$, and $X_3 \to X_0$. There are three steps to finding an order in which to solve the equations:

- Find the SCCs of the dependence graph.
- Collapse each SCC into a single node, to form a directed acyclic graph (DAG).
- Find a topological order of the DAG.

The set of nonterminals associated with a given node of the DAG corresponds to one of the strata referred to earlier. The equation solver can work through the strata in any topological order of the DAG.

3.8 Evaluation

In this section, we evaluate the effeteness and perforamnce of Nope and Nay on unrealizable SyGuS problems.

Benchmarks. We perform our evaluation on 132 variants of the 60 LIA benchmarks from the LIA SyGuS competition track [AFSSL16b]. We do not consider the other SyGuS benchmark track, Bit-Vectors, because the SeaHorn verifier is unsound for most bit-vector operations–e.g., bit-shifting. We used three suites of benchmarks. LimitedIf (resp. LimitedPlus) contains 57 (resp. 30) benchmarks in which the grammar bounds the number of times an IfThenElse (resp. Plus) operator can appear in an expression-tree to be 1 less than the number required to solve the original synthesis problem. We used the tool Quasi to automatically generate the restricted grammars. LimitedConst contains 45 benchmarks in which the grammar

allows the program to contain only constants that are coprime to any constants that may appear in a valid solution—e.g., the solution requires using odd numbers, but the grammar only contains the constant 2. The numbers of benchmarks in the three suites differ because for certain benchmarks it did not make sense to create a limited variant—e.g., if the smallest program consistent with the specification contains no IfThenElse operators, no variant is created for the LimitedIf benchmark. In all our benchmarks, the grammars describing the search space contain infinitely many terms.

Our experiments were performed on an Intel Core i7 4.00GHz CPU, with 32GB of RAM, running Lubuntu 18.10 via VirtualBox. We used version 4.8 of Z3, commit 97f2334 of SeaHorn, and commit d37c50e of ESolver. The timeout for each individual SeaHorn/ESolver call is set at 10 minutes.

Effectiveness of Nope

The complete results of our evaluation are shown in Tables 3.2 and 3.3. For brevity, in Table 3.2 we omit consecutive benchmarks on which Nope times out—e.g., the "..." between benchmarks max4 and max15 represents 10 benchmarks from max5 to max14 for which Nope times out.

The tables present the **number of nonterminals** and the **number of productions** in the grammar of each benchmark, the **number of examples** used to prove unrealizability, the total time taken by Nope, and the time taken by SeaHorn for the last (un)reachability problem. For benchmarks on which Nope times out, the value given for "**number of examples**" is the number of examples generated by the CEGIS loop when Nope times out.

Our experiments were designed to answer the questions posed below.

EQ 1. Can Nope prove unrealizability for variants of real SyGuS benchmarks, and how long does it take to do so?

Finding: Nope can prove unrealizability for 59/132 *benchmarks.* For the 59 benchmarks solved by Nope, the average time taken is 15.59. The time taken to perform

the last iteration of the algorithm—i.e., the time taken by SeaHorn to return **unsatisfiable**—accounts for 87% of the total running time.

Nope can solve all of the LimitedIf benchmarks for which the grammar allows at most one IfThenElse operator. Allowing more IfThenElse operators in the grammar leads to larger programs and larger sets of examples, and consequently the resulting reachability problems are harder to solve for SeaHorn.

For a similar reason, Nope can solve only one of the LimitedPlus benchmarks. All other LimitedPlus benchmarks allow 5 or more Plus statements, resulting in grammars that have at least 130 productions.

Nope can solve all LimitedConst benchmarks because these require few examples and result in small encoded programs.

EQ 2. How many examples does Nope use to prove unrealizability and how does the number of examples affect the performance of Nope?

Note that Z3 can produce different models for the same query, and thus different runs of NOPE can produce different sequences of example. Hence, there is no guarantee that NOPE finds a good sequence of examples that prove unrealizability. One measure of success is whether Nope is generally able to find a small number of examples, when it succeeds in proving unrealizability.

Finding: Nope used 1 to 9 examples to prove unrealizability for the benchmarks on which it terminated. Problems requiring large numbers of examples could not be solved because either ESolver or SeaHorn timeouts—e.g., on the problem max4, Nope gets to the point where the CEGIS loop has generated 17 examples, at which point ESolver exceeds the timeout threshold.

Finding: The number of examples required to prove unrealizability depends mainly on the arity of the synthesized function and the complexity of the grammar. The number of examples seems to grow quadratically with the number of bounded operators allowed in the grammar. In particular, problems in which the grammar allows zero IfThenElse operators require 2–4 examples, while problems in which the grammar allows one IfThenElse operator require 7–9 examples.

Figure 3.4 plots the running time of Nope against the number of examples generated by the CEGIS algorithm. *Finding: The solving time appears to grow exponentially*

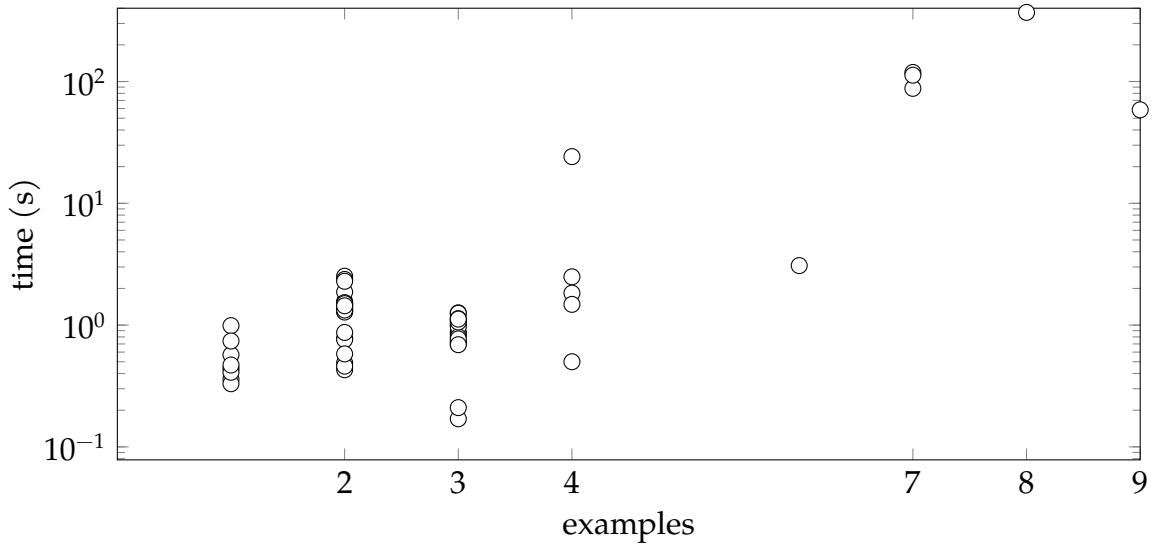


Figure 3.4: Time vs examples.

with the number of examples required to prove unrealizability.

Effectiveness of Nay

EQ 3. How effective is Nay at proving unrealizability?

We compare Nay-SL and Nay-Horn against Nope. For each benchmark, we run each tool 5 times on different random seeds, therefore generating different random sets of examples, and report whether a tool successfully terminated on at least one run. This process guarantees that all tools are evaluated on the same final example set that causes a problem to be unrealizable. Table 3.1 shows the results for the LimitedPlus and LimitedIf benchmarks that at least one of the three tools could solve. Because both tools use a CEGIS loop to produce input examples, only the last iteration of CEGIS is unrealizable. For Nay-SL and Nope, that iteration is the one that dominates the runtime. On average, it accounts for 60.4% of the running time for Nay-SL and 90.3% for Nope, but only 8.3% for Nay-Horn. (For Nay-Horn, counterexample generation is the most costly step.)

Table 3.1: Performance of Nay and Nope for LimitedIf and LimitedPlus benchmarks.[14] The table shows the number of nonterminals (|N|), productions (|δ|), and variables (|V|) in the problem grammar; the number of examples required to prove unrealizability (|E|); and the average running time of Nay-SL, Nay-Horn, and Nope. ✗ denotes a timeout.

	Problem	Grammar			|E|	time (s)		
		|N|	|δ|	|V|		Nay-SL	Nay-Horn	Nope
LimitedPlus	guard1	7	24	3	2	0.24	✗	✗
	guard2	9	34	3	3	12.86	✗	✗
	guard3	11	41	3	1	0.07	✗	✗
	guard4*	11	72	3	3.5	147.50	✗	✗
	plane1	2	5	2	1	0.07	0.55	0.69
	plane2	17	60	2	1.6	0.90	✗	✗
	plane3	29	122	2	1.5	15.73	✗	✗
	ite1*	7	2	3	2	1.05	✗	✗
	ite2*	9	34	3	4	294.88	✗	✗
	sum_2_5	11	40	2	4	15.48	✗	✗
	search_2	5	16	3	3	1.21	✗	✗
	search_3	7	25	4	4	2.65	✗	✗
LimitedIf	max2	1	5	2	4	0.13	1.13	1.48
	max3	3	15	3	-	✗	9.67	58.57
	sum_2_5	1	5	2	3	0.17	0.61	0.69
	sum_2_15	1	5	2	3	0.17	0.56	0.87
	sum_3_5	3	15	3	-	✗	17.85	101.44
	sum_3_15	3	15	3	-	✗	16.65	134.87
	search_2	3	15	3	-	✗	25.85	112.78
	example1	3	10	2	3	0.14	0.73	1.12
	guard1	1	6	2	4	0.13	0.44	0.43
	guard2	1	6	2	4	0.22	0.33	0.49
	guard3	1	6	2	4	0.16	0.27	0.46
	guard4	1	6	2	4	0.11	0.72	0.58
	ite1	3	15	3	-	✗	2.68	369.57

Findings. Nay-SL solved 70/132 benchmarks, with an average running time of 1.97s.[15] Nay-Horn and Nope solved identical sets of 59/132 benchmarks, with an average running time of 0.63s and 15.59s, respectively. All tools can solve all the LimitedConst benchmarks with similar performance. These benchmarks are easier than the other ones.

Nay-SL can solve 11 LimitedPlus benchmarks that Nope cannot solve. These

[15]Most of the benchmarks for which Nay-SL times out are actually crashes caused by a memory leak in CVC4. We have reported the bug.

benchmarks involve large grammars, a known weakness of Nope. In particular, NaySL can handle grammars with up to 29 nonterminals while Nope can only handle grammars with up to 3 nonterminals. For 8 benchmarks, Nay-SL only terminated for some of the random runs (certain random seeds triggered more CEGIS iterations, making the final problem harder for Nay to solve).

Nope solved 5 LimitedIf benchmarks that Nay-SL cannot solve. Nope solves these benchmarks using between 7 and 9 examples in the CEGIS loop. Because the size of the semi-linear sets computed by Nay-SL depends heavily on the number of examples, Nay-SL only solves benchmarks that require at most 4 examples. §3.8 analyzes the effect of the number of examples on Nay-SL's performance. When Nay-SL terminated, it took 1 to 15 iterations (avg. 6.6) to find a fixed point for IfThenElse guards, and the final abstract domain of each guard contained 2 to 16 Boolean vectors (avg. 5.9). On average, the running time for computing semi-linear sets is 70.6% of the total running time. On the benchmarks that all tools solved, all tools terminated in less than 2s.

Nay-Horn and Nope solved exactly the same set of benchmarks. This outcome is not surprising because Nope uses SeaHorn, a verification solver based on Horn clauses that builds on Spacer, which is the constrained-Horn-clause solver used by Nay-Horn. Nay-Horn directly encodes the equation-solving problem, while Nope reduces the unrealizability problem to a verification problem that is then translated into a potentially complex constrained-Horn-clause problem. For this reason, Nay-Horn is on average 19 times faster than Nope. On benchmarks for which Nope took more than 2 seconds, Nay-Horn is 82x faster than Nope (computed as the geometric mean).

The reason we use random examples in Algorithm 1 is that there is a trade-off between the size of solutions and the number of examples when we are proving the realizability of SyGuS-with-examples problems. On the one hand, ESolver is not affected by the number of examples, and can efficiently synthesize a solution when a small solution exists. On the other hand the time required to prove realizability by Nay-SL only depends on the size of grammars and the number of examples but not on the size of solutions. For the realizable SyGuS-with-examples problems

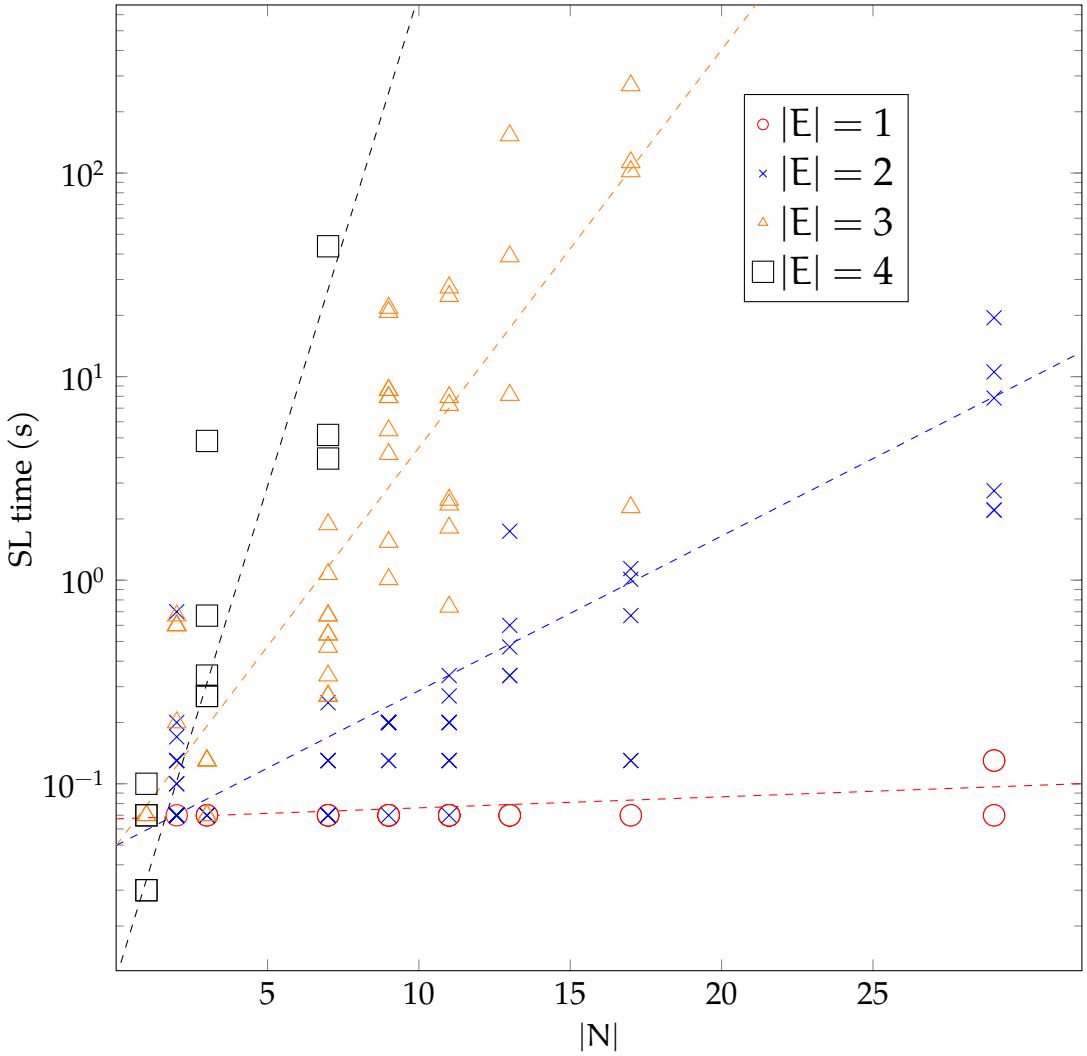


Figure 3.5: Time to compute semi-linear set vs. |N|.

produced during the CEGIS loop of our experiments, ESolver terminates on average in 1.9 seconds when there exists a solution with size no more than 10, but terminates on average in 54.5 seconds when there exists a solution with size greater than 10 (the largest solution has size 24). For the same problems, Nay-SL could not prove realizability for problems with more than 5 examples, but it did prove realizability for 7 problems on which ESolver failed. On the problems both ESolver and Nay-SL

solved, ESolver is 87% faster than Nay-SL calculated as a geometric mean.

To answer **EQ 1**: if both Nay techniques are considered together, *Nay solved 11 benchmarks that Nope did not solve, and was faster on the benchmarks that both tools solved.*

The Cost of Proving Unrealizability

EQ 4. How does the size of the grammar and the number of examples affect the performance of different solvers?

Finding. First, consider Nay-SL: when we fix the number of examples (different marks in Fig. 3.5), the time taken to compute the semi-linear set grows roughly exponentially. Also, the time grows roughly exponentially with respect to $2^{|E|}$.

Nay-Horn and Nope (shown in Fig. 3.6 and Fig. 3.7, respectively) can only solve benchmarks involving up to 3 nonterminals. When we fix the number of nonterminals, the running time of these two tools grows roughly exponentially with respect to the number of examples.

To answer **EQ 2**: the running time of Nay-SL grows exponentially with respect to $|N|2^{|E|}$, and the running time of Nay-Horn and Nope grows exponentially with respect to $|E|$.

Effectiveness of Grammar Stratification

EQ 5. Is the stratification optimization from §3.7 effective?

Finding. Using stratification, Nay-SL can compute the semi-linear sets for 9 benchmarks for which Nay-SL times out without the optimization. On benchmarks that take more than 1s to solve, the optimization results on average in a 3.1x speedup. To answer **EQ 3**: the grammar-stratification optimization is highly effective.

3.9 Summary

In this chapter, we presented

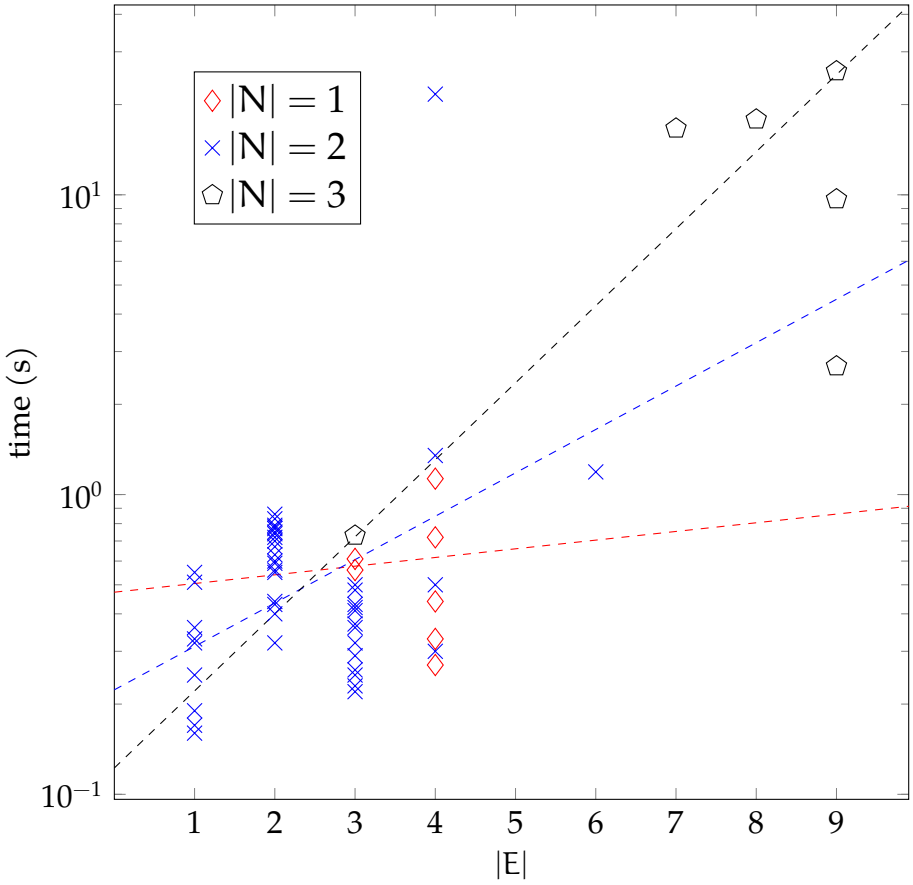


Figure 3.6: Running time of Nay-Horn vs. number of examples.

1. Nope, a general algorithm of proving unrealizability of SyGuS problems by reducing the problems to verification problems;

2. Nay, a algorithm of proving unrealizability of SyGuS problems in CLIA theory, which is based on the idea of grammar flow analysis and decidable.

The two algorithm of proving unrealizability allow us to strengthen the algorithm of QuaSi that solves quantitative syntactic objectives in the sense that they provide procedures to prove a solution is optimal instead of exhausting the search space.

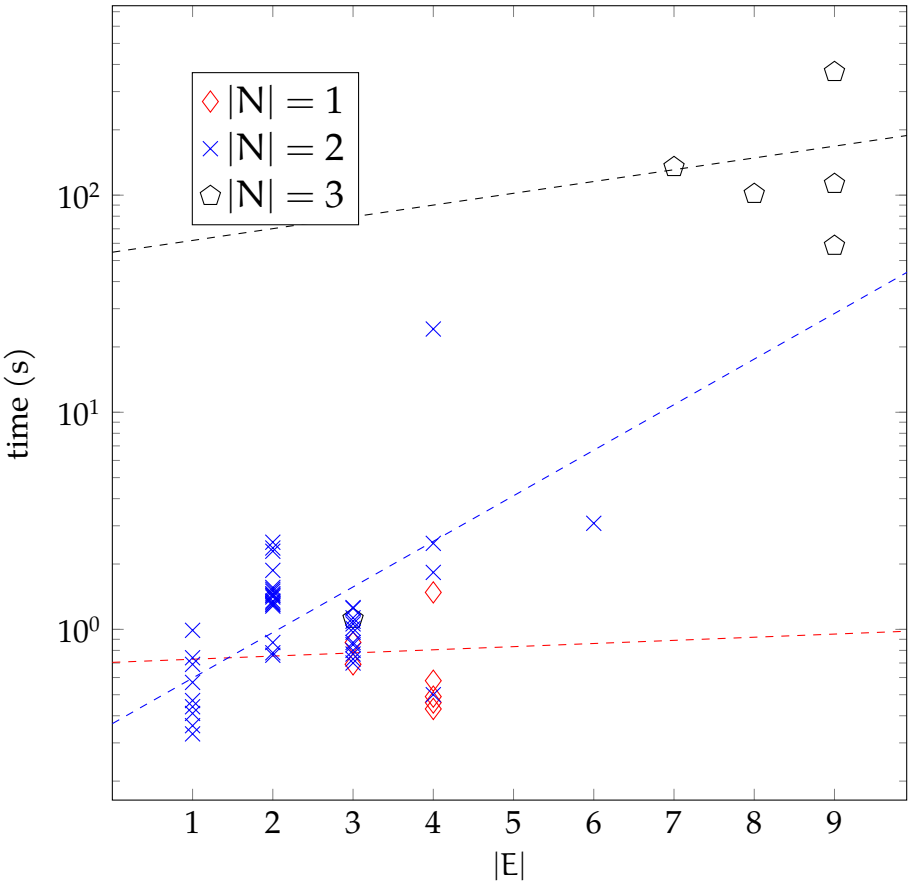


Figure 3.7: Running time of NOPE vs. number of examples.

Table 3.2: Performance of Nope on LimitedIf and LimitedPlus benchmarks. ✗ denotes a timeout.

	Problem	number of nonterminals	number of productions	number of examples	total time (s)	SeaHorn time (s)
LimitedIf	max2	1	5	4	1.48	0.53
	max3	3	15	9	58.57	50.21
	max4	5	34	17	✗	✗
	...				✗	✗
	max15	27	348	1	✗	✗
	array_sum_2_5	1	5	3	0.69	0.17
	array_sum_2_15	1	5	3	0.87	0.21
	array_sum_3_5	3	15	7	101.44	87.92
	array_sum_3_15	3	15	7	134.87	118.77
	array_sum_4_5	5	34	14	✗	✗
	array_sum_4_15	5	34	16	✗	✗
	...				✗	✗
	array_sum_10_5	19	149	1	✗	✗
	array_sum_10_15	19	149	1	✗	✗
	array_search_2	3	15	7	112.78	87.32
	array_search_3	5	34	17	✗	✗
	...				✗	✗
	array_search_15	27	348	1	✗	✗
	mpg_example1	1	7	3	1.12	0.38
	mpg_example2	9	60	17	✗	✗
	mpg_example3	5	34	12	✗	✗
	mpg_example4	5	34	19	✗	✗
	mpg_example5	9	60	11	✗	✗
	mpg_guard1	1	6	2	0.43	0.18
	mpg_guard2	1	6	2	0.49	0.19
	mpg_guard3	1	6	2	0.46	0.17
	mpg_guard4	1	6	2	0.58	0.18
	mpg_ite1	3	15	8	369.57	361.21
	mpg_ite2	5	29	11	✗	✗
LimitedPlus	array_sum_2_5	19	155	1	✗	✗
	...				✗	✗
	array_sum_10_5	461	51960	1	✗	✗
	array_sum_2_15	59	702	1	✗	✗
	...				✗	✗
	array_sum_8_15	641	64816	1	✗	✗
	mpg_example1	59	815	1	✗	✗
	mpg_example2	21	178	1	✗	✗
	mpg_example3	143	4186	1	✗	✗
	mpg_example4	443	36745	1	✗	✗
	mpg_example5	351	24872	1	✗	✗
	mpg_guard1	7	130	1	✗	✗
	mpg_guard2	9	237	1	✗	✗
	mpg_guard3	11	382	1	✗	✗
	mpg_guard4	13	580	1	✗	✗
	mpg_ite1	9	237	1	✗	✗
	mpg_ite2	13	580	1	✗	✗
	mpg_plane1	2	5	3	0.69	0.13
	mpg_plane2	17	139	1	✗	✗
	mpg_plane3	29	309	1	✗	✗

Table 3.3: Performance of Nope on LimitedConst benchmarks. ✗ denotes a timeout.

	Problem	number of nonterminals	number of productions	number of examples	total time (s)	SeaHorn time (s)
LimitedConst	array_search_2	2	9	2	0.78	0.32
	array_search_3	2	10	3	1.26	0.43
	array_search_4	2	11	3	1.25	0.22
	array_search_5	2	12	3	1.01	0.50
	array_search_6	2	13	3	0.87	0.41
	array_search_7	2	14	3	0.85	0.26
	array_search_8	2	15	3	0.97	0.36
	array_search_9	2	16	3	0.70	0.48
	array_search_10	2	17	3	0.80	0.37
	array_search_11	2	18	3	1.09	0.32
	array_search_12	2	19	3	1.13	0.25
	array_search_13	2	20	3	0.73	0.29
	array_search_14	2	21	3	0.77	0.42
	array_search_15	2	22	3	1.06	0.23
	array_sum_2_5	2	8	2	1.30	0.77
	array_sum_2_15	2	8	2	1.46	0.83
	array_sum_3_5	2	9	2	1.31	0.86
	array_sum_3_15	2	9	2	1.28	0.75
	array_sum_4_5	2	10	2	2.52	0.60
	array_sum_4_15	2	10	2	1.35	0.56
	array_sum_5_5	2	11	2	1.41	0.72
	array_sum_5_15	2	11	2	1.43	0.44
	array_sum_6_5	2	12	2	2.37	0.55
	array_sum_6_15	2	12	2	1.56	0.70
	array_sum_7_5	2	13	2	0.76	0.59
	array_sum_7_15	2	13	2	1.87	0.78
	array_sum_8_5	2	14	2	1.33	0.63
	array_sum_8_15	2	14	2	1.53	0.67
	array_sum_9_5	2	15	2	1.50	0.40
	array_sum_9_15	2	15	2	1.44	0.79
	array_sum_10_5	2	16	2	2.29	0.74
	array_sum_10_15	2	16	2	0.87	0.43
	mpg_example1	2	8	1	0.36	0.17
	mpg_example2	2	9	4	0.50	0.30
	mpg_example3	2	9	1	0.57	0.33
	mpg_example4	2	10	1	0.44	0.16
	mpg_example5	2	8	1	0.99	0.36
	mpg_guard1	2	9	6	3.08	1.19
	mpg_guard2	2	9	4	2.49	1.35
	mpg_guard3	2	9	4	1.83	0.50
	mpg_guard4	2	9	4	24.18	21.67
	mpg_ite1	2	9	1	0.33	0.19
	mpg_ite2	2	9	1	0.41	0.25
	mpg_plane2	2	9	1	0.47	0.32
	mpg_plane3	2	9	1	0.74	0.51

Part II

Quantitative Semantic Objectives in Synthesis

Chapter 4

Synthesis with Asymptotic Resource Bounds

4.1 Introduction

We have looked at quantitative syntactic objectives in the previous part. Now let us move to another kind of quantitative objective in program synthesis: quantitative *semantic* objectives—e.g., synthesizing a program that has a certain asymptotic complexity.

Recently, Knoth et al. [KWPH19] studied the problem of resource-guided program synthesis, where the goal is to synthesize programs with limited resource usage. Their approach, which combines refinement-type-directed synthesis [PKSL16] and automatic amortized resource analysis (AARA) [HAH11], is restricted to *concrete* resource bounds, where the user must specify the *exact* resource usage of the synthesized program as a *linear* expression. This limitation has two drawbacks: (i) the user must have insights about the coefficients to put in the supplied bound—which means that the user has to provide details about the complexity of code that does not yet exist; (ii) the limitation to linear bounds means that the user cannot specify resource bounds that involve logarithms, such as $O(\log n)$ and $O(n \log n)$, common in problems based on divide and conquer.

In this chapter, we introduce SynPlexity, a type-system paired with a type-directed synthesis technique that addresses these issues. In SynPlexity, the user provides as input a refinement type that describes both the functionality and the *asymptotic* (big-O) resource usage of a program. For example, a user might ask SynPlexity to synthesize an implementation of a sorting function with resource usage $O(n \log n)$, where n is the length of the input list. As in prior work, SynPlexity also takes as input a set of auxiliary functions that the synthesized program can use. SynPlexity then uses a type-directed synthesis algorithm to search for a program that has the desired functionality, and satisfies the asymptotic resource bound. SynPlexity's synthesis algorithm uses a new type system that can reason about the asymptotic complexity of functions. To achieve this goal, this type system uses two ideas.

1. The type system uses *recurrence relations* instead of concrete resource potentials [HAH11] to reason about the asymptotic complexity of functions. For example, the recurrence relation $T(u) \leqslant 2T(\lfloor \frac{u}{2} \rfloor) + O(u)$ denotes that on an input of size u, the function will perform at most two recursive calls on inputs of size at most $\lfloor \frac{u}{2} \rfloor$, and will use at most $O(u)$ resources outside of the recursive calls.[1] For a given recurrence relation, our type system uses refinement types to guarantee that a function typed with this recurrence relation performs the correct number of recursive calls on parameters of the appropriate sizes.

2. These typing rules are justified by classic theorems from the field of analysis of algorithms, such as the Master Theorem [CLRS09], the Akra-Bazzi method [AB98], or C-finite-sequence analysis [KP11].

Guéneau et al. observed that reasoning with O-notation can be tricky, and exhibited a collection of plausible-sounding, but flawed, inductive proofs [GCP18, §2]. We avoid this pitfall via SynPlexity's type system, which establishes whether

[1] The recurrence relation above is one possible instantiation of the Master Theorem [CLRS09, §4.5 and §4.6]; it can also be instantiated as $T(u) \leqslant 2T(\lceil \frac{u}{2} \rceil) + O(u)$. The type system makes use of certain templates for instantiating the algorithm-analysis theorems that we use. The use of templates means that the type system does not use all possible instantiations, but all instantiations used in the type system are valid ones.

a term satisfies a given recurrence relation. SynPlexity uses theorems that connect the form of a recurrence relation—e.g., the number of recursive calls, and the argument sizes in the subproblems—to its asymptotic complexity. In particular, the SynPlexity type system does not encode inductive proofs of the kind that Guéneau et al. show can go astray.

SynPlexity can synthesize functions with complexities that cannot be handled by existing type-directed tools [PKSL16, KWPH19], and compares favorably with existing tools on their benchmarks. Furthermore, for some domains, SynPlexity's type system allows us to discover auxiliary functions automatically (e.g., the split function of a merge sort), instead of requiring the user to provide them.

Contributions. The contributions of our work are as follows:

- A type system that uses refinement types to check whether a program satisfies a recurrence relation over a specified resource (§4.3).
- A type-directed algorithm that uses our type system to synthesize functions with given resource bounds (§4.5, §4.6).
- SynPlexity, an implementation of our algorithm that, unlike prior tools, can synthesize programs with desired asymptotic complexities (§4.7).

Complete proofs and details of the type system can be found in the appendices.

4.2 Overview

In this section, we illustrate the main components of our algorithm through an example. Consider the problem of synthesizing a function `prod` that implements the multiplication of two natural numbers, x and y. We want an efficient solution whose time complexity is $O(\log x)$ with respect to the value of the first argument x. In §4.2, we show how existing type-directed synthesizers solve this problem in the absence of a complexity-bound constraint. In §4.2, we illustrate how to specify asymptotic bounds in type-directed synthesis problems. In §4.2, we show how the

tracking of recurrence relations can be used to establish complexity bounds as well as guide the synthesis search.

Type-Directed Synthesis

We first review one of the state-of-the-art type-directed synthesizers, SYNQUID, through the aforementioned example—i.e., synthesizing a program `prod` that computes the product of two natural numbers. In SYNQUID, the specification is given as a refinement type that describes the desired behavior of the synthesized function. We specify the behavior of `prod` using the following refinement-type:

$$\texttt{prod} :: \texttt{x}{:}\{\texttt{Int} \mid v \geqslant 0\} \rightarrow \texttt{y}{:}\{\texttt{Int} \mid v \geqslant 0\} \rightarrow \{\texttt{Int} \mid v = \texttt{x} * \texttt{y}\}.$$

Here the types of the inputs `x` and `y`, as well as the return type of `prod` are refined with predicates. The refinement $\{\texttt{Int} \mid v \geqslant 0\}$ declares `x` and `y` to be non-negative, and the refinement $\{\texttt{Int} \mid v = \texttt{x} * \texttt{y}\}$ of the return type declares the output value to be an integer that is equal to the product of the inputs `x` and `y`. In addition to the specification, the synthesizer receives as input some signatures of auxiliary functions it can use. The specifications of auxiliary functions are also given as refinement types. In our example, we have the following functions:

$$\begin{array}{rcl} \texttt{even} & :: & \texttt{x}{:}\texttt{Int} \rightarrow \{\texttt{Bool} \mid \texttt{x mod } 2 = 0\} \qquad \texttt{dec} :: \texttt{x}{:}\texttt{Int} \rightarrow \{\texttt{Int} \mid v = \texttt{x} - 1\} \\ \texttt{double} & :: & \texttt{x}{:}\texttt{Int} \rightarrow \{\texttt{Int} \mid v = \texttt{x} + \texttt{x}\} \qquad \texttt{div2} :: \texttt{x}{:}\texttt{Int} \rightarrow \{\texttt{Int} \mid v = \lfloor \frac{\texttt{x}}{2} \rfloor\} \\ \texttt{plus} & :: & \texttt{x}{:}\texttt{Int} \rightarrow \texttt{y}{:}\texttt{Int} \rightarrow \{\texttt{Int} \mid v = \texttt{x} + \texttt{y}\} \end{array}$$

With the above specification and auxiliary functions, SYNQUID will output the implementation of `prod` shown in Eqn. (4.1).

$$\texttt{prod = } \lambda \texttt{x}.\lambda \texttt{y}.\ \texttt{if x==0 then x else plus y (prod (dec x) y)} \tag{4.1}$$

SYNQUID uses a sophisticated type system to guarantee that the synthesized term has the desired type. Furthermore, SYNQUID uses its type system to prune the

search space by only enumerating terms that can possibly be typed, and thus meet the specification. Terms are enumerated in a top-down fashion, and appropriate specifications are propagated to sub-terms. As an example, let us see how SYNQUID synthesizes the function body—an `if-then-else` term—in Eqn. (4.1), which is of refinement type $\{\texttt{Int} \mid \nu = \texttt{x} * \texttt{y}\}$. SYNQUID will first enumerate an integer term for the `then` branch—a variable term `x`. Then, with the `then` branch fixed, the condition guard must be refined by some predicate φ under which the `then` branch (the term `x` refined by $\nu = \texttt{x}$) fulfills the goal type $\{\texttt{Int} \mid \nu = \texttt{x} * \texttt{y}\}$, i.e., $\forall \texttt{x}, \texttt{y} \geqslant 0.\varphi \wedge \nu = \texttt{x} \implies \nu = \texttt{x} * \texttt{y}$. With this constraint, SYNQUID identifies the term $\texttt{x} == 0$ as the condition. Finally, SYNQUID propagates the negation of the condition to the `else` branch—the `else` branch should be a term of type $\{\texttt{Int} \mid \nu = \texttt{x} * \texttt{y}\}$ with the path condition $\texttt{x} \neq 0$—and enumerates the term `plus y (prod (dec x) y)` as the else branch, which has the desired type.

The program in Eqn. (4.1) is correct, but inefficient. Let us count each call to an auxiliary function as one step; and let $T(x)$ denote the number of steps in which the program runs with input x. The implementation in Eqn. (4.1) runs in $\Theta(x)$ steps because $T(x)$ satisfies the recurrence $T(x) = T(x-1) + 2$, implying $T(x) \in \Theta(x)$. Because, SYNQUID does not provide a way to specify resource bounds, such as $O(\log x)$; one cannot ask SYNQUID to find a more efficient implementation.

Adding Resource Bounds

In our tool, SYNPLEXITY, one can specify a synthesis problem with an asymptotic resource bound, and can ask SYNPLEXITY to find an $O(\log x)$ implementation of `prod`. To express this intent, the user needs to specify (1) the asymptotic resource-usage bound the synthesized program should satisfy, (2) the cost of each provided auxiliary function, and (3) the size of the input to the program.

Asymptotic Resource Bound. We extend refinement types with resource annotations. The annotated refinement types are of the form $\langle \tau; \alpha \rangle$ where τ is a regular refinement type, and α is a resource annotation. The following example asks the synthesizer to find a solution with the resource-usage bound $O(\log u)$:

$$\texttt{prod} :: \langle \texttt{x}{:}\{\texttt{Int} \mid v \geqslant 0\} \to \texttt{y}{:}\{\texttt{Int} \mid v \geqslant 0\} \to \{\texttt{Int} \mid v = \texttt{x} * \texttt{y}\}, O(\log u)\rangle$$

Cost of Auxiliary Functions. The auxiliary functions supplied by the user serve as the API in terms of which the synthesized program is programmed. Thus, the resource usage of the synthesized program is the sum of the costs of all auxiliary calls made during execution. We allow users to assign a polynomial cost $O(u^a)$, for some constant a, or a constant cost $O(1)$ to each auxiliary function. Here, u is a free variable that represents the size of the problem on which the auxiliary function is called.

In the `prod` example, all auxiliary functions are assigned constant cost, e.g., we give `even` the signature $\texttt{even} :: \langle \texttt{x}:\texttt{Int} \to \{\texttt{Bool} \mid \texttt{x mod } 2 = 0\}, O(1)\rangle$.

Size of Problems. The user needs to specify a size function, $\texttt{size}:\tau \to \texttt{Int}$, that maps inputs to their sizes, e.g., when synthesizing the sorting function for an input of type `list`, the size function can be $\lambda \texttt{l}.|\texttt{l}|$—the length of the input list. In the `prod` example, the size function is $\texttt{size} = \lambda x.\lambda y.x$.

Checking Recurrence Relations

We extend Synquid's refinement-type system with resource annotations, so that the extended type system enforces the resource usage of terms. The idea of the type system is to check if the given function satisfies some recurrence relation. If so, it can infer that the function also satisfies the corresponding resource bound. For example, according to the Master Theorem [BHS80], if a function `f` satisfies the recurrence relation $T(u) \leqslant T(\lfloor \frac{u}{2} \rfloor) + O(1)$ where u is the size of the input, then the resource usage of `f` is bounded by $O(\log u)$. Checking if a function satisfies a given recurrence relation can be performed by checking if the function contains appropriate recursive calls—e.g., if a function contains one recursive call to a sub-problem of half size, and consumes only a constant amount of resources in its body, then it satisfies $T(u) \leqslant T(\lfloor \frac{u}{2} \rfloor) + O(1)$.

The following rule is an example of how we connect recurrence annotations and resource bounds.

$$\frac{x : \tau_1, f : \tau_1 \to \tau_2, \Gamma \vdash t :: \langle \tau_2; ([1, \lfloor \frac{u}{2} \rfloor], O(1)) \rangle}{\Gamma \vdash (\texttt{fix}\ f.\ \lambda x.t) :: \langle \tau_1 \to \tau_2; O(\log u) \rangle}$$

The rule instantiates the Master Theorem example above. Note that, the annotation $([1, \lfloor \frac{u}{2} \rfloor], O(1))$ states that the function body contains up to one recursive call to a problem of size $\lfloor \frac{u}{2} \rfloor$, and the resource usage in the body of t (aside from calls to f itself) is bounded by $O(1)$. The rule states that if the function body t of type τ_2 contains one recursive call to a sub-problem of size $\lfloor \frac{u}{2} \rfloor$, then the function will be bounded by $O(\log u)$.

The implementation of `prod` shown in Eqn. (4.2) runs in $O(\log x)$ steps.

```
prod = λx.  λy.if x == 0 then x else                                    (4.2)
     if even x then  double (prod (div2 x) y)
               else  plus y (double (prod (div2 x) y))
```

To check that, SynPlexity's type system counts the number of recursive calls along any path of the function. There are three paths (two nested if-then-else terms) in the program, and at most one recursive call along each path. Also, one can check that the problem size of each recursive call is no more than $\lfloor \frac{x}{2} \rfloor$. For example, the recursive call `prod (div2 x) y` calls to a problem with size `div2 x`, which is consistent with $[1, \lfloor \frac{u}{2} \rfloor]$, and u is `x` because `size x y` $= x$. In addition, the condition that the resource usage of the body is bounded by $O(1)$ is satisfied because only auxiliary functions with constant cost are called.

4.3 The SynPlexity Type System

In this section, we present our type system. First, we give the surface language and the types, which extend the Synquid liquid-types framework with resource

$$
\begin{array}{lll}
\text{Term} & t ::= & e \mid b \\
\text{E-term} & e ::= & x \mid c \mid \texttt{true} \mid \texttt{false} \mid x\, e_1 \ldots e_n \\
\text{I-term} \left\{ \begin{array}{l} \text{Branching term} \\ \\ \text{Function term} \end{array} \right. & \begin{array}{l} b ::= \\ \\ f ::= \end{array} & \begin{array}{l} \texttt{if}\ e\ \texttt{then}\ t\ \texttt{else}\ t \\ \mid \texttt{match}\ e\ \texttt{with} \mid_i C_i\ (x_i^1 \ldots x_i^n) \mapsto t_i \\ \texttt{fix}\ f.\lambda x_1 \ldots \lambda x_n.t \end{array}
\end{array}
$$

Figure 4.1: SynPlexity syntax.

$$
\begin{array}{lll}
\text{Logical expr.} & \varphi, \phi, \psi ::= & x \mid \mathtt{m}(\psi) \mid \top \mid \bot \mid c \mid \psi\ \mathtt{mod}\ \psi \mid \psi \wedge \psi \mid \psi \vee \psi \\
& & \mid\ \neg\psi \mid \psi = \psi \mid \psi * \psi \mid \psi / \psi \mid \psi + \psi \mid \psi - \psi \\
\text{Ordinary type} & B ::= & \mathtt{Bool} \mid \mathtt{Int} \mid D \\
\text{Refinement type} & \tau ::= & \{B \mid \varphi\} \mid x_1 : \tau \to \ldots \to x_n : \tau_n \to y : \tau \\
\text{Annotated type} & \gamma ::= & \langle \tau; \alpha \rangle \\
\text{Recurrence ann.} & \alpha ::= & (\ [c_1, \phi_1]_f, \ldots, [c_n, \phi_n]_f; O(\psi)\ ) \\
\text{Environment} & \Gamma ::= & \cdot \mid x : \gamma; \Gamma \mid \varphi; \Gamma \mid \mathtt{recFun} := x; \Gamma \mid \mathtt{args} := x_1 \ldots x_n; \Gamma
\end{array}
$$

Figure 4.2: SynPlexity types.

annotations (§4.3). Then, we show the semantics of our language (§4.3). Finally, we present SynPlexity's type system (§4.3), which our synthesis algorithm uses to synthesize programs with desired resource bounds.

Syntax and Types

Syntax. Consider the language shown in Fig. 4.1. In the language, we distinguish between two kinds of terms: *elimination terms* (E-terms) and *introduction terms* (I-terms). E-terms consist of variable terms, constant values c, and application terms. Condition guards and match scrutinies can only be E-terms. I-terms are branching terms and function terms. The key property of I-terms is that if the type of any I-term is known, the types of its sub-terms are also known (which is not the case for E-terms).

Types. Our language of types, presented in Fig. 4.2, extends the one of Syn-

QUID [PKSL16] with *recurrence annotations*, which are used to track recurrence relations on functions. To simplify the presentation, we ignore some of the features of the type system of SYNQUID [PKSL16] that do not affect our algorithm. In particular, we do not discuss polymorphic types and the enumerating strategy that ensures that only terminating programs are synthesized. However, our implementation is built on top of SYNQUID, and supports both of those features.

Logical expressions are built from variables, constants, arithmetic operators, and other user-defined logical functions. Logical expressions in our type system can be used as refinements φ, size expressions ϕ, or bound expressions ψ. Refinements φ are logical predicates used to refine ordinary types in refinement types $\{B \mid \varphi\}$. We usually use a reserved symbol v as the free variable in φ, and let v represents the inhabitants, i.e., inhabitants of the type $\{B \mid \varphi\}$ are valuations of v that satisfy φ. For example, the type $\{\texttt{Int} \mid v \texttt{ mod } 2 = 0\}$ represents the even integers. Size expressions and bound expressions are used in recurrence annotations, and are explained later.

Ordinary types includes primitive types and user-defined algebraic datatypes D. Datatype constructors C are functions of type $\tau_1 \to \ldots \to \tau_n \to D$. For example, the datatype `List(Int)` has two constructors: `Cons : Int` $\to$ `List(Int)` $\to$ `List(Int)`, and `Nil : List(Int)`. Refinement types are ordinary types refined with some predicates ψ, or arrow types. Note that, unlike SYNQUID's type system, SYNPLEXITY's type system does not support higher-order functions—i.e., arguments of functions have to be non-arrow types. All occurrences of τ_i and τ in arrow types $x_1 : \tau \to \ldots \to x_n : \tau_n \to y : \tau$ have to be ordinary types or refined ordinary types.

We use `recFun` to denote the name of the function for which we are performing type-checking, and `args` to denote the tuple of arguments to `recFun`. For example, in the function `prod` shown in Eqn. (4.1), `recFun=prod` and `args=x y` An environment Γ is a sequence of variable bindings $x : \gamma$, path conditions φ, and assignments for variables `recFun` and `args`.

Recurrence Annotations. Annotated types are refinement types annotated with recurrence annotations. A recurrence annotation is a pair $([c_1, \phi_1]_f, \ldots, [c_n, \phi_n]_f;\ O(\psi))$ consisting of (1) a set of recursive-call costs of the form $[c_i, \phi_i]_f$, and (2) a resource-usage bound of the form $O(\psi)$. Intuitively, a recurrence annotation tracks the

number c_i of recursive calls to f of size ϕ_i in the first element $[c_1, \phi_1]_f, \ldots, [c_n, \phi_n]_f$ of the pair, as well as the asymptotic resource usage of the *body* of the function (the second element $O(\psi)$). Using these quantities, we can compute a recurrence relation describing the resource usage of the function `recFun`. For example, the recurrence annotation $([1, u-1]_f, [1, u-2]_f; O(1))$ corresponds to the recurrence relation $T_f(u) \leqslant T_f(u-1) + T_f(u-2) + O(1)$.

A *recursive-call cost* $[c, \phi]_f$ associated with a function f denotes that the body of f can contain up to c recursive calls to subproblems that have sizes up to the one specified by size expression ϕ. A size expression, ϕ, is a polynomial over a reserved variable symbol u that represents the size of the top-level problem. In this chapter, a *problem* with respect to a function $g :: x_1 : \tau_1 \to \ldots \to x_n : \tau_n \to y : \tau$ is a tuple of terms $e_1 \ldots e_n$, to which g can be applied—i.e., e_i has type τ_i for all i from 1 to n. For the problems of function g, the size of each problem is defined by a *size function* $\texttt{size}_g$—a user-defined logical function that has type $\tau_1 \to \ldots \to \tau_n \to \texttt{Int}$; i.e., it takes a problem of g as input and outputs a non-negative integer. In the body of g, we say that a recursive-call term $g\ e_1 \ldots e_n$ *satisfies* a size expression ϕ if for all $x_1, \ldots, x_n$, $\texttt{size}_g\ [\![e_1]\!] \ldots [\![e_n]\!] \leqslant [(\texttt{size}_g\ x_1 \ldots x_n)/u]\phi$, where the x_i's are the arguments of g and the $[\![e_i]\!]$'s are the evaluations of e_i on input $x_1 \ldots x_n$. (See §4.3 for the formal definition of $[\![\cdot]\!]$.) Note that one annotation can contain multiple recursive-call costs, which allows the function to make recursive calls to sub-problems with different sizes. We often abbreviate $\langle \tau, (O(1)) \rangle$ as τ and omit f in recursive-call costs if it is clear from context.

A resource bound $O(\psi)$ of a non-arrow type specifies the bound of the resource usage strictly within the top-level-function body. A resource bound in a signature of an auxiliary function f specifies the resource usage of f. *Bound expressions* ψ in $O(\psi)$ are of the form $u^a \log^b u + c$ where a, b, and c are all non-negative constants, and u represents the size of the top-level problem.

Example 4.1. *In the function* `prod` *(Eqn. (4.2)), the recursive-call term* `prod (div2 x)`

y satisfies the recursive-call cost $[1, \lfloor \frac{u}{2} \rfloor]$, *because* $\mathit{size}_{\mathit{prod}} = \lambda z.\lambda w.z$, *and*

$$\mathit{size}_{\mathit{prod}}\ [\![(\mathit{div2}\ x)]\!]\ [\![y]\!] = [\![\mathit{div2}\ x]\!] = \lfloor \frac{x}{2} \rfloor = [(\mathit{size}_{\mathit{prod}}\ x\ y)/u] \lfloor \frac{u}{2} \rfloor.$$

Semantics and Cost Model

We introduce the *concrete*-cost semantics of our language here. The semantics serves two goals: (1) it defines the evaluation of terms (i.e., how to obtain values), which can be used to compute the sizes of problems in application expressions, and (2) it defines the resource usages of terms.

Besides the syntax shown in Fig. 4.1, implementations of auxiliary functions can contain calls to a tick function $\texttt{tick}(c, t)$, which specifies that c units of a resource are used, and the overall value is the value of t. Note that in our synthesis language, we are not actually synthesizing programs with `tick` functions. We assume that `tick` functions are only called in the implementations of auxiliary functions. In the concrete-cost semantics, a configuration $\langle t, C \rangle$ consists of a term t and a nonnegative integer C denoting the resource usage so far. The evaluation judgment $\langle t, C \rangle \hookrightarrow \langle t', C + C_\Delta \rangle$ states that a term t can be evaluated in one step to a term (or a value) t', with resource usage C_Δ. We write $\langle t, C \rangle \hookrightarrow^* \langle t', C + C_\Delta \rangle$ to indicate the reduction from t to t' in zero or more steps. All of the evaluation judgments are standard, and are shown in §4.4. Here we show the judgment of the tick function, where resource usage happens.

$$\frac{}{\langle \texttt{tick}(c, t), C \rangle \hookrightarrow \langle t, C + c \rangle}\ \textsc{Sem-Tick}$$

For a term t, $[\![t]\!]$ denotes the evaluation result of t, i.e., $\langle t, \cdot \rangle \hookrightarrow^* \langle [\![t]\!], \cdot \rangle$.

Example 4.2. *Consider the following function that doubles its input.*

$$\mathit{fix}\ \mathit{double}.\lambda x.\mathbf{if}\ x = 0\ \mathbf{then}\ 0\ \mathbf{else}\ \mathit{tick}(1, 2 + \mathit{double}(x-1)).$$

Let t_{body} *denote the function body* $\mathbf{if}\ x{=}0\ \mathbf{then}\ 0\ \mathbf{else}\ \mathit{tick}(1, 2{+}\mathit{double}(x{-}1))$. *The*

result of evaluating `double` *on input* 5 *is* 10*, with resource usage* 5*.*

$$
\begin{aligned}
& \langle (\texttt{fix double}.\lambda x.\mathrm{t}_{body})\,5, 0\rangle \\
\hookrightarrow\ & \langle \texttt{if 5=0 then 0 else tick(1,2+double(4))}, 0\rangle \\
\hookrightarrow\ & \langle \texttt{if false then 0 else tick(1,2+ double(4))}, 0\rangle \\
\hookrightarrow\ & \langle \texttt{tick(1,2+double(4))}, 0\rangle \hookrightarrow \langle \texttt{2+double(4)}, 1\rangle \\
\hookrightarrow\ & \langle \texttt{2+(fix double}.\lambda x.\mathrm{t}_{body}\texttt{)}4, 1\rangle \hookrightarrow^* \langle \texttt{4+double(3)}, 2\rangle \hookrightarrow^* \langle \texttt{10+double(0)}, 5\rangle \\
\hookrightarrow\ & \langle \texttt{10+(if 0=0 then 0 else tick(1,2+double(0-1)))}, 5\rangle \\
\hookrightarrow\ & \langle \texttt{10+(if true then 0 else tick(1,2+double(0-1)))}, 0\rangle \hookrightarrow \langle \texttt{10+0}, 5\rangle
\end{aligned}
$$

With the standard concrete semantics, the complexity of a function f is characterized by its resource usage when the function is evaluated on inputs of a given size.

Definition 4.3 (Complexity)**.** *Given a function* $\texttt{fix}\ f.\lambda\overline{y}.t$ *of type* $: \tau_1 \to \tau_2$*, with size function* $\texttt{size}_f : \tau_1 \to \mathbb{N}$*, and suppose that for any possible input* $\overline{x}$*, the configuration* $\langle (\texttt{fix}\ f.\lambda\overline{y}.t)\overline{x}, 0\rangle$ *can be reduced to* $\langle v, C_{\overline{x}}\rangle$ *for some value* v*. Then, if* $T_f : \mathbb{N} \to \mathbb{N}$ *is a function such that, for all,* $u \geqslant 0$*,* $T_f(u) = \sup_{\overline{x}\ s.t.\ \texttt{size}_f(\overline{x})=u} C_{\overline{x}}$*, we say that* T_f *is* **the complexity function of** f*.*

Note that Defn. 4.3 assumes that the top-level term $(\texttt{fix}\ f.\lambda\overline{y}.t)\overline{x}$ can be reduced to some value. Thus, Defn. 4.3 only applies to terminating programs.

Definition 4.4 (Big-O notation)**.** *Given two integer functions* f *and* g*, we say that* f *dominates* g*, i.e.,* $g \in O(f)$*, if* $\exists c, M \geqslant 0.\ \forall x \geqslant c.\ g(x) \leqslant Mf(x)$*.*

In the rest of this chapter, we use T_f to denote the complexity function of the function f, and we say the complexity of f is *bounded* by a function g if $T_f \in O(g)$. As an example, the complexity of the `double` function shown in Ex. 4.2 is $T_{\texttt{double}}(u) := u$, and hence $T_{\texttt{double}}(u) \in O(u)$.

Auxiliary functions. We allow users to supply signatures for auxiliary functions, instead of implementations. It is an obligation on users that such signatures be sen-

sible; in particular, when the user gives the signature $\langle \tau_1 \to \{B \mid \varphi(v, \overline{y})\}, O(\psi(u))\rangle$ for auxiliary function f, the user asserts that there exists some implementation $\texttt{fix}\ f.\lambda\overline{y}.t$ of f, such that: 1)for any input $\overline{x}$, the output of f on $\overline{x}$ satisfies φ, i.e., $\varphi([\![(\texttt{fix}\ f.\lambda\overline{y}.t)\overline{x}]\!], \overline{x})$ is valid; and 2)for any input $\overline{x}$, the complexity of f is bounded by $\psi(u)$, i.e., $T_f(u) \in O(\psi(u))$. Signatures always over-approximate their implementations, as illustrated by the following example.

Example 4.5. *The signature* $\texttt{doubleRelaxed} :: \langle x{:}\texttt{Int} \to \{\texttt{Int} \mid v \leqslant 3 * x\}, O(u^2)\rangle$ *describes an auxiliary function that computes no more than the input times 3, and has quadratic resource usage. Note that the function* `double` *shown in Ex. 4.2 can be an implementation of this signature because* $[\![\texttt{double}(\overline{x})]\!] = 2 * x \leqslant 3 * x$*, and the complexity function* $T_{\texttt{double}}(u) = u$ *is in* $O(u^2)$.

Typing Rules

The typing rules of SynPlexity are inspired by bidirectional type checking [PT00] and type checking with cost sharing [KWPH19]. Recall that we use `recFun` to denote the name of the function for which we are performing type-checking, and `args` to denote the tuple of arguments to `recFun`.

An environment Γ is a sequence of variable bindings of the form $x : \gamma$, path conditions φ, and assignments of the form $x = \varphi$ for `recFun` and the components of `args`. SynPlexity's typing rules use three judgments:

- $\Gamma \vdash t :: \gamma$ states that t has type γ,
- $\Gamma \vdash \gamma_1 <: \gamma_2$ states that γ_2 is a subtype of γ_1, and
- $\Gamma \vdash \gamma \curlyvee \gamma_1 | \gamma_2$ states that γ_1 and γ_2 share the costs in γ

Subtyping. Subtyping judgments are shown in Fig. 4.3. The <:-Fun, <:-Sc, and <:-Refl are standard subtyping rules for refinement types. The remaining rules allow us to compare resource consumption of recurrence annotations. For example, if one branch of some branching term has type $\langle \tau, ([1, \lfloor \frac{u}{3} \rfloor], O(\psi))\rangle$, it can be over-approximated by a super type $\langle \tau, ([1, \lfloor \frac{u}{2} \rfloor], O(\psi))\rangle$. The idea is that the resource

$$\boxed{\Gamma \vdash \tau <: \tau'} \qquad \frac{\Gamma \vdash \tau_y <: \tau_x \qquad \Gamma; y : \tau_y \vdash [y/x]\tau <: \tau'}{\Gamma \vdash x : \tau_x \to \tau \;<:\; y : \tau_y \to \tau'} <:\text{-Fun}$$

$$\frac{\Gamma \vdash B <: B' \qquad \Gamma \models \varphi \implies \varphi'}{\Gamma \vdash \{B \mid \varphi\} <: \{B' \mid \varphi'\}} <:\text{-Sc} \qquad \frac{}{\Gamma \vdash B \;<:\; B} <:\text{-Refl}$$

$$\frac{c' > c \quad \Gamma \models [x/v]\phi \wedge [x'/v]\phi' \Rightarrow x' > x}{\Gamma \vdash [c, \phi] <: [c', \phi']} <:\text{-Rec} \qquad \frac{\Gamma \models \psi \in O(\psi')}{\Gamma \vdash O(\psi) <: O(\psi')} <:\text{-Bound}$$

<:-Rec-Split

$$\frac{\Gamma \models \phi_1 = \phi_2}{\Gamma \vdash \langle \tau', ([c_1 + c_2, \phi_1], [c_3, \phi_3] \dots [c_n, \phi_n]; O(\psi))\rangle <: \langle \tau, ([c_1, \phi_1], [c_2, \phi_2], \dots [c_n, \phi_n]; O(\psi))\rangle}$$

<:-Rec-Combine

$$\frac{\Gamma \models \phi_1 = \phi_2}{\Gamma \vdash \langle \tau, ([c_1, \phi_1], [c_2, \phi_2], \dots [c_n, \phi_n]; O(\psi))\rangle <: \langle \tau', ([c_1 + c_2, \phi_1], [c_3, \phi_3] \dots [c_n, \phi_n]; O(\psi))\rangle}$$

$$\frac{\Gamma \vdash \tau <: \tau' \qquad \forall i. \Gamma \vdash [c_i, \phi_i] <: [c'_i, \phi'_i] \qquad \Gamma \vdash O(\psi) <: O(\psi')}{\Gamma \vdash \langle \tau, ([c_1, \phi_1], \dots [c_n, \phi_n]; O(\psi))\rangle <: \langle \tau', ([c'_1, \phi'_1] \dots [c'_n, \phi'_n]; O(\psi'))\rangle} <:\text{-Anno-Rec}$$

Figure 4.3: Subtyping rules.

usage of an application calling to a problem of size $\lfloor \frac{u}{2} \rfloor$, will be larger than the application calling to a smaller problem of size $\lfloor \frac{u}{3} \rfloor$ (assuming all resource usages are monotonic).

Subtyping rules also allow the type system to compare branches with a different number of recursive calls. For example, base cases of recursive procedures have no recursive calls, and thus have types of the form $\langle \tau, ([], O(\psi))\rangle$. With subtyping, these types can be over-approximated by types of the form $\langle \tau, ([c, \phi], O(\psi))\rangle$.

Cost sharing. When a term has more than one sub-term in the same path, e.g., the condition guard and the `then` branch are in the same path in an `ite` term, the recursive-call costs of the term will be shared into its sub-terms. The sharing operator $\alpha \curlyvee \alpha_1 | \alpha_2$ partitions the recursive-call costs of α into α_1 and α_2—i.e., the sum of the costs in α_1 and α_2 equals the costs in α. Sharing rules are shown in

$$\boxed{\Gamma \vdash \gamma \curlyvee \gamma_1 | \gamma_2} \qquad \frac{c_1, c_2 \geqslant 0 \qquad c_1 + c_2 \leqslant c}{\Gamma \vdash [c, \phi] \curlyvee [c_1, \phi] \mid [c_2, \phi]}\ \text{S-Pot} \qquad \frac{}{\Gamma \vdash [c, \phi] \curlyvee [c, \phi]}\ \text{S-Refl}$$

$$\frac{\forall i. \Gamma \vdash [c_i, \phi_i] \curlyvee [c_i^1, \phi_i] \mid [c_i^2, \phi_i]}{\Gamma \vdash ([c_1, \phi_1], \ldots, [c_n, \phi_n]; O(\psi)) \curlyvee ([c_1^1, \phi_1], \ldots, [c_n^1, \phi_n]; O(\psi)) \mid ([c_1^2, \phi_1], \ldots, [c_n^2, \phi_n]; O(\psi))}\ \text{S-Anno}$$

$$\frac{\Gamma \vdash \alpha \curlyvee \alpha_1 \mid \alpha_2}{\Gamma \vdash \langle \tau, \alpha \rangle \curlyvee \langle \tau, \alpha_1 \rangle \mid \langle \tau, \alpha_2 \rangle}\ \text{S-Type} \qquad \frac{\Gamma \vdash \gamma \curlyvee \gamma \mid \gamma' \quad \Gamma \vdash \gamma' \curlyvee \gamma_2 \mid \gamma_3}{\Gamma \vdash \gamma \curlyvee \gamma_1 \mid \gamma_2 \mid \gamma_3}\ \text{S-Mul}$$

Figure 4.4: Sharing rules

Fig. 4.4. The idea is that a single cost c can be shared to two costs c_1 and c_2 such that their sum is no more than c. An annotation can be shared to two parts if every recursive cost $[c_i, \phi_i]$ in it can be shared to two parts $[c_i^1, \phi_1]$ and $[c_i^2, \phi_2]$. Finally, annotations can also be shared to more than two parts.

Example 4.6. *There are multiple ways to share the annotation* $([1, \lfloor \frac{u}{2} \rfloor], [1, \lceil \frac{u}{2} \rceil]; O(u))$*:*

$$\Gamma \vdash ([1, \lfloor \frac{u}{2} \rfloor], [1, \lceil \frac{u}{2} \rceil]; O(u)) \curlyvee ([1, \lfloor \frac{u}{2} \rfloor], [1, \lceil \frac{u}{2} \rceil]; O(u)) \mid ([\,], O(u)),$$

where one annotation contains both recursive-call costs $[1, \lfloor \frac{u}{2} \rfloor], [1, \lfloor \frac{u}{2} \rfloor]$*; and the other contains no recursive-call cost. And*

$$\Gamma \vdash ([1, \lfloor \frac{u}{2} \rfloor], [1, \lceil \frac{u}{2} \rceil]; O(u)) \curlyvee ([1, \lfloor \frac{u}{2} \rfloor]; O(u)) \mid ([1, \lceil \frac{u}{2} \rceil]; O(u)),$$

where each annotation contains one recursive-call cost.

Function terms. The rule T-Abs shown below is really a rule-schema that is parameterized in terms of an annotation (A) for a function body t, and a resource bound (B) for the function term. If the function body t has some recurrence relation described by the annotation A, then the function f will satisfy the resource-usage bound B.

Table 4.1: Annotations that can be used to instantiate the rule T-Abs.

	Bound (B)	Recurrence relation	Annotation (A)
Master Theorem	$O(\log u)$	$T(\lfloor \frac{u}{d} \rfloor) + O(1),\ d \geqslant 2$	$([1, \lfloor \frac{u}{d} \rfloor]; O(1)),\ d \geqslant 2$
	$O(u \log u)$	$dT(\lfloor \frac{u}{d} \rfloor) + O(u),\ d \geqslant 2$	$([d, \lfloor \frac{u}{d} \rfloor]; O(u)),\ d \geqslant 2$
Akra–Bazzi	$O(u \log u)$	$T(\lceil \frac{u}{2} \rceil) + T(\lfloor \frac{u}{2} \rfloor) + O(u)$	$([1, \lceil \frac{u}{2} \rceil], [1, \lfloor \frac{u}{2} \rfloor]; O(u))$
C-Finite Seq.	$O(u)$	$T(u - d) + O(1),\ d \geqslant 1$	$([1, u - d]; O(1)),\ d \geqslant 1$
	$O(u^2)$	$T(u - d) + O(u),\ d \geqslant 1$	$([1, u - d]; O(u)),\ d \geqslant 1$

Some example patterns are shown in Tab. 4.1.[2]

$$\frac{\begin{array}{c}\Gamma' = [\texttt{recFun} \leftarrow \texttt{f}][\texttt{args} \leftarrow x_1 \dots x_n]\Gamma \\ \gamma_f = \langle x_1 : \tau_1 \to \dots \to x_n : \tau_n \to \tau, (B)\rangle \\ \Gamma'; x_1 : \langle \tau_1, O(1)\rangle; \dots; x_n : \langle \tau_n, O(1)\rangle; \texttt{f} : \gamma_f \vdash t :: \langle \tau, (A)\rangle\end{array}}{\Gamma \vdash \texttt{fix f}.\lambda x_1 \dots \lambda x_n.t :: \langle x_1 : \tau_1 \to \dots \to x_n : \tau_n \to \tau, (B)\rangle}\ \text{T-Abs}$$

For example, if the annotation of the function body is $([1, \lfloor \frac{u}{2} \rfloor]; O(1))$, then the resource bound in the function type will be $O(\log u)$, i.e., the resource usage of `f` is bounded by $O(\log(\texttt{size}_f\ x_1 \dots x_n))$.

At the same time, the rule stores the name `f` of the recursive function into `recFun`, and its arguments as a tuple into `args`.

Example 4.7. *We use a function* $\texttt{fix bar}.\lambda x.\texttt{if}\ x = 1\ \texttt{then}\ 1\ \texttt{else}\ 1 + \texttt{bar}(\texttt{div2}\ x)$ *to illustrate the first pattern in Tab. 4.1. The body of* `bar` *has the annotated type* $([1, \lfloor \frac{u}{2} \rfloor]; O(1))$ *because* (i) *there exists only one recursive call to a sub-problem whose size is half of the top-level problem size* u*, and* (ii) *the resource usage inside the body is constant* (*with the assumption that all auxiliary functions have constant resource usage*)*. This type appears in row 1, column 4 of Tab. 4.1. Consequently, the recurrence relation of* `bar` *is* $T(\lfloor \frac{u}{2} \rfloor) + O(1)$ (*row 1, column 3*)*, where* $T(u)$ *is the resource usage of* `bar` *on problems with size* u*. Finally, according to the Master Theorem, the resource usage of* `bar` *is bounded by* $O(\log u)$ (*row 1, column 2*)*.*

[2]The patterns shown in Tab. 4.1 are those we used in the implementation. Patterns capturing other recurrence relations can be added to the type system if needed.

Branching terms. In rule T-If, the condition has type `Bool` with refinement φ_e. Two branches have different types—the `then` branch follows the path condition φ_e, and the refinement φ of the branch term, while the `else` branch follows the path condition $\neg\varphi_e$. By having both branches share the same recurrence annotation, T-If can introduce some imprecision. In particular, if the branches belong to different complexity classes, the annotation of the conditional term will be the upper bound of both branches.

$$\frac{\Gamma \vdash \alpha \curlyvee \alpha_1|\alpha_2 \quad \Gamma \vdash e :: \langle\{\texttt{Bool} \mid \varphi_e\}, \alpha_1\rangle \quad \Gamma, \varphi_e \vdash t_1 :: \langle\{B \mid \varphi\}, \alpha_2\rangle \quad \Gamma, \neg\varphi_e \vdash t_2 :: \langle\{B \mid \varphi\}, \alpha_2\rangle}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ t_1\ \texttt{else}\ t_2 :: \langle\{B \mid \varphi\}, \alpha\rangle}\ \text{T-If}$$

The rule T-Match is slightly different: (1) there can be more than two branches, (2) all branches have the same type $\langle\tau, \alpha_2\rangle$, and (3) variables in each case $C_i\ (x_i^1 \ldots x_i^n)$ are introduced in the corresponding branch.

$$\frac{\Gamma \vdash \alpha \curlyvee \alpha_1|\alpha_2 \quad \Gamma \vdash e :: \langle\tau_s, \alpha_1\rangle \quad C_i = \tau_1 \to \ldots \to \tau_n \to \tau_s \qquad \Gamma; x_i^1 : \tau_1; \ldots; x_i^n : \tau_n \vdash t_i :: \langle\tau, \alpha_2\rangle}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ |_i\ C_i\ (x_i^1 \ldots x_i^n) \mapsto t_i :: \langle\tau, \alpha\rangle}\ \text{T-Match}$$

E-terms. The typing rules for E-terms are shown in Fig. 4.5. The two rules for application terms are the key rules of our type system. Let us first look at the E-RecApp rule for recursive-call terms. Recall that the recursive-call annotation tracks the number of recursive calls and the sizes of sub-problems. If the term $\texttt{f}\ e_1 \ldots e_n$ is a recursive call—i.e., $\Gamma(\texttt{recFun}) = \texttt{f}$—the number of recursive calls in one of the recursive-call costs will increase by one—i.e., $[c_k, \phi_k]$ in the premise becomes $[c_k + 1, \phi_k]$ in the conclusion. Also, we want to make sure that the size of the subproblem this application term is called on satisfies the size expression ϕ_k. If each callee term is refined by the predicate φ_i, i.e., $\Gamma \vdash e_i :: \langle\{B_i \mid \varphi_i\}, \alpha_i\rangle$, then the fact that the size of the problem $e_1 \ldots e_n$ satisfies ϕ_k can be implied by

$$\boxed{\Gamma \vdash e :: \gamma} \qquad \frac{\Gamma \vdash x :: \gamma' \qquad \gamma' <: \gamma}{\Gamma \vdash x :: \gamma}\ \text{E-SubType} \qquad \frac{\Gamma(x) = \gamma}{\Gamma \vdash x :: \gamma}\ \text{E-Var}$$

$$\frac{\begin{array}{c}\Gamma \vdash g : \langle x_1{:}\tau_1 \to \ldots \to x_m{:}\tau_m \to \{B \mid \varphi\}, (O(\psi_g))\rangle \\ \Gamma(\texttt{recFun}) \neq g \qquad \Gamma \vdash ([c_1,\phi_1],\ldots,\ldots,[c_n,\phi_n]; O(\psi)) \curlyvee \alpha_1|\ldots|\alpha_m \\ \forall 1 \leqslant i \leqslant m \qquad \Gamma \vdash e_i :: \langle \{B_i \mid \varphi_i\}, \alpha_i\rangle \qquad \Gamma \vdash \{B_i \mid \varphi_i\} <: \tau_i \\ \Gamma \models \bigwedge_{i=1}^{m} [y_i/v]\varphi_i \Rightarrow \left([\texttt{size}_g\ y_1 \ldots y_m/u]\psi_g \in O([\texttt{size}\ \Gamma(\texttt{args})/u]\psi)\right) \\ \tau = \{B \mid [z_i/x_i]\varphi \wedge \bigwedge_{i=1} [z_i/v]\varphi_i\}\ z_i \notin FV(\varphi), z_i \notin FV(\varphi_i)\end{array}}{\Gamma \vdash g e_1 \ldots e_m :: \langle \tau, ([c_1,\phi_1],\ldots,[c_n,\phi_n]; O(\psi))\rangle}\ \text{E-App}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{f} : \langle x_1{:}\tau_1 \to \ldots \to x_m{:}\tau_m \to \{B \mid \varphi\}, \alpha\rangle \qquad \Gamma(\texttt{recFun}) = \texttt{f} \\ \Gamma \vdash ([c_1,\phi_1],\ldots,[c_k,\phi_k],\ldots,[c_n,\phi_n]; O(\psi)) \curlyvee \alpha_1|\ldots|\alpha_m \\ \forall 1 \leqslant i \leqslant m \qquad \Gamma \vdash e_i :: \langle \{B_i \mid \varphi_i\}, \alpha_i\rangle \qquad \Gamma \vdash \{B_i \mid \varphi_i\} <: \tau_i \\ \Gamma \models \bigwedge_{i=1}^{m} [y_i/v]\varphi_i \Rightarrow (\texttt{size}\ y_1 \ldots y_m \leqslant [\texttt{size}\ \Gamma(\texttt{args})/u]\phi_k) \\ \tau = \{B \mid [z_i/x_i]\varphi \wedge \bigwedge_{i=1} [z_i/v]\varphi_i\}\ z_i \notin FV(\varphi), z_i \notin FV(\varphi_i)\end{array}}{\Gamma \vdash \texttt{f} e_1 \ldots e_m :: \langle \tau, ([c_1,\phi_1],\ldots[c_k+1,\phi_k],\ldots,[c_n,\phi_n]; O(\psi))\rangle}\ \text{E-RecApp}$$

Figure 4.5: Typing rules of E-terms

the validity of the predicate $\bigwedge_{i=1}^{m}[y_i/v]\varphi_i \Rightarrow (\texttt{size}\ y_1 \ldots y_m \leqslant [\texttt{size}\ \Gamma(\texttt{args})/u]\phi_k)$. We introduce validity checking, written $\Gamma \models \varphi$, to state that a predicate expression φ is always true under any instance of the environment Γ.

Example 4.8. *Recall Eqn. (4.2). According to the rule T-RecApp, the recursive call* `prod (div2 x) y` *has type* $\langle \{Int \mid v = \lfloor \frac{x}{2} \rfloor * y\}, ([1, \frac{u}{2}]); O(1)\rangle$. *Note that the first argument* `(div2 x)` *has type* $\{Int \mid v = \lfloor \frac{x}{2} \rfloor\}$, *the second argument* `y` *has type* $\{Int \mid v = y\}$, *the size function is* $\texttt{size}_{prod} = \lambda z.\lambda w.z$, *and the arguments in the context are* $\Gamma(\texttt{args}) =$

$x\ y$. *Therefore, the following predicate is valid:*

$$[y_1/v](v = \lfloor \frac{x}{2} \rfloor) \wedge [y_2/v](v = y) \Rightarrow \mathit{size}_{\mathit{prod}}\ y_1\ y_2 = [\mathit{size}_{\mathit{prod}}\ \Gamma(\mathit{args}/u)]\lfloor \frac{u}{2} \rfloor$$
$$\Leftrightarrow \quad (y_1 = \lfloor \frac{x}{2} \rfloor) \wedge (y_2 = y) \Rightarrow y_1 = \lfloor \frac{x}{2} \rfloor.$$

The rule E-App states that callees have types τ_i, and the resource usage does not exceed the bound $O(\psi)$ in the annotation. Similar to the E-RecApp rule, the size of the problem g calls to is $[\mathtt{size}_g\ y_1 \ldots y_m/u]$ with the premise $\bigwedge_{i=1}^{m}[y_i/v]\varphi_i$. The validation checking $\bigwedge_{i=1}^{m}[y_i/v]\varphi_i \Rightarrow \left([\mathtt{size}_g\ y_1 \ldots y_m/u]\psi_g \in O([\mathtt{size}\ \Gamma(\mathtt{args})/u]\psi)\right)$ in the rule states that for any instance of Γ, the size of the problem in the application term is in the big-O class $O([\mathtt{size}\ \Gamma(\mathtt{args})/u]\psi)$. Note that the membership of big-O classes can be encoded as an $\exists\forall$ query. The query is non-linear, and hence undecidable in general. However, we observed in our experiments that for many benchmarks the query stays linear. Furthermore, even when the query is non-linear, existing SMT solvers are capable of handling many such checks in practice.

Soundness

We assume that the resource-usage function ψ and the complexities T of each function are all nonnegative and monotonic integer functions—both the input and the output are integers. We show soundness of the type system with respect to the resource model. The soundness theorem states that if we derive a bound $O(\psi)$ for a function `f`, then the complexity of `f` is bounded by ψ.

Theorem 4.9 (Soundness of type checking)**.** *Given a function* $\mathit{fix}\ f.\lambda x_1 \ldots \lambda x_n.t$ *and an environment* Γ*, if* $\Gamma \vdash \mathit{fix}\ f.\lambda x_1 \ldots \lambda x_n.t :: \langle \tau, O(\psi)\rangle$*, then the complexity of* f *is bounded by* ψ*.*

Our type system is incomplete with respect to resource usage. That is, there are functions in our programming language that are actually in a complexity class $O(p(x))$, but cannot be typed in our type system. The main reason why our type system is incomplete is that it ignores condition guards when building recurrence

relations, and over-approximates `if-then-else` terms by choosing the largest complexity among all the paths including even unreachable ones.

4.4 Semantics

In this section, we presented two kinds of semantics: 1) the concrete small step semantics which define the concrete complexity of functions, and 2) a loose semantics which over-approximates the concrete semantics and will be used in the proof of the soundness theorem.

Concrete semantics. The evaluation rules of concrete-cost semantics are shown in Fig. 4.6. In the concrete-cost semantics, a configuration $\langle t, C\rangle$ consists of a term t and a nonnegative integer C denoting the resource usage so far. The evaluation judgment $\langle t, C\rangle \hookrightarrow \langle t', C + C_\Delta\rangle$ states that a term t can be evaluated in one step to a term (or a value) t', with resource usage C_Δ. We write $\langle t, C\rangle \hookrightarrow^* \langle t', C + C_\Delta\rangle$ to indicate the reduction from t to t' in zero or more steps.

Polynomial-Bounded Refinement Type. Before introducing the loose semantics, we introduce a class of refinement type that can be over-approximated as polynomials.

The resource usage of a function call depends on the size of the problem it makes calls to. However, predicates used to refine functions could be imprecise, and thus we may have to reason about sizes of problems with imprecise refinements. For example, consider a function $\texttt{square} :: \langle \texttt{Int} \to \{\texttt{Int} \mid v \geqslant 0\}, (O(u))\rangle$. In the nested application term `square (square x)`, we know the size of the problem of the inner application is `x`, but we do not know the size of the problem `square x` of the outer application (all we know about `square x` is that it is non-negative). To reason about input sizes, we introduce a class of refinement types with which we can infer upper bounds of the sizes of problems.

Let us first assume that all terms and variables are integers or integer functions (we will discuss the case where terms and variables hold values other than integers later). For an integer refinement type $\tau = \{\texttt{Int} \mid \varphi\}$ refined by a predicate $\varphi(v, \overline{x})$ over v and a tuple of variables $\overline{x}$, we say that φ is *bounded* by a *polynomial* term $p(\overline{x})$,

written as $\varphi \sqsubset p$ if

$$\exists \overline{c} > \overline{0} \forall \overline{x} \forall v.\, \overline{x} > \overline{c} \wedge \varphi(v, \overline{x}) \implies |v| < p(\overline{x}).$$

That is, there exist some positive constants $\overline{c}$ such that, for any $\overline{x}$ (point-wise) greater than $\overline{c}$ and for any v, if $v, \overline{x}$ satisfying φ, the absolute value $|v|$ is always less than $p(\overline{x})$.

Example 4.10. *The refinement type* $\tau := \{\mathit{Int} \mid v \leqslant 2x + y \vee v \leqslant x + 2y\}$ *is bounded by the expression* $p_1 := 2x + 2y$ *and the expression* $p_2 := 3x + 3y$ *but not* $p_3 := 2x + y + 1$, *i.e.,* $\tau \sqsubset p_1$, $\tau \sqsubset p_2$, *and* $\tau \not\sqsubset p_3$.

For datatype D other than `Int`, we assume that there is an *intrinsic measure* with output type `Int` for every D, denoted by $|\cdot|_D$ (we omit the subscript if it is clear from the context). Intrinsic measures are specified by users for user-defined datatypes. For example, the intrinsic measure of lists can be defined as a function that computes the length of lists, i.e., $|l| = \texttt{len}\ l$ for any list l. The instinct measure of `Int` term is the absolute-value function. The condition of p bounding $\tau = \{D \mid \varphi(v, \overline{x})\}$ becomes

$$\exists \overline{c} > \overline{0} \forall \overline{x} \forall v\ \overline{|x|} > \overline{c} \wedge \varphi(v, \overline{x}) \implies |v| < p(\overline{|x|}).$$

A loose cost model. The semantics given previously give a standard notion of complexity. However, we find two challenges connecting these semantics to our synthesis algorithm. First, we allow users to supply auxiliary functions as signatures involving big-O notation as opposed to implementations. Second, our synthesis algorithm ensures complexity through the tracking of recursive calls, which are not present in the concrete semantics given above. To address these challenges we introduce an intermediate semantics that uses recurrence relations and big-O notation. We then show in Thm. 4.11 that this intermediate semantics approximates complexity in the sense of Defns. 4.3 and 4.4.

The signatures of auxiliary functions g are of the form $\langle \tau_1 \to \{B \mid \varphi(v, \overline{y})\}, O(\psi(u))\rangle$. Although we don't really have the implementation of g, we assume that there exists some implementation $\texttt{fix}\ g.\lambda\overline{y}.t$ of g, such that

$$\frac{}{\langle \texttt{tick}(c, t), C\rangle \hookrightarrow \langle t, C + c\rangle}\ \textsc{Sem-Tick}$$

$$\frac{\overline{v} \text{ are values}}{\langle (\texttt{fix}\ f.\lambda\overline{x}.t)\overline{v}, C\rangle \hookrightarrow \langle [(\texttt{fix}\ f.\lambda\overline{x}.t)/f][\overline{v}/\overline{x}]t, C\rangle}\ \textsc{Sem-App}$$

$$\frac{\langle t_1, 0\rangle \hookrightarrow \langle t_2, C_\Delta\rangle}{\langle (\texttt{fix}\ f.\lambda\overline{x}.t)t_1, C\rangle \hookrightarrow \langle (\texttt{fix}\ f.\lambda\overline{x}.t)t_2, C + C_\Delta\rangle}\ \textsc{Sem-App-Arg}$$

$$\frac{}{\langle \texttt{if true then } t_1 \texttt{ else } t_2, C\rangle \hookrightarrow \langle t_1, C\rangle}\ \textsc{Sem-Cond-True}$$

$$\frac{}{\langle \texttt{if False then } t_1 \texttt{ else } t_2, C\rangle \hookrightarrow \langle t_2, C\rangle}\ \textsc{Sem-Cond-False}$$

$$\frac{\langle t_c, 0\rangle \hookrightarrow^* \langle b, C_\Delta\rangle \quad b \text{ is a Boolean value}}{\langle \texttt{if } t_c \texttt{ then } t_1 \texttt{ else } t_2, C\rangle \hookrightarrow \langle \texttt{if } b \texttt{ then } t_1 \texttt{ else } t_2, C + C_\Delta\rangle}\ \textsc{Sem-Cond-Guard}$$

$$\frac{\overline{v} \text{ are values}}{\langle \texttt{match } C_j(\overline{v}) \texttt{ with } |_i C_i(\overline{x}_i) \mapsto t_i, C\rangle \hookrightarrow \langle [\overline{v}/\overline{x}_i]t_i, C\rangle}\ \textsc{Sem-Match}$$

$$\textsc{Sem-Match-Scrutinee}\quad \frac{\langle t_s, 0\rangle \hookrightarrow^* \langle C_j(\overline{v}), C_\Delta\rangle \quad \overline{v} \text{ are values}}{\langle \texttt{match } t_s \texttt{ with } |_i C_i(\overline{x}_i) \mapsto t_i, C\rangle \hookrightarrow \langle \texttt{match } C_j(\overline{v}) \texttt{ with } |_i C_i(\overline{x}_i) \mapsto t_i, C + C_\Delta\rangle}$$

Figure 4.6: Evaluation rules of the concrete small-step semantics.

- for any input $\overline{x}$, the output of g on $\overline{x}$ satisfies the signature, i.e., $\langle (\texttt{fix}\ g.\lambda\overline{y}.t)\overline{x}, 0\rangle \hookrightarrow^* \langle v_{\overline{x}}, C_{\overline{x}}\rangle$ implies $\varphi(v_{\overline{x}}, \overline{x})$; and
- for any input $\overline{x}$, the complexity of g is bounded by $\psi(u)$, i.e., $T_g(n) \in O(\psi(n))$.

For the top-level function we are evaluating, we assume that its signature $\langle \tau_1 \to \{B \mid \varphi(v, \overline{y})\}, O(\psi(u))\rangle$ is also given, whereas the semantics of f is over-approximated by its refinement, i.e., for any input $\overline{x}$, $\langle (\texttt{fix}\ f.\lambda\overline{y}.t)\overline{x}, 0\rangle \hookrightarrow^* \langle v_{\overline{x}}, C_{\overline{x}}\rangle$ implies $\varphi(v_{\overline{x}}, \overline{x})$.

Now we introduce our intermediate loose semantics. Formally, reductions are defined between configurations. Each configuration $\langle \hat{t}, \mathcal{R} \rangle^{\#}$ is a pair of an extended term $\hat{t}$ and a recurrence parameter $\mathcal{R}$. Extended terms $\hat{t} ::= t \mid \varphi$ are either terms t or formula expressions φ. Recurrence parameters $\mathcal{R} ::= \phi \mid \bot \mid \mathcal{R} \parallel \mathcal{R}$ are either size expressions ϕ, or parameters combined by a *parallel* operator $\parallel$, i.e., $\mathcal{R}$ is a collection of size expressions. The parallel operator $\parallel$ distributes over the plus, i.e., $(\mathcal{R}_1 \parallel \mathcal{R}_2) + \mathcal{R}_3 = \mathcal{R}_1 + \mathcal{R}_2 \parallel \mathcal{R}_1 + \mathcal{R}_3$. Intuitively, a parameter $\mathcal{R}$ without parallelism denotes the recurrence relation of the function along one path. When the function contains more than one path, the overall parameter will be sub-parameters in parallel.

We use $\langle t, \mathcal{R} \rangle^{\#} \mapsto \langle \varphi, \mathcal{R}' \rangle^{\#}$ to denote a step of a reduction. The goal is to reduce a term t in a function f to a predicate φ such that φ describes the behavior of t—φ is a refinement of t. At the same time, recurrence relations can be built by incrementally appending expressions representing resource usage to the recurrence parameter $\mathcal{R}$.

Because we are building recurrence relations for a function f, the reduction always starts from a **fix**-term and an empty parameter $\bot$. The result configuration is the refinement φ of the function body t with the recurrence parameter $\mathcal{R}$. We use the function $T : \texttt{Int} \to \texttt{Int}$ to denote the resource usage of the function f; hence, the recurrence parameter we build for f will be the recurrence relation of resource usage T.

$$\frac{\langle t, 0 \rangle^{\#} \mapsto \langle \varphi, \mathcal{R} \rangle^{\#}}{\langle \textbf{fix } f.\lambda x_1..\lambda x_n.t, \ \bot \rangle^{\#} \mapsto \langle \varphi, \ \mathcal{R} \rangle^{\#}} \ \textsc{LooseSem-Fix}$$

In our loose semantics, each auxiliary function g has a resource annotation $(O(\psi_g))$ denoting the resource usage of g, a logical signature φ_g denoting the behavior of g, and a size function $\texttt{size}_g$. Resource usage happens when an auxiliary

function is called.

$$\frac{\begin{array}{c} g : \langle x_1 : \tau_1 \to .. \to x_n : \tau_n \to \{B \mid \varphi_g\}, O(\psi_g)\rangle \\ \forall i. \langle e_i, 0\rangle^{\#} \mapsto \langle \varphi_i, \mathcal{R}_i\rangle^{\#} \qquad \varphi := \varphi_g \wedge \bigwedge_{i=1}^{n} [x_i/v]\varphi_i \quad \forall i. \varphi_i \sqsubset \nu_i \end{array}}{\langle g\ e_1 ..\ e_n, 0\rangle^{\#} \mapsto \langle \varphi,\ [(\mathtt{size}_g\ \nu_1 ..\ \nu_n)/u]\psi_g + \sum_{i=1}^{n} \mathcal{R}_i\rangle^{\#}}\ \textsc{LooseSem-App}$$

That is, if each callee e_i can be reduced to a predicate φ_i bounded by ν_i with the recurrence parameter change $\mathcal{R}_i$, the non-recursive application term $g\ e_1 ..\ e_n$ will be reduced to the predicate φ with resource usage $[\mathtt{size}_g\ \nu_1 ..\ \nu_n/u]\psi_g$ (the resource usage of g) and $\sum_{i=1}^{n} \mathcal{R}_i$ (resource usage used to evaluate $\{e_i\}_i$). We over-approximate the size of the problem $e_1..e_n$ using the upper bounds ν_i of callee's behavior predicates φ_i. The result predicate φ is actually the behavior φ_g of g with each argument x_i instantiated with the semantic predicates $[x_i/v]\varphi_i$ of callee e_i.

The semantics of performing a recursive call is a bit different. The resource usage is instead $T(\mathtt{size}_f\ \nu_1 ..\ \nu_n)$—the resource usage T on a sub-problem with size $\mathtt{size}_f\ \nu_1 ..\ \nu_n$ where the ν_i's are over-approximations of the callee e_i's.

$$\frac{\begin{array}{c} \forall i. \langle e_i, 0\rangle^{\#} \mapsto^{\langle}\ \varphi_i, \mathcal{R}_i\rangle^{\#} \qquad \forall i. \varphi_i \sqsubset \nu_i \\ f : \langle x_1 : \tau_1 \to .. \to x_n : \tau_n \to \{B \mid \varphi_f\}, O(\psi_f)\rangle \qquad \varphi := \varphi_f \wedge \bigwedge_{i=1}^{n} [x_i/v]\varphi_i \end{array}}{\langle f\ e_1 ..\ e_n, 0\rangle^{\#} \mapsto \langle \varphi,\ T(\mathtt{size}_f\ \nu_1 ..\ \nu_n) + \sum_{i=1}^{n} \mathcal{R}_i\rangle^{\#}}\ \textsc{LooseSem-RecApp}$$

The reduction of `if`-terms will result in `ite` predicates. The resulting recurrence parameter $\mathcal{R}_e + \mathcal{R}_1 \parallel \mathcal{R}_e + \mathcal{R}_2$ uses the parallel operator because there are two paths in an `ite` term.

$$\frac{\langle e, 0\rangle^{\#} \mapsto \langle \varphi_e, \mathcal{R}_e\rangle^{\#} \quad \langle t_1, 0\rangle^{\#} \mapsto \langle \varphi_1, \mathcal{R}_1\rangle^{\#} \quad \langle t_2, 0\rangle^{\#} \mapsto \langle \varphi_2, \mathcal{R}_2\rangle^{\#} \qquad \varphi := \varphi_1 \vee \varphi_2}{\langle \mathtt{if}\ e\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2, 0\rangle^{\#} \mapsto \langle \varphi,\ \mathcal{R}_e + \mathcal{R}_1 \parallel \mathcal{R}_e + \mathcal{R}_2\rangle^{\#}}\ \textsc{LooseSem-Cond}$$

The rules for match term and variable term are similar.

$$\textsc{LooseSem-Match}\quad \frac{\forall i.\ \langle t_i, 0\rangle^{\#} \mapsto \langle \varphi_i, \mathcal{R}_i\rangle^{\#} \quad \varphi := \varphi_1 \vee \ldots \vee \varphi_m}{\langle \texttt{match}\ e\ \texttt{with}\ |_i\ C_i\ (x_i^1 \ldots x_i^n) \mapsto t_i, 0\rangle^{\#} \mapsto \langle \varphi,\ \mathcal{R}_e + \mathcal{R}_1 \parallel \ldots \parallel \mathcal{R}_e + \mathcal{R}_m\rangle^{\#}}$$

$$\frac{}{\langle x, 0\rangle^{\#} \mapsto \langle |v| = |x|,\ 0\rangle^{\#}}\ \textsc{LooseSem-Var}$$

With the above rules, we can then say the complexity of a function to be any expression that satisfy the recurrence parameter of the function.

Theorem 4.11 (Complexity bounds)**.** *Given a function term* $\texttt{fix}\ f.\lambda\overline{x}.t_f$*, the signature type of* f*, and the signature types of all auxiliary function used in* f*, if*

- *the refinements of auxiliary functions and* f *are all bounded by some monotonic non-decreasing polynomials;*
- *the function body* t_f *can be reduced to* $\langle \cdot, \mathcal{R}_f\rangle^{\#}$*, where* $\mathcal{R}_f$ *is of form* $\mathcal{R}_{f,1} \parallel .. \parallel \mathcal{R}_{f,m}$*, and none of the* $\mathcal{R}_{f,i}$*'s contain an occurrence of the parallel operator; and*
- *there exists a function* ψ *that satisfies*

$$\forall i.\ T(size_f(\overline{x})) \leqslant \mathcal{R}_{f,i} \implies T(size_f(\overline{x})) \in O(\psi(size_f(\overline{x}))),$$

then the complexity T_f *of* f *is bounded by the function* ψ*, i.e.,* $T_f \in O(\psi)$*.*

Proof. We first show by induction on a loose semantic derivation that for any term t, if

- $\langle t, 0\rangle^{\#} \mapsto \langle \varphi_t, \mathcal{R}\rangle^{\#}$, where $\mathcal{R}$ is of form $\mathcal{R}_1 \parallel .. \parallel \mathcal{R}_l$, and none of the $\mathcal{R}_i$'s contains an occurrence of the parallel operator; and
- $\langle [(\texttt{fix}\ f.\lambda\overline{x}.t_f/f][(\texttt{fix}\ g.\lambda\overline{y}.t_g/g][\overline{in}/\overline{x}]t, 0\rangle \hookrightarrow^* \langle v_t, C_t\rangle$ for all $\overline{in}$,

then we have for any $\overline{in}$, $[v_t/v][\overline{in}/\overline{x}]\varphi_t$ is satisfiable—that is, the loose semantics φ_t over-approximate the concrete semantics v_t—, and $\exists c, k > 0\, \forall \overline{in}\, \exists j.\ \mathtt{size}_f(\overline{in}) > c \implies C_t < k * \mathcal{R}_j$.

Base case. The base case is variable term. The φ for variable term is precise and the resource usage are 0 in both semantics.

Non-recursive application terms. When $t = g\ e_1 .. e_n$ is an non-recursive application term, with the induction hypothesis, we have that all e_i's loose semantics φ_i over-approximates their concrete semantics v_{e_i}, i.e., $[v_{e_i}/v][\overline{in}/\overline{x}]\varphi_{e_i}$ is satisfiable for all e_i and $\overline{in}$. According to the first assumption of the signature of auxiliary functions, the predicate $\forall \overline{in}.\ \varphi_g(v_t, \overline{v}_{e_i})$ is valid. The loose semantic of t is $\varphi_t := \varphi_g(v, \overline{y}) \wedge_{i=1}^{n} [y_i/v]\varphi_{e_i}$. Finally, the predicate

$$\forall \overline{in}.[v_t/v][\overline{in}/\overline{x}]\varphi_t = \varphi_g(v_t, \overline{y}) \wedge_{i=1}^{n} [y_i/v][\overline{in}/\overline{x}]\varphi_{e_i}$$

is satisfiable with the assignment $y_i \leftarrow v_{e_i}$ for all i.

According to the second assumption on signatures of auxiliary functions, we have $T_g(n) \in O(\psi_g(n))$. That is, $\exists \overline{c}, k\ \forall \overline{in}.\ \mathtt{size}_g(\overline{in}) > \overline{c} \implies C_t \leqslant k * \psi_g(\mathtt{size}_g(\overline{|v_{e_i}|}))$.

Then we have $\exists \overline{c}, k \forall \overline{in}.\ \mathtt{size}_g(\overline{in}) > \overline{c} \implies C_t \leqslant k * \psi_g((\mathtt{size}_g\ v_1 .. v_n))$ because each v_i is the bound of φ_i and hence the bound of the concrete value v_{e_i}.

Recursive application terms. When $t = f\ e_1 .. e_n$ is a recursive application term, the proof of the behavior part is similar as above because we have the same behavior assumption on the signature of the top-level function. The concrete cost C_t of t is bounded by $T_f(\mathtt{size}_f(|v_{e_1}|, \ldots, |v_{e_n}|))$, where T_f is the complexity function of f, plus the concrete cost of evaluating each e_i (which is bounded by $\sum \mathcal{R}_i$ according to the induction hypothesis). Note that here T_f is an uninterpreted monotonic non-decreasing non-negative function. The loose semantics $T(\mathtt{size}_f(\ v_1 .. v_n))$ also contains an uninterpreted monotonic non-decreasing non-negative function T. In this proof, we generalize the comparison symbol $\leqslant$ to $T_f(n) \leqslant T(n')$ if $n \leqslant n'$, that is, the comparison between uninterpreted functions is the result of comparison between their inputs. With such generalization,

$T_f(\mathtt{size}_f(|v_{e_1}|,\ldots,|v_{e_n}|)) \leqslant T(\mathtt{size}_f(\, v_1..\, v_n))$ because each v_{e_i} is bounded by v_i according to the induction hypothesis.

Branching terms. When $t = \mathtt{if}\ e\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2$ is a conditional term, the concrete cost $C_{t_1} + C_e$ or $C_{t_1} + C_e$ of it is bounded by the loose cost $\mathcal{R}_e + \mathcal{R}_1$ or $\mathcal{R}_e + \mathcal{R}_2$, respectively, according to the induction hypothesis. The concrete semantics is either v_{t_1} or v_{t_2}. According to the induction hypothesis, $\varphi_{t_1} \vee \varphi_{t_2}$ over-approximates both branches.

The case of match term is similar to the conditional term.

Now, for any input $\overline{in}$, the complexity function T_f of the top-level function f should satisfy the recurrence parameter along one of the path, i.e.,

$$\exists k \forall \overline{in}\ \exists j. T_f(\mathtt{szie}_f(\overline{in})) \leqslant k * [T_f/T]\mathcal{R}_{f,j}.$$

So, if a bound ψ dominating the loose cost for every path, it will always dominate the complexity T_f of f.

□

Example 4.12. *For the program shown in Eqn. (4.2), there are three paths. At the beginning,* $\mathtt{fix}\ prod.\lambda x.\lambda y.t$ *is reduced to* $\langle \varphi, T(size_{prod}\, x\, y) \leqslant 0 + \mathcal{R}\rangle^{\#}$ *where* $\mathcal{R}$ *is the reduction result of the function body, and* $size_{prod}\, x\, y = x$ *since the size function for* $prod$ *is* $\lambda z.\lambda w.z$

The first ite *term* $\mathtt{if}\ x == 0\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2$ *is reduced to the configuration* $\langle ite(x == 0,\ \varphi_1,\ \varphi_2), \mathcal{R}_e + \mathcal{R}_1 \parallel \mathcal{R}_e + \mathcal{R}_2\rangle^{\#}$ *where the condition contain one equivalence operator* $(\mathcal{R}_e = 1)$*, the* $\mathtt{then}$ *branch has 0 resource usage (it is a variable term)* $(\mathcal{R}_1 = 0)$*, and the* $\mathtt{else}$ *branch has resource usage* $\mathcal{R}_\in$*, which we learn from the reduction of* t_2*.*

The application term $div2\ x$ *is reduced to* $\langle v = \frac{z}{2} \wedge z = x,\ 1\rangle^{\#}$ *since the resource usage of* $div2$ *is* $O(1)$*.*

The recursive-application term $prod\ (div2\ x)\ y$ *is reduced to* $\langle v = z * w \wedge z = \frac{x}{2} \wedge w = y,\ \mathcal{R}_0 + T(\frac{x}{2}) + 1\rangle^{\#}$

Overall the recurrence relation can be built as $T(x) \leqslant 1 \parallel T(x) \leqslant 4 + T(\frac{x}{2}) \parallel T(x) \leqslant 5 + T(\frac{x}{2})$*. Thus the complexity of* $prod$ *is bounded by* $\log x$ *since* $T(x) \leqslant 1 \implies T(x) \in O(\log x)$*, and,* $T(x) \leqslant 5 + T(\frac{x}{2}) \implies T(x) \in O(\log x)$ *according to the Master Theorem.*

4.5 The SynPlexity Synthesis Algorithm

In this section, we present the SynPlexity synthesis algorithm, which uses annotated types to guide the search of terms of given types.

Overview of the Synthesis Algorithm

The algorithm takes as input a goal type $\texttt{f} : \langle \tau, O(\psi) \rangle$, an environment Γ that includes type information of auxiliary functions, and the `size` functions for `f` and all auxiliary functions. The goal is to find a function term of type $\langle \tau, O(\psi) \rangle$.

The algorithm uses the rules of the SynPlexity type system to decompose goal types into sub-goals, and then applies itself recursively on the sub-goals to synthesize sub-terms. Concretely, given a goal γ, the algorithm tries all the rules shown in §4.3, where the type in the conclusion matches γ, to construct sub-goals: for each sub-term t in the conclusion, there must be a judgment $\Gamma \vdash t :: \gamma'$ in the premise; thus, we construct the sub-goal γ'—the desired type of t. For each I-term rule, the type of each sub-term is always known, and thus a fixed set of sub-goals is generated. For each E-term rule (Fig. 4.5), the algorithm enumerates E-terms up to a certain depth (the depth can be given as a parameter or it can automatically increase throughout the search). If the algorithm fails to solve some sub-goal using some E-term rule, it backtracks to an earlier choice point, and tries another rule.

Because the top-level goal is always a function type, the algorithm always starts by applying the rule T-Abs, which matches the resource bound $O(\psi)$ using Tab. 4.1 to infer a possible recurrence annotation for the type of the function body. Also T-Abs constructs a sub-goal type for the function body. In the rest of this section, we assume that goals are not function types.

Algorithm 2 GenerateE(Γ, γ, d)

Input : Context Γ, goal type $\gamma = \langle \{B \mid \varphi\}, \alpha \rangle$, depth bound d
for $t \leftarrow$ *EnumerateE*(Γ, d, B) **do**
  if *CheckE*(t, Γ, γ) **then return** t
end
return $\bot$

Variables. Given a goal γ, the algorithm first tries to apply rule E-Var, which simply checks if any variable in the environment is of type γ, and hence is a solution. If no variable in the environment could be a solution, it starts enumerating E-terms up to a certain depth.

Synthesizing Application Terms. To enumerate application terms, the algorithm first enumerate a term t that satisfies the base types B in the goal annotated type $\langle\{B \mid \varphi\}, ([c_1, \phi_1]..[c_n, \phi_n]; O(\psi))\rangle$. Then, the algorithm checks if the total number of recursive calls in the term t exceeds the bound $\sum_i c_i$. If yes, the term t is rejected. Otherwise, the sizes of sub-problems of recursive calls are checked. Formally, to check if a recursive application term $f(t_1, .., t_m)$ is consistent with some $[c_k, \phi_k]$, the algorithm queries the validity of the following predicate

$$(\bigwedge_{i=1}^{m} [y_i/v]\varphi_i \Rightarrow (\texttt{size}_f(y_1 \,..\, y_m) = [\texttt{size}_f(\Gamma(\texttt{args}))/v]\phi_k)),$$

where the y_i's are fresh variables, and the φ_i's are the refinements of the t_i terms. If the sizes of sub-problems are not consistent with the recursive-call costs $[c_1, \phi_1]..[c_n, \phi_n]$, the term t is rejected. Note that one recursive call can possibly satisfy more than one $[c_k, \phi_k]$. The algorithm will enumerate all possible matches.

Checking the validity of auxiliary application terms is similar. The following predicate is checked, which states that the resource usage of an auxiliary function call should not exceed the bound $O(\psi)$.

$$\bigwedge_{i=1}^{m} [y_i/v]\varphi_i \Rightarrow \left([\texttt{size}_g\ y_1..y_m/v]\psi_g \in O([\texttt{size}\ \Gamma(\texttt{args})/v]\psi)\right).$$

Recall that the above query is undecidable in general, and is checked using an SMT solver in SynPlexity. An enumerated term is accepted if its refinement implies the goal refinement φ.

Rules for Branching Term. When the algorithm chooses to apply the rule T-If to synthesize a term of the form `if` e `then` t_1 `else` t_2 for a given goal $\langle\{B \mid \varphi\}, \alpha\rangle$, there are three steps to construct sub-goals for sub-terms e, t_1, and t_2: (1) sharing the

$$\texttt{prod} = ??_1 : \langle \texttt{x}{:}\{\texttt{Int} \mid v \geqslant 0\} \to \texttt{y}{:}\{\texttt{Int} \mid v \geqslant 0\} \to \{\texttt{Int} \mid v = \texttt{x} * \texttt{y}\}, (O(\log u)) \rangle$$

$$\xrightarrow{\text{T-Abs}} \quad \texttt{prod} = \lambda \texttt{x}.\lambda \texttt{y}.??_2 : \langle \{\texttt{Int} \mid v = \texttt{x} * \texttt{y}, ([1, \lfloor \tfrac{u}{2} \rfloor]; O(1)) \rangle$$

$$\xrightarrow{\text{T-If}} \quad \texttt{prod} = \lambda \texttt{x}.\lambda \texttt{y}.\texttt{if}\ ??_3 \quad \xleftarrow{\text{E-App}}\ \texttt{x==0} : \langle \{\texttt{Bool} \mid \texttt{x} = 0\}, ([0, \lfloor \tfrac{u}{2} \rfloor]; O(1)) \rangle$$
$$\texttt{then}\ ??_4 : \langle \{\texttt{Int} \mid v = \texttt{x} * \texttt{y} \wedge \texttt{x} = 0, ([1, \lfloor \tfrac{u}{2} \rfloor]; O(1)) \rangle$$
$$\xleftarrow{\text{E-App}}\ \texttt{x} : \langle \{\texttt{Bool} \mid v = 0\}, ([0, \lfloor \tfrac{u}{2} \rfloor]; O(1)) \rangle$$
$$\texttt{else}\ ??_5 : \langle \{\texttt{Int} \mid v = \texttt{x} * \texttt{y} \wedge \texttt{x} > 0, ([1, \lfloor \tfrac{u}{2} \rfloor]; O(1)) \rangle$$

$$??_5 \xrightarrow{\text{T-If}} \texttt{if}\ ??_6 \quad \xleftarrow{\text{E-App}}\ \texttt{even x} : \langle \{\texttt{Bool} \mid \texttt{x mod 2} = 0\}, ([0, \lfloor \tfrac{u}{2} \rfloor]; O(1)) \rangle$$
$$\texttt{then}\ ??_7 : \langle \{\texttt{Int} \mid v = \texttt{x} * \texttt{y} \wedge \texttt{x mod 2} = 0, ([1, \lfloor \tfrac{u}{2} \rfloor]; O(1)) \rangle$$
$$\xleftarrow{\text{E-App}}\ \texttt{double (prod (div2 x) y)}$$
$$\texttt{else}\ ??_9 : \langle \{\texttt{Int} \mid v = \texttt{x} * \texttt{y} \wedge \texttt{x mod 2} = 1, ([1, \lfloor \tfrac{u}{2} \rfloor]; O(1)) \rangle$$
$$\xleftarrow{\text{E-App}}\ \texttt{plus y (double (prod (div2 x) y))}$$

Figure 4.7: Trace of the synthesis of an $O(\log x)$ implementation of `prod`.

recursive-call costs in α, (2) enumerating the condition guard e, and (3) propagating sub-goals to the two branches t_1 and t_2. Note that there can be multiple ways to share α to α_1 and α_2, and the algorithm will try them one by one (because the numbers of recursive calls are natural numbers, SynPlexity will just enumerate all possible ways to split them). Once a sharing α_1, α_2 is chosen, the algorithm constructs a goal $\langle \texttt{Bool}, \alpha_1 \rangle$ for the condition guard e. For a candidate e of type $\langle \{\texttt{Bool} \mid \varphi_e\}, \alpha_1 \rangle$, the algorithm constructs two sub-goals $\langle \{B \mid \varphi\}, \alpha_2 \rangle$ along with the path condition φ_e, and $\langle \{B \mid \varphi\}, \alpha_2 \rangle$ along with the path condition $\neg \varphi_e$ for the two branches t_1 and t_2, respectively. Applying the rule T-Match is similar to T-If.

Example 4.13. *We illustrate in Fig. 4.7 how the algorithm synthesizes the* $O(\log x)$ *implementation of* `prod` *presented in Eqn. (4.2). We omit the type contexts in the example. We will use "*`??`*" to denote intermediate terms being synthesized (i.e., holes in the program). At the beginning, the type of* $??_1$ *(i.e., the term we are synthesizing) is an arrow type with resource bound* $O(\log u)$ *specified by the input goal. In this example,* SynPlexity *applies to the arrow type the rule T-Abs, parameterized according to the first rule in Tab. 4.1. This*

step produces the sub-problem of synthesizing the function body $??_2$*, whose annotation is* $([1, \lfloor \frac{u}{2} \rfloor]; O(1))$*—which means that* $??_2$ *should contain at most one recursive call to sub-problems with size* $\lfloor \frac{u}{2} \rfloor$*.*

Next, SynPlexity *chooses to fill* $??_2$ *with an* `if-then-else` *term (by applying the T-If rules) with three sub-problems: the condition guard* $??_3$*, the* `then` *branch* $??_4$ *and the* `else` *branch* $??_5$*. Note that here we share the number of recursive calls* $[1, \frac{u}{2}]$ *as follows: 0 recursive calls in the condition guard, and 1 in the* `then` *branch and the* `else` *branch. The left arrow* `E-App` *shows how* SynPlexity *enumerates terms and checks them against the goal types of sub-problems. For example, to fill* $??_4$*,* SynPlexity *enumerates terms of type* $\langle \{\texttt{Int} \mid \nu = x * y \wedge x = 0\}, ([1, \frac{u}{2}]; O(1)) \rangle$*, which are restricted to contain at most one recursive call to* `prod`*. In Fig. 4.7,* SynPlexity *has picked the term* `x` *to fill* $??_4$*. The refinement type of the variable term* `x` *is* $\{\texttt{Int} \mid \nu = x \wedge x = 0\}$ *where* $x = 0$ *is the path condition. To check that* `x` *also satisfies the type of* $??_4$*, the algorithm needs to apply rule E-SubType, and check that, for any* ν *and* `x`*,* $\nu = x \wedge x = 0$ *implies* $\nu = x * y \wedge x = 0$*, and* $[0, \lfloor \frac{u}{2} \rfloor]$ *is approximated by* $[1, \lfloor \frac{u}{2} \rfloor]$*.*

After applying another T-If rule for $??_5$*,* SynPlexity *produces three new sub-problems* $??_6$*,* $??_7$*, and* $??_8$*. When enumerating terms to fill* $??_7$*,* SynPlexity *finds an application term* `double (prod (div2 x) y)` *that satisfies the goal* $\langle \{\texttt{Int} \mid \nu = x * y \wedge x \ \texttt{mod}\ 2 = 0\}, ([1, \lfloor \frac{u}{2} \rfloor]; O(1)) \rangle$*. To check that the size of the problem in the recursive call* `prod (div2 x) y` *satisfies the recursive-call cost* $[1, \lfloor \frac{u}{2} \rfloor]$*, the type system first checks the refinement of the callee. The refinement of the first argument* `(div2 x)` *is* $\varphi_1 := \nu = \lfloor \frac{x}{2} \rfloor$*. The refinement of the second argument* `y` *is* $\varphi_2 := \nu = y$*. Consequently, the size of the sub-problem* `prod (div2 x) y` *satisfies* $[1, \lfloor \frac{u}{2} \rfloor]$ *because* $[z/\nu]\varphi_1 \wedge [w/\nu]\varphi_2 \implies \texttt{size}\ z\ w = [(\texttt{size}\ x\ y)/\nu]\lfloor \frac{u}{2} \rfloor$*, which can be simplified to* $z = \lfloor \frac{x}{2} \rfloor \wedge w = y \implies z = \lfloor \frac{x}{2} \rfloor$*. (Recall that the size function for* `prod` *is* `size` $:= \lambda z.\lambda w.z$*.)*

Optimization. Algorithm 3 discussed above is based on bidirectional type-guided synthesis with liquid types (Synquid [PKSL16]). Therefore, *liquid abduction* and *match abduction*, two optimizations used in Synquid, can also be used in SynPlexity. These two techniques allow one to synthesize the branches of `if`- and `match`-terms, and then use logical abduction to infer the weakest assumption under which the

Algorithm 3 GenerateI(Γ, γ, d, m).

Input : Context Γ, goal type γ, depth bound d, match bound m
if $t \leftarrow$ *GenerateE*(Γ, γ, d) **then return** t
if $m > 0$ **then for** $s \leftarrow$ *EnumerateE*($\Gamma, d,$ `dataType`) **do**
  `patterns` $\leftarrow$ GeneratePatterns(Γ, TypeOf(s))
  $\gamma' \leftarrow$ UpdateCost(s, γ)
  for $i \in [1,$ *Size*(`patterns`)$]$ **do**
    $t_i \leftarrow$ GenerateI(UpdateContext($\Gamma, s ==$ `patterns`$[i]), \gamma', d, m-1$)
    if $t_i == \bot$ **then return** $\bot$
  end
  return `Match` s `with` $|_i$ `patterns`$[i] \rightarrow t_i$
end

for `cond` $\leftarrow$ *EnumerateE*($\Gamma, d,$ *Bool*) **do**
  $\gamma' \leftarrow$ UpdateCost(s, γ)
  $t_T \leftarrow$ GenerateI(UpdateContext($\Gamma,$ `cond`$), \gamma', d, m$)
  $t_F \leftarrow$ GenerateI(UpdateContext($\Gamma, \neg$`cond`$), \gamma', d, m$)
  if $t_T \neq \bot \wedge t_F \neq \bot$ **then return** `If cond then` t_T `else` t_F
end
return $\bot$

branch fulfills the goal type.

The algorithm is sound because it only enumerates well-typed terms.

Theorem 4.14 (Soundness of the synthesis algorithm)**.** *Given a goal type* $\langle \tau, O(\psi) \rangle$ *and an environment* Γ*, if a term* `fix` $f.\lambda x_1..\lambda x_n.t$ *is synthesized by* SynPlexity*, then the complexity of* f *is bounded by* ψ*.*

4.6 Extensions to the SynPlexity Type System

In this section, we introduce two extensions to the SynPlexity type system.

Recurrence Relations with Correlated Sizes

The type system shown in §4.3 only tracks sub-problems with independent sizes. For example, consider the recurrence relation $T(u) = T(l) + T(r) + O(1)$, where

the variables $\mathtt{l}$ and $\mathtt{r}$ are correlated by the constraint $\mathtt{l} + \mathtt{r} < u$. This relation is needed to reason about programs that manipulate binary trees or binary heaps, where $\mathtt{l}$ and $\mathtt{r}$ represent the sizes of the two children. To support such a recurrence relation, we extend SynPlexity's type system with recursive-call costs of the form $[1, \mathtt{l}], [1, u - 1 - \mathtt{l}]$, where $\mathtt{l}$ is a free variable. When correlated recurrence relations are present, the synthesis algorithm will: (1) match the first enumerated recursive-call term to $[1, \mathtt{l}]$, and instantiate the size $\mathtt{l}$ with s, where s is the size of the recursive-call term (s should be smaller than the size u of the top-level function); and (2) use the size s of the recursive-call term computed in step 1 to constrain the algorithm to enumerate only recursive-call terms of sizes at most $u - 1 - s$.

Synthesis of Auxiliary Functions

Most of the existing type-directed approaches require the input to the problem to contain all needed auxiliary functions. With SynPlexity, some of the auxiliary functions needed to solve synthesis problems with resource annotations can be synthesized automatically.

For example, consider the problem `prod` described in §4.2. In this problem, we observe that one of the provided auxiliary functions, `div2`, strongly resembles one of the elements of the recurrence relation, $T(u) \leqslant T(\lfloor \frac{u}{2} \rfloor) + O(1)$, needed to synthesize a program with the desired resource usage. In particular, we know that one needs an auxiliary function that can take an input of size $\mathtt{u}$ and produce an output of size $\lfloor \frac{u}{2} \rfloor$. In this example, the required auxiliary function `div2` merely needs to divide the input by 2 (and round down), but in certain cases it might need a more precise refinement type than merely changing the size of the input. For example, the auxiliary function split used by merge sort needs to split the input list `xs` into two lists `v1` and `v2` that are half the length of the input *and* such that $\mathtt{elems(v1)} \uplus \mathtt{elems(v2)} = \mathtt{elems(xs)}$. However, all we know from the refinement is that the output lists must be half the length of the original list.

Although we do not know what this auxiliary function should do exactly, we can use the size constraint appearing in the recurrence relation to define part of

the refinement type we want the auxiliary function to satisfy. SynPlexity builds on this idea and incorporates an (optionally enabled) algorithm, SynAuxRef, that while trying to synthesize a solution to the top-level synthesis problem also tries in parallel to synthesize auxiliary functions that can create sub-problems with the size constraints needed in the recurrence relation. To address the problem mentioned above—i.e., that we do not know the exact refinement type the auxiliary function should satisfy—SynAuxRef enumerates *auxiliary refinements*, which are possible specifications that the auxiliary function `aux` we are trying to synthesize might satisfy.

SynPlexity with Higher-Order Functions

Recall that the type system of SynPlexity shown in §4.3 does not support higher-order functions. That is, no argument of a function is allowed to be of an arrow type. Inferring the resource usages of higher-order functions is challenging for two reasons. First, the resource usages of function arguments g can be unknown, in which case the resource usages of applications of g will also be unknown. Second, the behavior of function arguments g can be unbounded. Hence, the resource usages of nested applications $f(g(...), ...)$ can also be unbounded when the resource usage of f grows along with its arguments.

For example, the following program is a higher-order function. It takes as input a function argument g and an integer-list argument xs, and constructs a new list as output by applying an auxiliary function `square` and the function argument g to each element in xs. When the resource usage of g is unknown, the resource usage of the application $(g\ x)$ is also unknown. Also, suppose we assume that the resource usage of `square` is linear in its argument. In that case, the resource usage of the application `square` $(g\ x)$ is unbounded because the value of its argument

(g x) is unbounded.

```
map_square = λg.λxs. match   xs   with
                       Nil   →   Nil
                  Cons x xt  →   Cons (square (g x)) (map_square g xt)
```

Although SynPlexity does not support higher-order functions in general, we can extend it to support programs with higher-order functions in practice by introducing four restrictions on target programs. First, we assume that the resource usage of each function argument g is a constant, i.e., $g : \langle \tau, O(1) \rangle$. Second, function arguments in recursive calls in the synthesized programs are the same as the top-level function's function arguments. For example, in the body of a higher-order function `fix f.λgλxλy.t`, all recursive application terms must be of form f(g, _, _) where each _ can be any well-typed term. Third, we assume that the behavior of function arguments does not affect the asymptotic resource usage of higher-order functions. To satisfy this restriction, we want to avoid nested application terms where the outer functions have non-constant resource usage and the value of arguments of the outer function depends on some function arguments. Finally, function arguments cannot appear in size functions.

In the rest of this section, we first introduce the extensions to SynPlexity's type system and then formally state the restrictions we introduced.

Extended Syntax and Types. The extended syntax of the surface language contains two new rules: 1) an E-term can be a function term, which means that arguments of application terms can be function terms, and 2) a function term can be a lambda term.

$$\text{E-term} \quad e ::= f \qquad \text{Function term} \quad f ::= \lambda x.t$$

The extended type system contains a new kind of arrow type

$$\tau ::= x_1 : \gamma_1 \to \ldots \to x_n : \gamma_n \to y : \tau_y,$$

which extends standard arrow types by allowing arguments to be annotated types.

The idea of the extended arrow types is that arguments can be of function types with annotated resource usage. Recall that with the first restriction, we assume that all recurrence annotations in the higher-order arrow type are $O(1)$. That is, the resource usages of function arguments are always constant.

Restriction on the Synthesis algorithm. With the extended type system, we also modify the synthesis algorithm to prune E-terms that breaks the second or third restriction mentioned above.

To support the second restriction (i.e., that we need to call the same function arguments in recursive calls), the synthesis algorithm first stores the function arguments of the top-level functions. Later, when a recursive call is enumerated, the synthesizer checks whether it calls the same function arguments, and rejects the candidate if it does not.

To support the third restriction (i.e., that the behavior of function arguments should not affect the resource usage), the synthesis algorithm avoids enumerating nested application terms where the resource usage of the outer application depends on the value of an inner application term that calls a function argument.

4.7 Evaluation

In this section, we evaluate the effectiveness and performance of SynPlexity, and compare it to existing tools.[3] We implemented SynPlexity in Haskell on top of Synquid by extending its type system with recurrence annotations as presented in §4.3.

Comparison to Prior Tools

We compared SynPlexity against two related tools: Synquid [PKSL16] and ReSyn [KWPH19], which are also based on refinement types.

[3] All the experiments were performed on an Intel Core i7 4.00GHz CPU, with 8GB of RAM. We used version 4.8.9 of Z3. The timeout for each benchmark was 10 minutes.

Benchmarks. We considered a total of 77 synthesis problems: 56 synthesis problems from ReSyn (each benchmark specifies a concrete linear-time resource annotation), 16 synthesis problems from Synquid (which do not include resource annotations) that are not included in ReSyn, and 5 new synthesis problems involving non-linear resource annotations. In these synthesis problems, synthesis specifications and auxiliary functions are all given as refinement types. For 3 of the new benchmarks, the auxiliary function required to split the input into smaller ones is not given—i.e., the synthesizer needs to identify it automatically.

The three solvers (SynPlexity, Synquid, and ReSyn) have different features, and hence not all synthesis problem can be encoded as synthesis benchmarks for a single solver. In the rest of this section, we describe what benchmarks we considered for each tool, and how we modified the benchmarks when needed.

Synquid*:* Synquid does not support resource bounds, so we encoded 77 synthesis problems as Synquid benchmarks by dropping the resource annotations. Synquid returns the first program that meet the synthesis specification, and cannot provide any guarantees about the resource usage of the returned program. Synquid can solve 75 benchmarks, and takes on average 3.3s. For 10 benchmarks Synquid synthesizes a non-optimal program—i.e., there exists another program with better concrete resource usage. For example, on the ReSyn-triple-2 benchmark (where the input is a list xs), Synquid found a solution with resource usage $O(|xs|^2)$, while both SynPlexity and ReSyn can synthesize a more efficient implementation with resource usage $O(|xs|)$. The two benchmarks that Synquid failed to solve include the new benchmark SynPlexity-merge-sort'. In this benchmark, the auxiliary function required to break the input into smaller inputs is not given, without which the sizes of solutions become much larger. Therefore Synquid times out.

ReSyn*:* We ran ReSyn on the 56 ReSyn benchmarks with the corresponding concrete resource bounds. We could not encode 16 problems because ReSyn does not support non-linear resources bounds—e.g., the bound $\log |y|$ in the AVL-insert Synquid benchmark. ReSyn solved all 56 benchmarks with an average running time of 18.3s.

SynPlexity*:* We manually added resource usages and resource bounds to existing

problems to encode them for SynPlexity. For Synquid benchmarks without concrete resource bounds, we chose well-known time complexities as the bounds, e.g., we added the resource bound $O(u \log u)$ to the Sort-merge-sort problem. For the ReSyn benchmarks, we translated the concrete resource usage and resource bounds to the corresponding asymptotic ones—e.g., for the ReSyn-common' benchmark with the concrete resource bound $|ys| + |zs|$, we constructed a SynPlexity variant with the asymptotic bound $O(u)$ and a size function $\lambda ys.\lambda zs.|ys| + |zs|$. We could not encode 3 synthesis problems as SynPlexity benchmarks: two of them involved higher-order functions that does not satisfy the assumptions introduced in §4.6, and the other one is with exponential resource-usage bound $O(2^u)$ (the Tree-create-balanced problem from Synquid).

SynPlexity solved 73 benchmarks with an average running time of 8.1s. Unlike Synquid, SynPlexity guarantees that the synthesized program satisfies the given resource bounds. For 10 benchmarks, SynPlexity found programs that had better resource usage than those synthesized by Synquid. Furthermore, SynPlexity can encode and solve 9 problems that ReSyn could not solve because the resource bounds involve logarithms. However, SynPlexity cannot encode and solve 3 benchmarks that involve higher-order functions. SynPlexity could solve 3 problems that required synthesizing both the main function (e.g., SynPlexity-merge-sort) and its auxiliary function (e.g., the function splitting a given list into two balanced partitions). No other tool could solve the SynPlexity-merge-sort' benchmark.

Finding. SynPlexity can express and solve 68/77 benchmarks. SynPlexity has comparable performance to existing tools, and can synthesize programs with resource bounds that are not supported by prior tools.

Pruning the Search Space with Annotated Types

SynPlexity uses recurrence annotations to guide the search and avoids enumerating terms that are guaranteed to not match the specified complexity. We compared the numbers of E-terms enumerated by SynPlexity and Synquid for 56 benchmark on

which both tool produced same solutions. Synquid always enumerated at least as many E-terms as SynPlexity, and SynPlexity enumerated strictly fewer E-terms for 26/56 benchmarks. For these 26 benchmarks, SynPlexity can on average prune the search space by 6.2%. For example, in one case (BST-delete) SynPlexity enumerated 2,059 E-terms, while Synquid enumerated 2,202.

Finding. On average, SynPlexity reduces the size of the search space by 6.2% for approximately half of the benchmarks.

4.8 Summary

In this chapter, we introduce the formalization of program-synthesis problems with asymptotic resource usage and a type-guided algorithm of solving the problems. We implemented the algorithm in a tool named SynPlexity. The experiment shows that SynPlexity can synthesize problems with complexity that cannot be expressed by prior work, and guided the search for divide-and-conque solutions.

Table 4.2: Evaluation results of SYNQUID, RESYN, and SYNPLEXITY on benchmarks that can be encoded as SYNPLEXITY benchmarks. T denotes running time. B denotes the given resource bounds. TO denotes a timeout. The benchmarks cannot be encoded by some tools are shown as -. Rec. rel. represents the recurrence-relation pattern SYNPLEXITY chose to use. C=C-finite sequence. M=Master Theorem. A=Akra-Bazzi method. T=the tree recurrence we introduced in §4.6. N=non-recursive.

	Problem	SYNQUID	RESYN		SYNPLEXITY		Rec.
		T(sec)	B	T(sec)	O(B)	T(sec)	rel.
List	is empty	0.64	0	0.65	1	0.64	N
	is member	0.92	$\|xs\|$	0.89	$\|xs\|$	0.84	C
	duplicate each element	0.87	$\|xs\|$	1.63	$\|xs\|$	0.93	C
	replicate	1.02	n	8.31	n	1.30	C
	append two lists	0.94	$\|xs\|$	3.70	$\|xs\|$	1.95	C
	concatenate list of lists	0.93	-	-	$\|xss\|^2$	0.97	C
	take firstn elements	1.03	n	7.75	n	1.31	C
	drop firstn elements	0.89	n	40.82	n	11.51	C
	delete value	0.90	$\|xs\|$	2.04	$\|xs\|$	1.30	C
	zip	0.91	$\|xs\|$	2.44	$\|xs\|$	1.22	C
	i-th element	0.70	$\|xs\|$	1.01	$\|xs\|$	0.97	C
	index of element	0.97	$\|xs\|$	1.76	$\|xs\|$	1.29	C
	insert at end	1.06	$\|xs\|$	1.65	$\|xs\|$	1.04	C
	reverse	1.10	$\|xs\|$	1.49	$\|xs\|$	1.09	C
Unique list	insert	1.01	$\|xs\|$	2.24	$\|xs\|$	2.89	C
	delete	0.81	$\|xs\|$	1.61	$\|xs\|$	2.13	C
	remove duplicates	0.74	-	-	$\|xs\|^2$	3.68	C
	compress	2.62	$\|xs\|$	10.25	$\|xs\|$	7.91	C
	integer range	4.83	$size$	206.19	$size$	7.42	C
Strictly sorted list	insert	1.24	$\|xs\|$	4.92	$\|xs\|$	1.49	C
	delete	0.75	$\|xs\|$	1.92	$\|xs\|$	1.24	C
	intersect	4.45	$\|xs\| + \|ys\|$	7.17	$\|xs\| + \|ys\|$	8.91	C
Sorting	insert (sorted)	0.90	$\|xs\|$	3.92	$\|xs\|$	2.16	C
	insertion sort	0.73	-	-	$\|xs\|^2$	6.03	C
	extract minimum	2.45	$\|xs\|$	28.66	$\|xs\|$	10.09	C
	quick sort	6.76	-	-	$\|xs\|^2$	39.29	C
	selection sort	1.84	-	-	$\|xs\|^2$	3.42	C
	balanced split	4.09	$\|xs\|$	28.59	$\|xs\|$	9.59	C
	merge	7.14	-	-	$\|xs\| + \|ys\|$	37.71	C
	merge sort	6.89	-	-	$\|xs\| \log \|xs\|$	69.63	A
	partition	5.77	$\|xs\|$	40.55	$\|xs\|$	10.77	C
	append with pivot	1.33	-	-	$\|xs\|$	1.96	C
Tree	is member	0.97	$2\|t\|$	8.88	$\|t\|$	5.47	T
	node count	0.84	$2\|t\|$	8.94	$\|t\|$	2.94	T
	preorder	1.07	$2\|t\|$	7.42	$\|t\|$	5.69	T

Table 4.3: Continuation of Table 4.2.

	Problem	SynQuid	ReSyn		SynPlexity		Rec.
		T(sec)	B	T(sec)	O(B)	T(sec)	rel.
BST	is member	0.75	2|t|	1.83	|t|	1.54	T
	insert	2.14	|t|	12.87	|t|	6.56	T
	delete	9.89	2|t|	98.04	|t|	24.20	T
	BST sort	4.23	3|t|	54.47	|t|	6.28	T
Binary Heap	is member	1.45	2|t|	0.97	|t|	2.09	T
	insert	2.01	|t|	11.89	|t|	4.02	T
	1-element constructor	0.90	1	2.11	1	1.29	N
	2-element constructor	1.15	2	2.63	1	1.04	N
	3-element constructor	5.34	3	62.69	1	5.06	N
AVL	rotate left	9.84	-	-	1	9.28	N
	rotate right	28.67	-	-	1	30.44	N
	balance	3.95	-	-	1	4.22	N
	insert	3.92	-	-	$\log \lvert t \rvert$	13.36	M
	delete	7.99	-	-	$\log \lvert t \rvert$	13.82	M
	extract minimum	8.22	-	-	$\log \lvert t \rvert$	12.26	M
RBT	balance left	12.63	-	-	1	11.15	N
	balance right	14.81	-	-	1	15.70	N
	insert	3.00	-	-	$\log \lvert t \rvert$	9.68	M
User	make address book	6.50	-	-	$\lvert \mathtt{adds} \rvert$	4.89	C
	merge address books	1.50	-	-	1	1.64	N
HOF	map	0.03	|xs|	0.33	|xs|	0.58	C
	zip with function	0.07	|xs|	0.82	|xs|	1.11	C
	foldr	0.10	|xs|	1.88	|xs|	2.57	C
	length using fold	0.03	|xs|	0.67	|xs|	0.56	N
	append using fold	0.04	|xs|	0.34	|xs|	0.72	N
ReSyn only	triple-1	1.01	$2\lvert xs \rvert$	2.93	$\lvert xs \rvert$	1.75	C
	triple-2	1.01	$2\lvert xs \rvert$	6.16	$\lvert xs \rvert$	1.45	C
	concat list of lists	1.30	$\lvert xss \rvert$	9.68	$\lvert xss \rvert$	1.79	C
	common	2.57	$\lvert ys \rvert + \lvert zs \rvert$	40.77	$\lvert ys \rvert + \lvert zs \rvert$	57.55	C
	list difference	1.36	$\lvert ys \rvert + \lvert zs \rvert$	419.23	$\lvert ys \rvert + \lvert zs \rvert$	41.88	C
	insert	1.18	$\mathtt{numgt}(x, xs)$	48.82	$\mathtt{numgt}(x, xs)$	3.76	C
	range	TO	$hi - lo$	128.8	$hi - lo$	7.63	C
	compare	1.02	$\lvert xs \rvert + \lvert ys \rvert$	3.78	$\lvert xs \rvert + \lvert ys \rvert$	8.32	C
SynPlexity only	binary search	1.53	-	-	$\log \lvert xs \rvert$	5.20	M
	product	1.09	-	-	$\log x$	1.37	M
	binary search′	1.64	-	-	$\log \lvert xs \rvert$	24.68	M
	product′	0.98	-	-	$\log x$	14.13	M
	merge sort′	TO	-	-	$\lvert xs \rvert \log \lvert xs \rvert$	75.73	A

Part III

Summary and Future Work

Chapter 5

Related Work

In this chapter, we compare the work described in this dissertation to related work.

Qualitative Synthesis

Existing program synthesizers fall into three categories: (*i*) enumeration solvers, which typically output the smallest program [Sin], (*ii*) symbolic solvers, which reduce the synthesis problem to a constraint-solving problem and output whatever program is produced by the constraint solver [SL13], (*iii*) probabilistic synthesizers, which randomly search the space for a solution and are typically unpredictable [SSA16]. Since the introduction of the SyGuS format [ABJ+13], these techniques have been used to build several SyGuS solvers that have competed in SyGuS competitions [AFSSL16b]. The most effective ones, which are used in the chapter about QSyGuS, are ESolver and EUSolver [Sin] (enumeration), and CVC4 [BCD+11] (symbolic).

Prior work [FN21, ARU17c] on synthesizing divide-and-conquer programs are focusing on finding solution equivalent to some reference programs. However, they are not resource-aware approaches.

Quantitative synthesis

Domain-specific synthesizers typically employ hard-coded ranking functions that guide the search towards a "preferable" program [PG15], but these functions are typically hard to write and are decoupled from the functional specification. Unlike QSyGuS, these synthesizers allow arbitrary ranking functions to be expressed in general-purpose languages, but typically only support limited grammars for synthesis. Moreover, in many practical applications the ranking functions are very simple. For example, the popular spreadsheet-formula synthesizer FlashFill [Gul11] uses a ranking function to prefer small programs with few constants. This type of objective is expressible in QSyGuS framework.

The Sketch synthesizer supports optimization objectives over variables in sketched programs [SGSL13]. This work differs from QSyGuS in that sketches are a different specification mechanism from SyGuS. In Sketch, the search space is encoded as a program with holes to facilitate synthesis by constraint solving. Translating SyGuS problems into sketches is non-trivial and results in poor performance.

The work closest to QSyGuS is Synapse [BTGC16], which combines sketching with an approach similar to QSyGuS. For the same reasons as for Sketch, Synapse differs from QSyGuS because it proposes a different search-space mechanism. However, there are a few analogies related ideas in QSyGuS and Synapse that are worth explaining in detail. Synapse supports syntactic-cost functions that are defined using a decidable theory, and separately from the sketch search space. Synthesis is done using an iterative search where sketches—i.e., set of partial programs with holes—of increasing sizes are given to the synthesizer. At a high level, the intermediate sketches are related to the notion in QSyGuS of reduced grammars—i.e., they accept solution of weight less than a given constant. However, while QuaSi generates reduced grammars automatically for a well-defined family of semirings, Synapse requires the user to provide a function for generating the intermediate sketches. Moreover, since Synapse requires cost functions that are defined using a decidable theory, it would not support certain families of costs that QSyGuS supports—e.g., the probabilistic semiring.

Koukoutos et al. [KRKK17] have proposed the use of probabilistic tree grammars to guide the search of enumerative synthesizers on applications outside of SyGuS. Their algorithm enumerates all terms accepted by the grammar in decreasing probability using a variant of the search algorithm A* and requires the grammar to not contain transitions of weight 1 to avoid getting stuck. Probabilistic tree grammars are a special case of the QuaSi algorithm with limitations of what weights can appear in the grammar. Moreover, the QuaSi algorithm does not require implementing a new solver when changing the cost semiring.

SyGuS

The SyGuS formalism was introduced as a unifying framework to express several synthesis problems [ABJ+13]. Caulfield et al. [CRST15] proved that it is undecidable to determine whether a given SyGuS problem is realizable. Despite this negative result, there are several SyGuS solvers that compete in yearly SyGuS competitions [AFSSL16b] and can efficiently produce solutions to SyGuS problems when a solution exists. Existing SyGuS synthesizers fall into three categories: (*i*) Enumeration solvers enumerate programs with respect to a given total order [Sin]. If the given problem is unrealizable, these solvers typically only terminate if the language of the grammar is finite or contains finitely many functionally distinct programs. While in principle certain enumeration solvers can prune infinite portions of the search space, none of these solvers could prove unrealizability for any of the benchmarks considered in the evaluation of Nope and Nay. (*ii*) Symbolic solvers reduce the synthesis problem to a constraint-solving problem [BCD+11]. These solvers cannot reason about grammars that restrict allowed terms, and resort to enumeration whenever the candidate solution produced by the constraint solver is not in the restricted search space. Hence, they also cannot prove unrealizability. (*iii*) Probabilistic synthesizers randomly search the search space, and are typically unpredictable [SSA16], providing no guarantees in terms of unrealizability.

Synthesis as Reachability

CETI [NWKF17] introduces a technique for encoding template-based synthesis problems as reachability problems. The CETI encoding only applies to the specific setting in which (*i*) the search space is described by an imperative program with a *finite number* of holes—i.e., the values that the synthesizer has to discover—and (*ii*) the specification is given as a finite number of input-output test cases with which the target program should agree. Because the number of holes is finite, and all holes correspond to values (and not terms), the reduction to a reachability problem only involves making the holes global variables in the program (and no more elaborate transformations).

In contrast, our reduction technique in Nope handles search spaces that are described by a grammar, which in general consist of an infinite set of terms (not just values). Due to this added complexity, our encoding used in Nope has to account for (i) the semantics of the productions in the grammar, and (ii) the use of non-determinism to encode the choice of grammar productions. Our encoding creates one expression-evaluation computation for each of the example inputs, and threads these computations through the program so that each expression-evaluation computation makes use of the *same* set of non-deterministic choices.

Using the input-threading, our technique used in Nope can handle specifications that contain nested calls of the synthesized program (e.g., $f(f(x)) = x$).

The input-threading technique builds a *product program* that perform multiple executions of the same function as done in relational program verification [BCK11]. Alternatively, a different encoding could use multiple function invocations on individual inputs and require the verifier to thread the same bit-stream for all input evaluations. In general, verifiers perform much better on product programs [BCK11], which motivate the choice of encoding used in Nope.

Unrealizability in Program Synthesis

For certain synthesis problems—e.g., reactive synthesis [Blo15]—the realizability problem is decidable. The SyGuS framework is orthogonal to such problems, and it

is undecidable to check whether a given SyGuS problem is realizable.

Mechtaev et al. [MGCR18] propose to use a variant of SyGuS to efficiently prune irrelevant paths in a symbolic-execution engine. In their approach, for each path π in the program, a synthesis problem p_π is generated so that if p_π is unrealizable, the path π is infeasible. The synthesis problems generated by Mechtaev et al. (which are not directly expressible in SyGuS) are decidable because the search space is defined by a finite set of templates, and the synthesis problem can be encoded by an SMT formula. To the best of our knowledge, our technique is the first one that can check unrealizability of general SyGuS problems in which the search space is an *infinite set of functionally distinct terms*.

Abstractions in Program Synthesis

SYNGAR [WDS18] uses predicate abstraction to prune the search space of a synthesis-from-examples problem. Given an input example i and a regular-tree grammar A representing the search space, SYNGAR builds a new grammar A_α in which each nonterminal is a pair (q, a), where q is a nonterminal of A and a is a predicate of a predicate-abstraction domain α. Any term that can be derived from (q, a) is guaranteed to produce an output satisfying the predicate a when fed the input i. A_α is constructed iteratively by adding nonterminals in a bottom-up fashion; it is guaranteed to terminate because the set α is finite. SYNGAR can be viewed as a special case of our framework of Nay in which the set of values $n_\mathcal{G}(X)$ is based on predicate abstraction (see §3.4). SYNGAR's approach is tied to finite abstract domains, while the equational approach used in Nay extends to infinite domains—e.g., semi-linear sets—because it does not specify how the equations must be solved.

Resource-Bound Analysis

Rather than determining whether a given program satisfies a specification, a synthesizer determines whether there exists a program that inhabits a given specifica-

tion. The branch of verification that we draw upon for resource-based synthesis is resource-bound analysis [Weg75].

Within the literature on automated resource-bound analysis, there are methods that extract and solve recurrence relations for imperative code [AAGP11, FM17, BCKR20, KCBR18]. However, these methods are unlike the type system presented in Chapter 4 because they extract concrete complexity bounds as recurrence relations, and then solve the recurrences to find a concrete upper bound on resource usage. The dominant terms of the resulting concrete bounds can then be used to state a big-O complexity bound. In contrast, we want to synthesize programs with respect to a big-O complexity directly, which is more similar to the manual reasoning of [Ebe17, GCP18]. Thus, if we were to use these techniques in SynPlexity, the first step in the SynPlexity synthesis algorithm would be to pick a concrete complexity function given a big-O complexity, and then reverse the verification problem with regards to that concrete complexity. However, for any big-O complexity, there are an infinite number of functions that satisfy that complexity, which presents a significant challenge at the outset. Our design choice in SynPlexity also has some drawbacks. As noted in [GCP18], reasoning compositionally with big-O complexity is challenging due to the hidden quantifier structure of big-O notation. Thus, to maintain soundness, the SynPlexity type system has to sacrifice precision and generality in some places. For example, when a function has multiple paths, the SynPlexity type system over-approximates by choosing the largest complexity among all the paths.

Another set of methods to generate resource bounds are type-based [HAH11, HAH12, WWC17, KH20]. As we discussed in Chapter 4, the complexities generated by these methods are concrete functions and not expressed with big-O notation, although [WWC17] is sometimes able to pattern match a case of the Master Theorem. These type systems differ from ours in a few ways. The AARA line of research [HAH11, HAH12, KH20] is able to assign amortized complexity to programs, but is not able to generate logarithmic bounds. [WWC17] is also able to perform amortized analysis; however, the technique is not fully automated, and instead requires the user to provide type annotations on terms, which are then checked by the type

system.

Type- and Resource-Aware Synthesis

The SynPlexity implementation is built on top of Synquid [PKSL16] a type-directed synthesis tool based on refinement types and polymorphism. The work that most closely resembles ours is ReSyn [KWPH19]. As in SynPlexity, they combine the type-directed synthesizer Synquid with a type system that is able to assign complexity bounds to functional programs. The type system used in ReSyn is based on one originally used in the context of verification [HAH12]. That work uses a sophisticated type system to assign amortized resource-usage bounds to a given program. The type system of ReSyn differs from the one presented in §4.3 in a few significant ways.

As highlighted earlier, the technique of ReSyn for automatically inferring bounds on recursive functions is based on amortized analysis, and restricted to linear bounds, whereas SynPlexity system is able to synthesize complexities of the form $O(n^a \log^b n + c)$.

Another difference is that ReSyn synthesizes programs with a concrete complexity bound. This approach has advantages and disadvantages. For instance, it places an extra burden on the human to provide the correct bound with precise coefficient. On the other hand, the user might want an implementation that has a complexity with a small coefficient, whereas SynPlexity system provides no guarantee that the complexity of an implementation will have a small coefficient in the dominant term: SynPlexity only guarantees asymptotic behavior.

ReSyn can synthesize programs with higher-order functions, which are supported only in a restricted manner by SynPlexity. To handle higher-order functions, ReSyn attaches resource units to types, which gives it *resource polymorphism*. Moreover, costs of inputs with function types can be written generally as polymorphic types (i.e., costs can be polymorphic with respect to the size of the specific input types). SynPlexity does not have *asymptotic resource polymorphism* because it cannot directly compose unknown big-O functions (i.e., the complexity of higher-order

inputs). We envision that with carefully crafted restrictions on the resource annotations of higher-order functions, SynPlexity could handle synthesis problems involving such functions, e.g., assuming that the complexity of input functions is known and the refinements of input functions are precise enough. Detailed discussion about these restrictions can be found in §4.6. Because big-O functions cannot be directly composed, developing a more general extension to SynPlexity that supports higher-order functions is a challenging direction for future work.

Chapter 6

Future Work

This dissertation demonstrates that quantitative objectives and proof of unrealizability are effective guarantees for making program synthesis more reliable and predictable. However, our work does not include all possible guarantees in program synthesis. This final chapter identifies some other promising guarantees in program synthesis for future work.

6.1 More about Nope

Solving QSyGuS with Nope

As described in §3.3, we can use Nope to show that a solution to some QSyGuS problem is optimal by combining QuaSi and Nope. Recall that QuaSi will keep refining the cost of the current solution by solving SyGuS sub-problems with restricted grammars. And in the last iteration of the cost refinement, the SyGuS sub-problem is unrealizable only if the current QSyGuS solution is optimal. However, in such a combined technique, the SyGuS sub-problem can have very large grammars, e.g., with more than 200 production rules, which is challenging for verification tools like SeaHorn. To address this large-grammar issue, we propose a new reduction encoding in Nope such that we can avoid grammars with large sizes and solve QSyGuS problem directly with Nope.

The high-level idea is that we add a new global variable to the encoded program to record the cost of productions applied so far. Every time we enter an if-statement body in the encoded program, we add to the cost variable the weight of the corresponding production. Finally, when the encoded program returns, the cost variable will store the overall weight of the evaluated expression.

Symbolic Constants

In most of SyGuS benchmarks, there are only two constants 0 and 1 appearing in the grammars, Other constants are presented as the sum of multiple 1s, which enlarge the size of solutions and slow down the enumerating technique used in enumerative solvers [ABJ+13, ARU17a]. One promising optimization of Nope to deal with constants is using symbolic constants in grammars. Using symbolic constants in program synthesis has been recently studied in CEGIS[$\mathcal{T}$] [ADK+18] for counter-example-guided synthesis. What we envision differs from CEGIS[$\mathcal{T}$] because it involves a new algorithm to determine the allowed ranges of symbolic constants in a given grammar and thus we can use symbolic constants in SyGuS problems with non-trivial grammars while with only CEGIS[$\mathcal{T}$] we cannot.

The high-level idea is that for each nonterminal N, we first decide the range ϕ_{const} of constants derived from it and then replace the concrete constant in the encoded program by a symbolic variable `constant_N` and add an assertion `assert` ϕ_{const}`(constant_N)` in the if-statement body corresponds to the constant production.

The correctness of the symbolic constant can be shown by a bijection map to the original constant encoding. Denote by P the encoded program without using symbolic constants and P^{sym} the encoded program with symbolic constants. The leftward direction shows that for any expression e evaluated in P, e can also be evaluated in P^{sym}, which is trivial. The other direction shows that for each expression e evaluated in P^{sym} there exists an expression e' in P such that $[\![e]\!] = [\![e']\!]$.

Imperative Programs

The technique used in Nope is not restricted to the SyGuS framework. That is, it should be possible to generalize the technique used in Nope to imperative programs. Imperative programs have more expressiveness than expressions produced by regular tree grammars. The synthesis of imperative programs has been studied using a deductive approach [SI99]. Our approach based on Nope cannot only synthesize a correct solution but also has the ability to answer unrealizable and to incorporate syntactic quantitative objectives.

The main challenge when using the encoding presented in Nope to encode imperative synthesis problems is the assignment statement. For SyGuS problems, we only care about the evaluated values of terms while, for imperative program synthesis, we need to also consider program states, e.g., when we encode a rule $r : S \rightarrow \text{Concate}(\text{Assign}(x, E), S)$, the evaluation of E will influence the evaluation of S.

To deal with program states, we evaluate a sub-term t to a pair $(eval_t, \sigma)$ where $eval_t$ is the evaluated value of t, similar to what happens in Nope, and σ is a store of values assigned to variables. Recall that in the original Nope, we use global variables `g_i_j` only to store the evaluated values of sub-terms. In the encoding for imperative programs, we would use additional global variables to record program states σ, i.e., for each variable x we add a new global variable `g_x_i_j`, pass its value to sub-terms via function arguments and update it when a function call returns. For example, with only one input example, we encode the rule r as

```
funcE(v)
τ temp_x = g
g = funcS(temp_x)
```

The new encoding is promising because in the encoding used in Nope we evaluate the whole tree using a postorder traversals with which we can make sure that early statements will be evaluated first and program states can be passed in sequential order.

Abstraction in Solving Synthesis Problems

In the algorithm of NOPE and NAY, to synthesize solutions and prove unrealizability, we use concrete values to build the example set E and evaluate candidate solutions. One possible generalization of this approach is to use abstraction instead of concrete values. Using abstraction in solving synthesis problems differs from using concrete values in two ways: first, we can have a symbolic example set in the CEGIS loop, and we evaluate abstractly sub-terms, instead of concretely. Using abstraction in program synthesis has been studied for synthesis problems with examples [WDS18].

6.2 Complex Quantitative Objectives

Semantic Quantitative Objectives

When we know a program-synthesis problem is unrealizable, a question naturally arises: *how much of the correct specification can we satisfy on a best-effort basis?* With such objectives, we can find a solution to satisfy, for example, as many input-output examples as possible. Such quantitative objectives can also be used to train a neural network. Another example application is approximate synthesis [BTCG15], which allow us to find an approximate solution, instead of finding an exact solution that fulfills the correct specification within a given amount of solver solving time. We now describe a possible formalization of semantic quantitative objectives.

A natural way to formalize a synthesis problem with semantic quantitative objectives is that, instead of only one correct specification, we allow a program-synthesis problem to contain a set of weight-specification pairs (φ_i, w_i). We define the semantic weight of a solution e as $\sum_i [\varphi_i(e) = \mathtt{true}] w_i$. Similar to the syntactic quantitative objectives we presented in Chapter 2, a semantic objective could be specified as a range of allowed weights of solutions, or a requirement of optimizing the solutions' weight.

Some possible algorithms to solve synthesis problems with semantic quantitative objectives include

- Iteratively refining costs by construct sub-problems that produce terms with better costs. For example, if a current solution satisfy all but one specification clause, to refine it we need to solve a synthesis problem with specification as the conjunction of all original specification clauses.
- Encoding the whole problem as a MAX-SAT problem and solving it meaning a call to a MAX-SAT solver.

Multi-Objective Synthesis Framework

Specifications associated with semantic quantitative objectives are *soft* and ambiguous—there can be multiple solutions satisfying different parts of the specifications. We propose to allow multiple objectives to empower users to mitigate this problem.

Example 6.1. *Consider a scenario where an end-user is using a synthesizer to write a regular expression* [PHXD19]. *The user provides a set of positive examples* {*12:14, 06:4, 07:03*} *and a set of negative examples* {*07:4, 7:04*} *for synthesizing a time-format expression accepting strings of the form XX:YY. Here, the user has made a mistake and the string 06:4 is a false positive.*

If we assign weight 1 to each positive example and ask for a solution with weight at least 2 (*i.e., the user can make one mistake*)*, there will be two possible solutions* $e_1 := [0\text{-}9](2) : [0\text{-}9](2)$ *which satisfies only two positive examples, and* $e_2 := ([0\text{-}9](2) : [0\text{-}9](2)) \mid (06 : 4)$ *which satisfies all three positive examples. The syntactic constraint* $|e^*| \leqslant 15$ *will rule out the overfitted solution* e_2.

One might hope to develop a multi-objective synthesis framework that supports 1) soft specifications as semantic quantitative objectives, 2) hard specifications, and 3) syntactic quantitative objectives. Such a framework might use type-guided synthesis with refinement types [KWPH19], which can contain both syntactic and semantic information. It might also take a compositional approach to solve individual objectives separately and combine the obtained solutions.

User-Friendly Specification Language

Sometimes users know the main functionality of the expected program, but find it hard to write a formal specification covering all the corner cases. We plan to allow the user to provide a partial specification that constrains only parts of the inputs.

Example 6.2. *A user wants to synthesize the absolute-value function, but writing the full logical specification* $\varphi_{abs} = (x > 0 \to f(x) = x) \wedge (x < 0 \to f(x) = -x)$ *is hard for an end-user. One might allow the user to provide a partial specification "*$f(x) = x$ *on infinite number of inputs" along with input-output examples* $f(-1) = 1, f(-2) = 2$ *and a syntactic objective that minimizes the size of the solution. This specification is enough for synthesizing* `abs`*, and is composed of many natural easy-to-write individual objectives.*

In Regel [CWY+19], users can specify synthesis problems with a mixture of natural language and examples, which are then solved with a combination of learning and symbolic techniques. One might attack the problem in Ex. 6.2 using a similar approach.

Bibliography

[AAGP11] Elvira Albert, Puri Arenas, Samir Genaim, and Puebla Puebla, Germán. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 2011.

[AB98] Mohamad Akra and Louay Bazzi. On the solution of linear recurrence equations. *Computational Optimization and Applications*, 10:195–210, 1998.

[ABJ+13] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 1–8. IEEE, 2013.

[ADK+18] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*, pages 270–288. Springer, 2018.

[AFSSL16a] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Results and analysis of SyGuS-comp'15. *arXiv preprint arXiv:1602.01170*, 2016.

[AFSSL16b] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

[ARU17a] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336. Springer, 2017.

[ARU17b] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336. Springer, 2017.

[ARU17c] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 319–336, 2017.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *International Symposium on Formal Methods (FM)*, pages 200–214. Springer, 2011.

[BCKR20] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas Reps. Templates and recurrences: Better together. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2020.

[BET03] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures.

International Journal of Foundations of Computer Science, 14(04):551–582, 2003.

[BHS80] Jon Louis Bentley, Dorothea Haken, and James B Saxe. A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, 12(3):36–44, 1980.

[Blo15] Roderick Bloem. Reactive synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 3–3, 2015.

[BTCG15] James Bornholt, Emina Torlak, Luis Ceze, and Dan Grossman. Approximate program synthesis. 2015.

[BTGC16] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 775–788, New York, NY, USA, 2016. ACM.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, 1978.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3nd edition, 2009.

[CRST15] Benjamin Caulfield, Markus N Rabe, Sanjit A Seshia, and Stavros Tripakis. What's decidable about syntax-guided synthesis? *arXiv preprint arXiv:1510.08393*, 2015.

[CWY+19] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. Multi-modal synthesis of regular expressions. *arXiv preprint arXiv:1908.03316*, 2019.

[DGH+16] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*, pages 345–356, 2016.

[DKV09] Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of Weighted Automata*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DSS16] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 383–401, 2016.

[DV06] Manfred Droste and Heiko Vogler. Weighted tree automata and weighted logics. *Theoretical Computer Science*, 366(3):228 – 247, 2006. Automata and Formal Languages.

[Ebe17] Manuel Eberl. Proving divide and conquer complexities in isabelle/hol. *Journal of Automated Reasoning*, 58(4):483–508, 2017.

[EKL10] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newtonian program analysis. *J. ACM*, 57(6):33:1–33:47, 2010.

[FM17] Antonio Flores Montoya. *Cost Analysis of Programs Based on the Refinement of Cost Relations*. PhD thesis, TU Darmstadt, 2017.

[FN21] Azadeh Farzan and Victor Nicolet. Phased synthesis of divide and conquer programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 974–986, 2021.

[GCP18] Armaël Guéneau, Arthur Charguéraud, and François Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *European Symposium on Programming*, pages 533–560. Springer, 2018.

[GKN15] Arie Gurfinkel, Temesghen Kahsai, and Jorge A. Navas. SeaHorn: A framework for verifying C programs (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 447–450, 2015.

[GM14] Sumit Gulwani and Mark Marron. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 803–814, 2014.

[Gul11] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330, 2011.

[Gul16] Sumit Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 9–14, 2016.

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *Proceedings of the 38th annual ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, pages 357–370, 2011.

[HAH12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ml. In *The International Conference on Computer Aided Verification*, CAV, 2012.

[HD17] Qinheping Hu and Loris D'Antoni. Automatic program inversion using symbolic transducers. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 376–389, 2017.

[HSSD19] Qinheping Hu, Roopsha Samanta, Rishabh Singh, and Loris D'Antoni. Direct manipulation for imperative programs. In *International Static Analysis Symposium*, pages 347–367. Springer, 2019.

[KCBR18] Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. Non-linear reasoning for invariant synthesis. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL, 2018.

[KH20] David M. Khan and Jan Hoffmann. Exponential automatic amortized resource analysis. In *International Conference on Foundations of Software Science and Computation Structures*, FoSSaCs, 2020.

[KP11] Manuel Kauers and Peter Paule. *The Concrete Tetrahedron: Symbolic Sums, Recurrence Equations, Generating Functions, Asymptotic Estimates*. Springer Science & Business Media, 2011.

[KRKK17] Manos Koukoutos, Mukund Raghothaman, Etienne Kneuss, and Viktor Kuncak. On repair with probabilistic attribute grammars. *CoRR*, abs/1707.04148, 2017.

[KT10] Eryk Kopczynski and Anthony Widjaja To. Parikh images of grammars: Complexity and applications. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 80–89. IEEE, 2010.

[KU77] J.B. Kam and J.D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–318, 1977.

[KWPH19] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–268, 2019.

[LGS13] Vu Le, Sumit Gulwani, and Zhendong Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 193–206, 2013.

[MGCR18] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. Symbolic execution with existential second-order constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 389–399, 2018.

[MW91] Ulrich Möncke and Reinhard Wilhelm. Grammar flow analysis. In *International Summer School on Attribute Grammars, Applications, and Systems*, pages 151–186. Springer, 1991.

[NDAFH17] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. Verifying and synthesizing constant-resource implementations with types. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 710–728, May 2017.

[NWKF17] ThanhVu Nguyen, Westley Weimer, Deepak Kapur, and Stephanie Forrest. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 301–318, 2017.

[Par66] Rohit J. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, October 1966.

[PG15] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126, 2015.

[PHXD19] Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. Automatic repair of regular expressions. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[PKSL16] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[PR89] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, 1989.

[PT00] Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[Ram96] Ganesan Ramalingam. *Bounded incremental computation*, volume 1089. Springer, 1996.

[RDK+15] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in smt. In *International Conference on Computer Aided Verification*, pages 198–216. Springer, 2015.

[RSY04] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, 2004.

[SG15] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 398–414, 2015.

[SGSL13] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of PLDI'13*, pages 15–26, New York, NY, USA, 2013. ACM.

[SI99] Jamie Stark and Andrew Ireland. Towards automatic imperative program synthesis through proof planning. In *14th IEEE International Conference on Automated Software Engineering*, pages 44–51. IEEE, 1999.

[Sin] Rishabh Singh. *ESolver*. `https://github.com/abhishekudupa/sygus-comp14`.

[SL08] Armando Solar-Lezama. *Program synthesis by sketching*. Citeseer, 2008.

[SL13] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5):475–495, Oct 2013.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[SSA16] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic program optimization. *Commun. ACM*, 59(2):114–122, 2016.

[WDS18] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *PACMPL*, 2(POPL):63:1–63:30, 2018.

[Weg75] Ben Wegbreit. Mechanical program analysis. In *Communications of the ACM*, 1975.

[WWC17] Peng Wang, Di Wang, and Adam Chlipala. Timl: A functional language for practical complexity analysis with invariants. 2017.