Exact and Approximate Methods for Proving Unrealizability of Syntax-Guided Synthesis Problems

Qinheping Hu
University of Wisconsin-Madison
USA

John Cyphert
University of Wisconsin-Madison
USA

Loris D'Antoni
University of Wisconsin-Madison
USA

Thomas Reps
University of Wisconsin-Madison
USA

Abstract

We consider the problem of automatically establishing that a given syntax-guided-synthesis (SyGuS) problem is unrealizable (i.e., has no solution). We formulate the problem of proving that a SyGuS problem is unrealizable over a finite set of examples as one of solving a set of equations: the solution yields an overapproximation of the set of possible outputs that any term in the search space can produce on the given examples. If none of the possible outputs agrees with all of the examples, our technique has proven that the given SyGuS problem is unrealizable. We then present an algorithm for exactly solving the set of equations that result from SyGuS problems over linear integer arithmetic (LIA) and LIA with conditionals (CLIA), thereby showing that LIA and CLIA SyGuS problems over finitely many examples are decidable. We implement the proposed technique and algorithms in a tool called NAY. NAY can prove unrealizability for 70/132 existing SyGuS benchmarks, with running times comparable to those of the state-of-the-art tool NOPE. Moreover, NAY can solve 11 benchmarks that NOPE cannot solve.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; • **Theory of computation** → **Abstraction**.

Keywords: Program Synthesis, Unrealizability, Grammar Flow Analysis, Syntax-Guided Synthesis (SyGuS)

ACM Reference Format:
Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. 2020. Exact and Approximate Methods for Proving Unrealizability of Syntax-Guided Synthesis Problems. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3385979

PLDI '20, June 15–20, 2020, London, UK
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385979

1 Introduction

The goal of program synthesis is to find a program in some search space that meets a specification—e.g., satisfies 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 regular-tree grammar that describes the search space of programs, and a logical formula that constitutes the behavioral specification. Many synthesizers support a specific format for SyGuS problems [1], and compete in annual synthesis competitions [2]. These solvers are now quite mature and are finding a wealth of applications [9, 12].

While existing SyGuS synthesizers are good at finding a solution when one exists, there has been only a small amount of work on methods to prove that a given SyGuS problem is *unrealizable*—i.e., the problem does not admit a solution. The problem of proving unrealizability arises in applications such as pruning infeasible paths in symbolic-execution engines [16] and computing syntactically optimal solutions to SyGuS problems [13]. However, proving that a SyGuS problem is unrealizable is particularly hard and, in general, undecidable [6]. When 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.

Although we cannot hope to have a complete algorithm for establishing unrealizability, the goal of this paper is to develop a framework for solving the kinds of problems that appear in practice. Our framework can be used in tandem with existing synthesizers that use the *counterexample-guided inductive synthesis* (CEGIS) approach, in which the synthesizer iteratively builds a set of input examples and finds programs consistent with the examples.

Our approach builds on the observation that unrealizability of a SyGuS problem *sy* can be proved by showing, for some finite set of examples E, that sy^E—the same problem with the weaker specification of merely satisfying the examples in E—is unrealizable [11]. We combine this observation with techniques from the abstract-interpretation literature to show that determining realizability of a linear integer arithmetic (LIA) SyGuS problem over a finite set of examples is actually *decidable*. Our work gives a decision procedure to show unrealizability for a sy^E instance, whereas the prior work by Hu et al. [11] reduced the problem to a program-reachability problem. In their approach, if an assertion inside a constructed program is shown to be valid, then the original problem is unrealizable. The issue with prior work is that the resulting reachability problem is passed to an incomplete solver that may not terminate or may only return unknown.

Even though we consider a finite set of examples, showing realizability is non-trivial because the grammar can still generate an infinite set of terms. The main idea of this paper is to use an abstract domain to overapproximate the possibly infinite set of outputs that the terms derivable from each non-terminal of the grammar of sy^E can produce on examples E. The overapproximation is formalized using *grammar-flow-analysis* (GFA), a method that extends dataflow analysis to grammars [17]. We define a GFA problem whose solution associates an overapproximating abstract-domain value with each non-terminal of the SyGuS grammar. We then use the notion of *symbolic concretization* [20] to represent the abstract values as logical formulas, which get combined with the SyGuS specification to produce an SMT query whose result can imply that the original problem is unrealizable.

Using this framework, a variety of abstract domains can be used to show unrealizability for arbitrary SyGuS problems. However, we also give a particular instantiation of the framework to obtain a *decision procedure for (un)realizability of LIA SyGuS problems over a finite set of examples*. The key to this reduction is the use of the abstract domain of *semi-linear sets*. We show that the GFA problem over semi-linear sets can be solved to yield a semi-linear set that *exactly* captures the set of possible outputs of the SyGuS grammar. The problem sy^E is unrealizable if and only if the semi-linear set for the start non-terminal of the grammar contains no value that satisfies the specification. We extend this result to SyGuS problems whose grammar contains LIA terms and conditionals (CLIA).

Our work makes the following three contributions:

(1) We reduce the problem of proving unrealizability of a SyGuS problem, where the specification is given by examples, to the problem of solving a set of equations in an abstract domain (§2). The correctness of our reduction is based on the framework of grammar-flow analysis (§3 and §4).

(2) We show that the equations resulting from our reduction can be solved exactly for SyGuS problems in which the grammars only generate terms in LIA (§5) and CLIA (§6), therefore yielding the first *decision procedures* for LIA and CLIA SyGuS problems over a finite set of examples.

(3) We implement our technique in a tool, NAY (§7). NAY can prove unrealizability for 70/132 benchamrks that were used to evaluate the state-of-the-art tool NOPE. In particular, NAY can solve 11 benchmarks that NOPE could not solve (§8).

§9 discusses related work. Proofs and additional details can be found in the supplementary material.

2 Illustrative Examples

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:[1]

$$\mathit{Start} ::= \text{Plus}(\text{Var}(x), \text{Var}(x), \text{Var}(x), \mathit{Start}) \mid \text{Num}(0) \quad (1)$$

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

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 $n_{1,E}(A)$ [2] 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 §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:

$$\begin{aligned} n_{1,E}(\mathit{Start}) = [\![\text{Plus}]\!]^\#_E([\![\text{Var}(x)]\!]^\#_E, [\![\text{Var}(x)]\!]^\#_E, [\![\text{Var}(x)]\!]^\#_E,\\ n_{1,E}(\mathit{Start})) \oplus [\![\text{Num}(0)]\!]^\#_E. \end{aligned} \quad (2)$$

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, $[\![\text{Var}(x)]\!]^\#_E$ is the vector of inputs E projected onto the

[1] 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:
$\mathit{Start} ::= \text{Plus}(\mathit{S1}, \mathit{Start}) \mid \text{Num}(0) \qquad \mathit{S1} ::= \text{Plus}(\mathit{S2}, \text{Var}(x))$
$\mathit{S2} ::= \text{Plus}(\mathit{S3}, \text{Var}(x)) \qquad \mathit{S3} ::= \text{Var}(x).$

[2] This section uses a simplified notation for readability. In §4 the term $n_{1,E}(A)$ is written $n_{\mathcal{G}_{1E}}$ where $\mathcal{G}_1$ is used to denote a GFA problem.

x coordinate—i.e., $[\![\text{Var}(x)]\!]^{\#}_E = \{i_1(x)\} = \{1\}$. We rewrite $[\![\text{Plus}]\!]^{\#}_E$ as $\otimes$, with $x \otimes y$ being the semi-linear set representing $\{a + b \mid a \in x, b \in y\}$

We rewrite Eqn. (2) to use semi-linear sets:

$$n_{1,E}(\mathit{Start}) = \big(\{1\} \otimes \{1\} \otimes \{1\} \otimes n_{1,E}(\mathit{Start})\big) \oplus \{0\}, \tag{3}$$

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 [10].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:

$$\exists \lambda.[i_1 = 1 \wedge o_1 = 0 + \lambda 3 \wedge \lambda \geq 0] \wedge o_1 = 2i_1 + 2. \tag{4}$$

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:

$$\begin{aligned}
\mathit{Start} &::= \text{IfThenElse}(\mathit{BExp}, \mathit{Exp}3, \mathit{Start}) \mid \mathit{Exp}2 \mid \mathit{Exp}3\\
\mathit{BExp} &::= \text{LessThan}(\text{Var}(x), \text{Num}(2))\\
&\quad\mid \text{LessThan}(\text{Num}(0), \mathit{Start}) \mid \text{And}(\mathit{BExp}, \mathit{BExp})\\
\mathit{Exp}2 &::= \text{Plus}(\text{Var}(x), \text{Var}(x), \mathit{Exp}2) \mid \text{Num}(0)\\
\mathit{Exp}3 &::= \text{Plus}(\text{Var}(x), \text{Var}(x), \text{Var}(x), \mathit{Exp}3) \mid \text{Num}(0)
\end{aligned} \tag{5}$$

Consider again the input example i_1=1 with output o_1=4. The 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:[3]

$$\begin{aligned}
n_{2,E}(\mathit{Start}) &= [\![\text{IfThenElse}]\!]^{\#}_E(n_{2,E}(\mathit{BExp}), n_{2,E}(\mathit{Exp}3),\\
&\qquad n_{2,E}(\mathit{Start})) \oplus n_{2,E}(\mathit{Exp}2) \oplus n_{2,E}(\mathit{Exp}3)\\
n_{2,E}(\mathit{BExp}) &= [\![\text{LessThan}]\!]^{\#}_E([\![\text{Var}(x)]\!]^{\#}_E, [\![\text{Num}(2)]\!]^{\#}_E)\\
&\quad \oplus [\![\text{LessThan}]\!]^{\#}_E([\![\text{Num}(0)]\!]^{\#}_E, n_{2,E}(\mathit{Start}))\\
&\quad \oplus [\![\text{And}]\!]^{\#}_E(n_{2,E}(\mathit{BExp}), n_{2,E}(\mathit{BExp}))\\
n_{2,E}(\mathit{Exp}2) &= [\![\text{Plus}]\!]^{\#}_E([\![\text{Var}(x)]\!]^{\#}_E, [\![\text{Var}(x)]\!]^{\#}_E, n_{2,E}(\mathit{Exp}2))\\
&\quad \oplus [\![\text{Num}(0)]\!]^{\#}_E\\
n_{2,E}(\mathit{Exp}3) &= [\![\text{Plus}]\!]^{\#}_E([\![\text{Var}(x)]\!]^{\#}_E, [\![\text{Var}(x)]\!]^{\#}_E, [\![\text{Var}(x)]\!]^{\#}_E,\\
&\qquad n_{2,E}(\mathit{Exp}3)) \oplus [\![\text{Num}(0)]\!]^{\#}_E
\end{aligned} \tag{6}$$

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 $n_{2,E}(\mathit{Start})$, $n_{2,E}(\mathit{Exp}2)$ and $n_{2,E}(\mathit{Exp}3)$, and a set of Boolean vectors for $n_{2,E}(\mathit{BExp})$—e.g., $n_{2,E}(\mathit{BExp})$ could be a set $\{(\text{t}, \text{f}), (\text{t}, \text{t})\}$, which denotes that a Boolean expression generated by BExp can be true for i_1 and false for i_2, or true for both. We can now instantiate all constant terminals and variable terminals with their abstraction, e.g., $[\![\text{Var(x)}]\!]^{\#}_E$ with $\{(1, 2)\}$ and $[\![\text{Num}(0)]\!]^{\#}_E$ with $\{(0, 0)\}$. We then start solving part of our equations by observing that $\mathit{Exp}2$ and $\mathit{Exp}3$ are only recursive in themselves. Therefore, we can compute their summaries independently, obtaining $n_{2,E}(\mathit{Exp}2) = \{(0, 0) + \lambda(2, 4)\}$, $n_{2,E}(\mathit{Exp}3) = \{(0, 0) + \lambda(3, 6)\}$. We can now replace all instances of $n_{2,E}(\mathit{Exp}2)$ and $n_{2,E}(\mathit{Exp}3)$, and obtain the following set of equations:

$$\begin{aligned}
n_{2,E}(\mathit{Start}) &= [\![\text{IfThenElse}]\!]^{\#}_E(n_{2,E}(\mathit{BExp}), \{(0, 0) + \lambda(3, 6)\},\\
&\qquad n_{2,E}(\mathit{Start})) \oplus \{(0, 0) + \lambda(2, 4)\}\\
&\quad \oplus \{(0, 0) + \lambda(3, 6)\}\\
n_{2,E}(\mathit{BExp}) &= \{(\text{t}, \text{f})\} \oplus [\![\text{LessThan}]\!]^{\#}_E(\{(0, 0)\}, n_{2,E}(\mathit{Start}))\\
&\quad \oplus [\![\text{And}]\!]^{\#}_E(n_{2,E}(\mathit{BExp}), n_{2,E}(\mathit{BExp}))
\end{aligned} \tag{7}$$

We now have to face the problem of solving equations over $n_{2,E}(\mathit{BExp})$ and $n_{2,E}(\mathit{Start})$, which represent different types of values and are mutually recursive. Because the domain of $n_{2,E}(\mathit{BExp})$ 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 $n^{\emptyset}_{2,E}(\mathit{BExp}) = \{(\text{t}, \text{f})\}$ (the superscript denotes the iteration the algorithm is in). We can replace $n_{2,E}(\mathit{BExp})$ with the value of $n^{\emptyset}_{2,E}(\mathit{BExp})$ in the equation for $n^1_{2,E}(\mathit{Start})$ as follows:

$$\begin{aligned}
n^1_{2,E}(\mathit{Start}) &= [\![\text{IfThenElse}]\!]^{\#}_E(\{(\text{t}, \text{f})\}, \{(0, 0) + \lambda(3, 6)\},\\
&\qquad n^1_{2,E}(\mathit{Start})) \oplus \{(0, 0) + \lambda(2, 4)\}\\
&\quad \oplus \{(0, 0) + \lambda(3, 6)\}
\end{aligned} \tag{8}$$

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

[3] 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.

problem by rewriting the above equations as follows:

$$\begin{aligned} n^1_{2,E}(Start^{(\mathrm{t},\mathrm{t})}) &= \{(0,0)+\lambda(3,0)\} \otimes n^1_{2,E}(Start^{(\mathrm{f},\mathrm{t})}) \\ &\quad \oplus \{(0,0)+\lambda(2,4)\} \oplus \{(0,0)+\lambda(3,6)\} \\ n^1_{2,E}(Start^{(\mathrm{f},\mathrm{t})}) &= \{(0,0)+\lambda(0,0)\} \otimes n^1_{2,E}(Start^{(\mathrm{f},\mathrm{t})}) \\ &\quad \oplus \{(0,0)+\lambda(0,4)\} \oplus \{(0,0)+\lambda(0,6)\} \end{aligned} \tag{9}$$

Intuitively, $n^1_{2,E}(Start^{(\mathrm{f},\mathrm{t})})$ 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 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 $n^1_{2,E}(Start^{(\mathrm{t},\mathrm{t})})$ (which is also the value of $n^1_{2,E}(Start)$), 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 $n^1_{2,E}(Start)$. We now plug that value into the equation for *BExp* and compute the value of $n^1_{2,E}(BExp)$,

$$\begin{aligned} n^1_{2,E}(BExp) = \{(\mathrm{t},\mathrm{f})\} &\oplus [\![\mathrm{LessThan}]\!]^\#_E(\{(0,0)\}, n^1_{2,E}(Start)) \\ &\oplus [\![\mathrm{And}]\!]^\#_E(n^1_{2,E}(BExp), n^1_{2,E}(BExp)) \end{aligned} \tag{10}$$

Because $n^1_{2,E}(BExp)$ has a finite domain, equations over such a domain can be solved iteratively, in this case yielding the fixed-point value $n^1_{2,E}(BExp) = \{(\mathrm{t},\mathrm{f}),(\mathrm{t},\mathrm{t}),(\mathrm{f},\mathrm{f})\}$. We now plug this solution into the equation for *Start* and compute the value of $n^2_{2,E}(Start)$ similarly to how we computed that of $n^1_{2,E}(Start)$. We then use $n^2_{2,E}(Start)$ to compute $n^2_{2,E}(BExp)$ and discover that $n^2_{2,E}(BExp) = n^2_{2,E}(BExp)$. 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 $n^1_{2,E}(Start)$ 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. (4) we get that no output set satisfies the formula on the given input examples, and therefore this SyGuS problem is unrealizable.

3 Background

In this section, we recall the definition of syntax-guided synthesis over a finite set of examples.

3.1 Trees and Tree Grammars

A *ranked alphabet* is a tuple (Σ, rk_Σ) where Σ is a finite set of symbols and $rk_\Sigma : \Sigma \longrightarrow \mathbb{N}$ associates a rank to each symbol. For every $m \geq 0$, 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., $\Sigma = \{Plus^{(2)}, Var(x)^{(0)}\}$. (For brevity, the superscript is sometimes omitted.) 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 $\sigma^{(k)} \in \Sigma^{(k)}$ and $t_1, \ldots, t_k \in T_\Sigma$, then $\sigma^{(k)}(t_1, \cdots, t_k) \in T_\Sigma$. In what follows, we assume a fixed ranked alphabet (Σ, rk_Σ).

Definition 3.1 (Regular-Tree Grammar)**.** A *regular tree grammar* (RTG) is a tuple $G = (N, \Sigma, S, \delta)$, where N is a finite set of nonterminal symbols of arity 0; Σ is a ranked alphabet; $S \in N$ is an initial nonterminal; and δ is a finite set of productions of the form $A_0 \rightarrow \sigma^{(i)}(A_1, \ldots, A_i)$, where for $1 \leq j \leq i$, each $A_j \in N$ is a nonterminal.

Given a tree $t \in T_{\Sigma \cup N}$, applying a production $r = A \rightarrow \beta$ 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 nonterminal S. $\delta_A \subseteq \delta$ denotes the set of productions associated with nonterminal A, and $\Sigma_A := \{\sigma^{(i)} \mid A \rightarrow \sigma^{(i)}(A_1, ..., A_i) \in \delta_A\}$.

3.2 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, $\psi(f(\bar{x}), \bar{x})$, 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 terms 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.[4]

A SyGuS problem is ***realizable*** if there exists an 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 [6])**.** *Given a SyGuS problem sy, it is undecidable to check whether sy is realizable.*

Many SyGuS solvers do not solve the problem of finding a term that satisfies the specification on *all* inputs. Instead, they look for an expression that satisfies the specification on a *finite* example set E. If such a term is found, it is then checked if it can be generalized to all inputs. We take a similar approach to show unrealizability.

Definition 3.4. Given a SyGuS problem $sy = (\psi(f, \bar{x}), G)$ and a finite set of inputs $E = \langle i_1, \ldots, i_n \rangle$, let $sy^E := (\psi^E(f), G)$ denote the problem of finding a term $e \in L(G)$ such that $[\![e]\!]$ is only required to be correct on the examples in E. Let $[\![e]\!]_E$ denote the vector of outputs $\langle [\![e]\!](i_1), \ldots, [\![e]\!](i_n) \rangle$

[4]In this paper, we focus on *single-invocation* SyGuS problems for which the formula ψ only contains instances of the function f that are called on the input $\bar{x}$. We write $\psi(f, \bar{x})$ instead of $\psi(f(\bar{x}), \bar{x})$ for brevity.

($= \langle o_1, \ldots, o_n \rangle$) produced by e on E. A sy^E problem is ***realizable*** if $\psi^E(\llbracket e \rrbracket_E) \overset{\text{def}}{=} \bigwedge_{i_j \in E} \psi(\llbracket e \rrbracket(i_j), i_j)$ holds, and ***unrealizable*** otherwise.

Lemma 3.5 ([11]). *If sy^E is unrealizable then sy is unrealizable.*

Example 3.6. The regular tree grammar of all *linear integer arithmetic* (LIA) terms is

T_{LIA} ::= Plus($T_{\text{LIA}}, T_{\text{LIA}}$) | Minus($T_{\text{LIA}}, T_{\text{LIA}}$) | Num($c$) | Var($x$)

where $c \in \mathbb{Z}$, and $x \in \mathcal{V}$ is an input variable to the function being synthesized. The semantics of these productions is as expected, and is extended to terms in the usual way.

In the case of a sy^E instance, we consider the restricted semantics of LIA with respect to a set of examples $E = \langle i_1, \ldots, i_n \rangle$, given by a function $\llbracket \cdot \rrbracket_E : T_{LIA} \to \mathbb{Z}^n$. $\llbracket \cdot \rrbracket_E$ maps an LIA term to the corresponding output vector produced by evaluating the term with respect to all of the examples in E. Let $\mu_E : \mathcal{V} \to \mathbb{Z}^n$ be the function that projects the inputs onto the x coordinate—i.e., $\mu_E(x) = \langle i_1(x), \ldots, i_n(x) \rangle$. The semantics of the LIA operators with respect to an example set E is then defined as follows:

$$\llbracket \text{Plus} \rrbracket_E(\vec{v}_1, \vec{v}_2) := \vec{v}_1 + \vec{v}_2 \qquad \llbracket \text{Num}(c) \rrbracket_E := \langle c, \ldots, c \rangle$$
$$\llbracket \text{Minus} \rrbracket_E(\vec{v}_1, \vec{v}_2) := \vec{v}_1 - \vec{v}_2 \qquad \llbracket \text{Var}(x) \rrbracket_E := \mu_E(x)$$

where + (resp. −) denotes the component-wise addition (resp. subtraction) of two vectors. $\llbracket \cdot \rrbracket_E : T_{\text{LIA}} \to \mathbb{Z}^n$ is extended to terms in the usual way. For brevity, we overload the term "LIA" to refer both to the *logic* LIA and to LIA *grammars*—i.e., grammars over the alphabet {Plus, Minus, Num(c), Var(x)}.

In §4.3, we present an algorithm based on Counterexample-Guided Inductive Synthesis (CEGIS) to show unrealizability of a SyGuS problem, sy, by showing unrealizability of a sy^E problem. The idea is to check unrealizability of sy^E for some set E. If sy^E is unrealizable, the algorithm reports unrealizable, otherwise it generates a new example, i_{n+1}, adds it to $E' = E \cup \{i_{n+1}\}$, and tries to prove unrealizability of $sy^{E'}$, and so on. In §5, we show that the unrealizability problem for a sy^E instance is decidable for LIA grammars. However, we note that there are SyGuS problems for which CEGIS-style algorithms *cannot* prove unrealizability [11]. Despite this negative result, we will show that a CEGIS algorithm can prove unrealizability for many SyGuS instances (§8).

4 Proving Unrealizability using Grammar Flow Analysis

In this section, we present a formalism called *grammar flow analysis* (GFA) [17], which connects regular tree grammars to equation systems, and show how to use GFA to prove unrealizability of SyGuS problems for finitely many examples.

4.1 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 4.1 (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. 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 4.2. [GFA] [17, 19] 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

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

In a GFA problem $\mathcal{G} = (G, \mathcal{D})$, each production is associated with a *production function* $\llbracket \cdot \rrbracket^\#$ that provides an interpretation of g—i.e., $\llbracket g \rrbracket^\# : D^k \to D$. $\llbracket \cdot \rrbracket^\#$ is extended to trees in $L(G)$ in the usual way, by thinking of each tree $e \in L(G)$ as a term over the operations $\llbracket g \rrbracket^\#$. Term e denotes a composition of functions, and corresponds to a unique value in D, which we call $\llbracket e \rrbracket^\#_{\mathcal{G}}$ (or simply $\llbracket e \rrbracket^\#$ 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:

$$m_{\mathcal{G}}(X) = \bigoplus_{e \in L_G(X)} \llbracket e \rrbracket^\#_{\mathcal{G}}.$$

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

$$n_{\mathcal{G}}(X_0) = \bigoplus_{X_0 \to g(X_1, \ldots, X_k) \in \delta} \llbracket g \rrbracket^\#(n_{\mathcal{G}}(X_1), \ldots, n_{\mathcal{G}}(X_k)). \quad (11)$$

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 $m_{\mathcal{G}}(X) = \bigoplus_{e \in L(G)} \llbracket e \rrbracket^\#_{\mathcal{G}}$. 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

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

where J is a finite or infinite index set.

Theorem 4.3. *[17, 19] If every production function $[\![g]\!]^{\#}$, $g\in\Sigma$, is infinitely distributive in each argument position, then for all nonterminals X, $m_{\mathcal{G}}(X)=n_{\mathcal{G}}(X)$.*

This theorem is key to our decision procedures for LIA and CLIA grammars, because the domain of semi-linear sets has this property (§5.3).

4.2 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 4.4. 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 $\gamma\colon D\rightarrow Val^{|E|}$, 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

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 §2 based on semi-linear sets (see §5 and §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* [20] 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 $\widehat{\gamma}(\cdot,\vec{o}):\mathcal{D}\rightarrow\mathcal{L}_{\mathcal{D}}$ 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 4.5. *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*

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

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.*

4.3 Algorithm for Showing Unrealizability

Alg. 1 summarizes our strategy for showing unrealizability.

Example 4.6. Recall the SyGuS problem, from §2, of synthesizing a function $e_f(x)=2x+2$ using the grammar from Eqn. (1). Suppose that we call Alg. 1 with the example set $E=\{1\}$, and use the abstract domain of semi-linear sets. Alg. 1 first creates a GFA problem $\mathcal{G}_E$, which is shown as the recursive equation system given as Eqn. (3). 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 $\exists\lambda\geq 0.o_1=0+\lambda 3$. Then, on line (3) the LIA formula $\exists\lambda\geq 0.o_1=0+\lambda 3\wedge o_1=2i_1+2\wedge i_1=1$ 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

Algorithm 1: Checking whether sy^E is unrealizable

```
Function : CheckUnrealizable($G,\psi,E$)
Input    : Grammar $G$, specification $\psi$, set of examples $E$
1 $\mathcal{G}_E \leftarrow (G,\mathcal{D})$   // GFA problem from $G$ and $E$ (Def. 4.4) ;
2 $s \leftarrow n_{\mathcal{G}_E}(Start)$   // Compute solution to the GFA problem;
3 if $\widehat{\gamma}(s,\vec{o})\wedge\bigwedge_{i_j\in E}\psi(o_j,i_j)$ is unsatisfiable then
4     return Unrealizable
5 return $\begin{cases}\text{Realizable}, & \mathcal{G}_E \text{ is an exact abstraction}\\ \text{Unknown}, & \text{otherwise}\end{cases}$
```

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 Alg. 1 could be instantiated with Kleene iteration and predicate abstraction to attempt to show unrealizabilty, for arbitrary SyGuS problems. However, in this paper we are focused on SyGuS problems using integer arithmetic, which does have infinite ascending chains. Thus, while predicate abstraction, and other domains with finite height, 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 [7]. 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 $P_1(\vec{v}) \leq P_2(\vec{v})$ 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 [8]. In this case, Alg. 1 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 Alg. 1 would use the formula in line (3) as the Horn-clause query, instead of having a separate SMT check.

Example 4.7. The GFA problem in Eqn. (2) can be encoded using the following constrained Horn clause:

$$\forall v, v'.\ \mathit{Start}(v) \leftarrow (v = 1+1+1+v' \wedge \mathit{Start}(v')) \vee v = 0 \quad (12)$$

A Horn-clause solver can prove that the LIA SyGuS problem from §2 is unrealizable by showing that the following formula is unsatisfiable: Eqn. (12) $\wedge\ \mathit{Start}(o_1) \wedge o_1 = 2i_1 + 2$.

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 [10]. 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 (3) imply unrealizability on a set of examples 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.

5 Proving Unrealizability of LIA SyGuS Problems with Examples

In this section, we instantiate the framework underlying Alg. 1 to obtain a *decision procedure* for (un)realizability of SyGuS problems in *linear integer arithmetic* (LIA), where the specification is given by examples (as defined in Ex. 3.6). 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 (§5.1). 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* (§5.3). Finally, we show that semi-linear sets admit symbolic concretization (§5.4). Thus, by Thm. 4.5, we obtain a decision procedure for checking (un)realizability.

5.1 Solving Equations using Newton's Method

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

Definition 5.1. A *semiring* $\mathcal{S} = (D, \oplus, \otimes, \underline{0}, \underline{1})$ 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 has a *Kleene-star operator* ${}^{\circledast} : D \rightarrow D$ defined as follows: $a^{\circledast} = \oplus_{i \in \mathbb{N}}\, a^i$.

A semiring is *idempotent* if for all $a \in D$, $a \oplus a = a$.

Recently, Esparza et al. [10] developed an iterative method, called *Newtonian Program Analysis* (NPA), which solves a set of semiring equations by an iterative computation.

Lemma 5.2. *[Newton's Method [10, Theorem 7.7]] 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. 5.2 is a powerful result because it applies even in cases when the semiring has infinite ascending chains.

5.2 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$^+$,

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

where the semantics of the Plus, Num, and Var operators are the same as for LIA, and $[\![\mathrm{NegVar}(x)]\!]_E := -\mu_E(x)$. We say a regular-tree grammar is an LIA$^+$ grammar if its alphabet is $\{\text{Plus}, \text{Num}(c), \text{Var}(x), \text{NegVar}(x)\}$.

The next example shows how our algorithm uses a function h to push negations to the leaves of LIA terms to yield an LIA$^+$ grammar.

Example 5.3. Consider the LIA grammar G:

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

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

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

5.3 Grammar Flow Analysis using Semi-Linear Sets

Thanks to §5.2, 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* (see [10, §2.3.3] and [5, §3.4.4]). 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 5.4 (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

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

To apply Newton's method for solving equations (Lem. 5.2), we need a commutative idempotent semiring over semi-linear sets. Fortunately, such a semiring exists [5, §3.4.4], with the operators $\otimes$, $\oplus$ and $\circledast$ defined as follows:

$$\{\langle \vec{u}_{1,i}, V_{1,i}\rangle\}_i \oplus \{\langle \vec{u}_{2,j}, V_{2,j}\rangle\}_j = \{\langle \vec{u}_{1,i}, V_{1,i}\rangle\}_i \cup \{\langle \vec{u}_{2,j}, V_{2,j}\rangle\}_j$$

$$\{\langle \vec{u}_{1,i}, V_{1,i}\rangle\}_i \otimes \{\langle \vec{u}_{2,j}, V_{2,j}\rangle\}_j = \bigcup_{i,j} \{\langle \vec{u}_{1,i} + \vec{u}_{2,j}, V_{1,i} \cup V_{2,j}\rangle\}$$

$$(\{\langle \vec{u}_i, V_i\rangle\}_i)^{\circledast} = \{\langle \vec{0}, \bigcup_i (\{\vec{u}_i\} \cup V_i)\rangle\} \tag{13}$$

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

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

$$[\![\text{Plus}]\!]^\#_E(sl_1, sl_2) = sl_1 \otimes sl_2 \tag{14}$$

$$[\![\text{Num}(c)]\!]^\#_E = \{\langle\langle c, \cdots, c\rangle, \emptyset\rangle\} \tag{15}$$

$$[\![\text{Var}(x)]\!]^\#_E = \{\langle \mu_E(x), \emptyset\rangle\} \tag{16}$$

$$[\![\text{NegVar}(x)]\!]^\#_E = \{\langle -\mu_E(x), \emptyset\rangle\} \tag{17}$$

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. (14)–(17), 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. 3.6. In particular, $\oplus$ two lines above Eqn. (13) preserves singleton sets, and hence for singleton sets, $\otimes$ one line above Eqn. (13) 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 $\oplus$ ([10, Defn. 2.1 and §2.3.3]), $m_{\mathcal{G}_E}(X) = n_{\mathcal{G}_E}(X)$ holds by Thm. 4.3, 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

$$n_{\mathcal{G}_E}(X_0) = \bigoplus_{X_0 \to g(X_1, \ldots, X_k) \in \delta} [\![g]\!]^\#_E(n_{\mathcal{G}_E}(X_1), \ldots, n_{\mathcal{G}_E}(X_k)). \tag{18}$$

Example 5.5. Consider again the LIA$^+$ grammar G_1 from Eqn. (1), written out in the expanded form given in footnote 1. 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

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

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. (18) can be solved using Newton's method.

Proposition 5.6. $(\mathcal{SL}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ *is a commutative, idempotent, ω-continuous semiring.*

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 [15].

5.4 Checking Unrealizability

We now show how symbolic concretization for $\mathcal{SL}$ can be used to prove that no element $\vec{o}$ in $n_{\mathcal{G}}(Start)$ 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:

$$\widehat{\gamma}(\{\langle \vec{u}_i, V_i\rangle\}_i, \vec{o}) \overset{\text{def}}{=} \bigvee_i \widehat{\gamma}(\langle \vec{u}_i, V_i\rangle, \vec{o}). \tag{19}$$

Note that $\vec{o}$ is shared among all disjuncts.

Our decidability result follows directly from Thm. 4.5.

Theorem 5.7. *Given an LIA SyGuS problem sy and a finite set of examples E, it is decidable whether the SyGuS problem sy^E is realizable.*

6 Proving Unrealizability of CLIA SyGuS Problems with Examples

In this section, we instantiate the framework from §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 §5. The main difference is a technique for solving equations generated from grammars that involve both Boolean and integer operations.

6.1 Conditional Linear Integer Arithmetic

The grammar of all CLIA terms is the following:

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

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. The example grammar presented in Eqn. (5) in §2 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 $\text{PROJ}_{\vec{\mathbb{Z}}}(\vec{v}, b)$ 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:

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

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

$$\begin{array}{c} [\![\text{IfThenElse}]\!]_E(\vec{b}, \vec{v}_1, \vec{v}_2) = \text{PROJ}_{\vec{\mathbb{Z}}}(\vec{v}_1, \vec{b}) + \text{PROJ}_{\vec{\mathbb{Z}}}(\vec{v}_2, \neg\vec{b}) \\ [\![\text{Not}]\!]_E(\vec{b}) = \neg\vec{b} \qquad [\![\text{And}]\!]_E(\vec{b}_1, \vec{b}_2) = \vec{b}_1 \wedge \vec{b}_2 \\ [\![\text{LessThan}]\!]_E(\vec{v}_1, \vec{v}_2) = \vec{v}_1 < \vec{v}_2 \end{array}$$

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

Similarly to what we did in §5.2, 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 (§6.2). Second, we show how to compute an exact abstraction of Boolean nonterminals in grammars without IfThenElse (§6.3). Third, we show how to solve SyGuS problems with CLIA grammars containing arbitrary operators, in particular IfThenElse and mutual recursion (§6.4).

6.2 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 $\text{PROJ}_{\mathcal{SL}}(sl, \vec{b})$ be the semi-linear set obtained by zeroing out elements at all index positions for which $\vec{b}$ is false:

$$\begin{array}{rcl} \text{PROJ}_{\mathcal{SL}}(\{\langle \vec{u}_i, \Omega_i\rangle\}_i, \vec{b}) & = & \{\text{PROJ}_{\mathcal{S}}(\langle \vec{u}_i, \Omega_i\rangle, \vec{b})\}_i \\ \text{PROJ}_{\mathcal{S}}(\langle \vec{u}, \{\vec{v}_1, ..., \vec{v}_n\}\rangle, \vec{b}) & = & \langle \text{PROJ}_{\vec{\mathbb{Z}}}(\vec{u}, \vec{b}), \{\text{PROJ}_{\vec{\mathbb{Z}}}(\vec{v}_i, \vec{b})\}_i\rangle \end{array}$$

Next, we lift the concrete semantics to semi-linear sets and define the abstract semantics of CLIA operators.

$$\begin{array}{l} [\![\text{IfThenElse}]\!]^\#_E(bset, sl_1, sl_2) = \\ \qquad\qquad \bigoplus_{\vec{b} \in bset} \text{PROJ}_{\mathcal{SL}}(sl_1, \vec{b}) \otimes \text{PROJ}_{\mathcal{SL}}(sl_2, \neg\vec{b}) \\ \qquad [\![\text{LessThan}]\!]^\#_E(sl_1, sl_2) = \{v_1 {<} v_2 \mid v_1 \in sl_1, v_2 \in sl_2\} \\ \qquad\qquad [\![\text{Not}]\!]^\#_E(bset) = \bigcup_{\vec{b} \in bset}\{\neg\vec{b}\} \\ \qquad [\![\text{And}]\!]^\#_E(bset_1, bset_2) = \bigcup_{\vec{b}_1 \in bset_1, \vec{b}_2 \in bset_2}\{\vec{b}_1 \wedge \vec{b}_2\} \end{array}$$

Operationally, the semantics of the LessThan symbol can be implemented using an SMT solver. As shown in §5.4, a semi-linear set sl can be symbolically concretized as a formula $\widehat{\gamma}(sl, \vec{o})$ in LIA (a decidable SMT theory). Therefore, the set $[\![\text{LessThan}]\!]^\#_E(sl_1, sl_2) = bset$ can be computed by performing $2^{|E|}$ SMT queries—i.e., for every Boolean vector $\vec{b} = \langle b_1, \ldots, b_{|E|}\rangle$, we have that $\vec{b} \in bset$ iff the following formula is satisfiable: $\widehat{\gamma}(sl_1, \vec{o}_1) \wedge \widehat{\gamma}(sl_2, \vec{o}_2) \wedge \vec{b} = \vec{o}_1 < \vec{o}_2$.

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

6.3 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 §5.1, and then plugging the corresponding values into the equations that involve Boolean nonterminals. After this step, we are left with a set of equations $eqs_{\mathbb{B}}$ that involve only Boolean nonterminals and Boolean symbols. Because the domain of sets of Boolean vectors is finite, the least fixed point of $eqs_{\mathbb{B}}$ can be found using an algorithm that iteratively computes finer under-approximations of $n_{\mathcal{G}_E^{\text{CLIA+}}}$ as $n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}$—i.e., the under-approximation at iteration k—until it reaches the least fixed point, which—by Thm. 4.3—is an exact abstraction. 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.

6.4 CLIA Equations with Mutual Recursion

We have seen how to compute exact abstractions for grammars without mutual recursion, for both integer (§5.3) and Boolean (§6.3) 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. (7) in §2 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 §6.3. 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 under-approximation $n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(X)$ of $n_{\mathcal{G}_E^{\text{CLIA+}}}(X)$. Initially, $n^{-1}_{\mathcal{G}_E^{\text{CLIA+}}}(X) = \mathbf{0}$ for all nonterminals X of type $\mathbb{Z}$. At iteration $\mathsf{k} \geq 0$ the algorithm does the following:

Step 1. Replace each integer nonterminal Z with the value $n^{\mathsf{k-1}}_{\mathcal{G}_E^{\text{CLIA+}}}(Z)$ from iteration k-1 and use the technique in §6.3 to compute $n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(B)$ for each Boolean nonterminal B.

Step 2. Replace each Boolean nonterminal B with the value $n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(B)$ from ***Step 1*** and compute $n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(Z)$ for each integer nonterminal Z (see Eqn. (8) in §2 for an example).

The equations obtained at ***Step 2*** only contain integer nonterminals, but they may contain IfThenElse symbols for which the abstract semantics is not directly supported by the equation-solving technique presented in §5.1. 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 iterative algorithm SolveMutual is guaranteed to terminate in $|N|2^{|E|}$ iterations.

$[\![\text{IfThenElse}]\!]^{\#}_E$ 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:

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

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 $n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(X^b)$, so that the solution to the set of equations for this variable is $\text{PROJ}_{\mathcal{SL}}(n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(X), b)$.

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 eqs' over the set of variables $N^{\mathbb{B}^d}$ as follows. For every equation $n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(X) = \bigoplus_i \alpha_i$ in eqs and $b \in \mathbb{B}^d$, there exists an equation $n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(X^b) = \bigoplus_i \pi_b(\alpha_i)$ 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 $\text{PROJ}_{\mathcal{SL}}(n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(X), b')/n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(X^{b \wedge b'})$.
2. For every $X \in N$, π_b applies $n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(X)/n^{\mathsf{k}}_{\mathcal{G}_E^{\text{CLIA+}}}(X^b)$.
3. For any semi-linear set sl appearing in eqs, π_b applies the substitution $sl/\text{PROJ}_{\mathcal{SL}}(sl, b)$. Because sl is a constant, this substitution yields a constant semi-linear set.

Example 6.1. Figure 1 illustrates how Eqn. (8) is rewritten into Eqns. (9). We omit equations for variables $n^1_{2,E}(Start^{\{\mathsf{f},\mathsf{f}\}})$ and $n^1_{2,E}(Start^{\{\mathsf{t},\mathsf{f}\}})$ because they do not contribute to the solving of $n^1_{2,E}(Start^{\{\mathsf{t},\mathsf{t}\}})$. After expanding the definition of $[\![\text{IfThenElse}]\!]^{\#}$, we apply the substitutions to obtain Eqns. (9). Substitution 2 is not applied because there are no variables of the form $n^1_{2,E}(X)$ after applying substitution 1.

6.5 Checking Unrealizability

Using the symbolic-concretization technique described in §5.4 , and the complexities described throughout this section, we obtain the following decidability theorem.

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

$$\Downarrow \text{ Generate equations for } Start^b$$

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

$$\Downarrow \text{ Expand definition of } [\![\text{IfThenElse}]\!]^\#$$

$$\begin{aligned} n^1_{2,E}(Start^{(\text{t},\text{t})}) = {} & \pi_{\{\text{t},\text{t}\}}\big(\textsc{proj}_{\mathcal{SL}}(\{(0,0)+\lambda(3,6)\}, \{\text{f},\text{t}\})\big) \\ & \otimes \pi_{\{\text{t},\text{t}\}}\big(\textsc{proj}_{\mathcal{SL}}(n^1_{2,E}(Start), (\text{f},\text{t}))\big) \\ & \oplus \pi_{\{\text{t},\text{t}\}}\big(\{(0,0)+\lambda(2,4)\}\big) \oplus \pi_{\{\text{t},\text{t}\}}\big(\{(0,0)+\lambda(3,6)\}\big) \\ n^1_{2,E}(Start^{(\text{f},\text{t})}) = {} & \pi_{\{\text{f},\text{t}\}}\big(\textsc{proj}_{\mathcal{SL}}(\{(0,0)+\lambda(3,6)\}, \{\text{f},\text{t}\})\big) \\ & \otimes \pi_{\{\text{f},\text{t}\}}\big(\textsc{proj}_{\mathcal{SL}}(n^1_{2,E}(Start), (\text{f},\text{t}))\big) \\ & \oplus \pi_{\{\text{f},\text{t}\}}\big(\{(0,0)+\lambda(2,4)\}\big) \oplus \pi_{\{\text{f},\text{t}\}}\big(\{(0,0)+\lambda(3,6)\}\big) \end{aligned}$$

$$\Big\Downarrow \begin{array}{l} \text{Apply } \textsc{proj}_{\mathcal{SL}} \text{ to constants} \\ \text{Apply substitution 1} \end{array}$$

$$\begin{aligned} n^1_{2,E}(Start^{(\text{t},\text{t})}) = {} & \pi_{\{\text{t},\text{t}\}}\big(\{(0,0)+\lambda(3,0)\}\big) \otimes n^1_{2,E}(Start^{(\text{t},\text{t})\wedge(\text{f},\text{t})}) \\ & \oplus \pi_{\{\text{t},\text{t}\}}\big(\{(0,0)+\lambda(2,4)\}\big) \oplus \pi_{\{\text{t},\text{t}\}}\big(\{(0,0)+\lambda(3,6)\}\big) \\ n^1_{2,E}(Start^{(\text{f},\text{t})}) = {} & \pi_{\{\text{f},\text{t}\}}\big(\{(0,0)+\lambda(3,0)\}\big) \otimes n^1_{2,E}(Start^{(\text{f},\text{t})\wedge(\text{f},\text{t})}) \\ & \oplus \pi_{\{\text{f},\text{t}\}}\big(\{(0,0)+\lambda(2,4)\}\big) \oplus \pi_{\{\text{f},\text{t}\}}\big(\{(0,0)+\lambda(3,6)\}\big) \end{aligned}$$

$$\Downarrow \text{ Apply substitution 3}$$

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

Figure 1. Rewriting Eqn. (8) into Eqns. (9).

Theorem 6.2. *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.*

7 Implementation

We implemented 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 [3]), which verifies the correctness of candidate solutions and produces counterexamples, 2) a synthesizer (ESolver—the enumerative solver introduced in [2]), which synthesizes solutions from examples, and 3) an unrealizability verifier, which proves whether the problem is unrealizable on the current set of examples.

Alg. 2 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 (1)), and then, in parallel, ❶ calls ESolver to find a solution of sy^E (line (4)), and ❷ uses grammar flow analysis (Alg. 1) to decide whether $sy^{E \cup E_r}$ is unrealizable (line (11)), 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 (16)). 2) If GFA returns realizable, nay adds a temporary random example to E_r (line (18)), 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 (6)): if not, nay adds a counterexample to E (line (7)) and triggers the next CEGIS iteration, otherwise, nay return e^* as a solution to the given SyGuS problem sy (line (10)).

nay currently has two modes: nay_{Horn} and $\text{nay}_{\mathcal{SL}}$.

nay_{Horn} implements the constrained-Horn-clauses technique for solving equations presented in §4.3, and uses Z3's Horn-clause solver, Spacer [8], to solve the Horn clauses.

$\text{nay}_{\mathcal{SL}}$ implements the decision procedures presented in §5 and §6 for solving LIA and CLIA problems. $\text{nay}_{\mathcal{SL}}$ also implements two optimizations: (i) $\text{nay}_{\mathcal{SL}}$ eagerly removes a linear set from a semi-linear set whenever it is trivially subsumed by another linear set; and (ii) $\text{nay}_{\mathcal{SL}}$ uses the optimization presented in the following paragraph.

Algorithm 2: CEGIS with random examples

```
   Function: Nay($G, \psi$)
   Input: Grammar $G$, specification $\psi$
1  $i \leftarrow$ Random($-50, 50$)    Set of examples $E \leftarrow \{i\}$
2  while True do
3      do in parallel
4          ❶ {$e^* \leftarrow$ESolver($G, \psi, E$)
5              kill ❷
6              if $\exists i_{cex}.\neg\psi([\![e^*]\!], i_{cex})$ then
7                  $E \leftarrow E \cup \{i_{cex}\}$
8                  continue
9              else
10                 return $e^*$ }
11         ❷ { $E_r \leftarrow \emptyset$
12             while True do
13                 $result \leftarrow$CheckUnrealizable($G, \psi, E \cup E_r$)
14                 if $result$ =Unrealizable then
15                     kill ❶
16                     return Unrealizable
17                 $i \leftarrow$ Random($-50, 50$)
18                 $E_r \leftarrow E_r \cup \{i\}$
19                 continue }
```

Solving GFA Equations via Stratification. The $n_{\mathcal{G}}$ equations (Eqn. (11)) 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 $X_0 \rightarrow g(X_1, X_2) \mid h(X_2, X_3)$, then the dependence graph has three edges into node X_0: $X_1 \rightarrow X_0$, $X_2 \rightarrow X_0$, and $X_3 \rightarrow X_0$. 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.

8 Evaluation

In this section, we evaluate the effectiveness and performance of NAY$_{\mathcal{SL}}$ and NAY$_{\text{Horn}}$.[5]

Benchmarks. We perform our evaluation using 132 variants of the 60 CLIA benchmarks from the CLIA SyGuS competition track [2]. These benchmarks are the same ones used in the evaluation of the tool we compare against, NOPE [11], which like NAY only supports LIA and CLIA SyGuS problems.

The benchmarks are divided into three categories, and arise from a tool used to synthesize terms in which a certain syntactic feature appears a minimal number of times [13]. LimitedPlus (resp. LimitedIf) contains 30 (resp. 57) benchmarks in which the grammar bounds the number of times a Plus (resp. IfThenElse) operator can appear in an expression-tree to be one less than the number required to solve the original synthesis problem. LimitedConst contains 45 benchmarks that restrict what constants appear in the grammar. In each of the benchmarks, the grammar that specifies the search space generates infinitely many terms.

8.1 Effectiveness of NAY

EQ 1. How effective is NAY at proving unrealizability?

Table 1. Performance of NAY and NOPE for LimitedIf and LimitedPlus benchmarks.[6] The table shows the number of nonterminals ($|N|$), productions ($|\delta|$), and variables ($|V|$) in the problem grammar; the number of examples required to prove unrealizability ($|E|$); and the average running time of NAY$_{\mathcal{SL}}$, NAY$_{\text{Horn}}$, and NOPE. ✗ denotes a timeout.

	Problem	Grammar			$\lvert E\rvert$	time (s)		
		$\lvert N\rvert$	$\lvert\delta\rvert$	$\lvert V\rvert$		NAY$_{\mathcal{SL}}$	NAY$_{\text{Horn}}$	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

We compare NAY$_{\mathcal{SL}}$ and NAY$_{\text{Horn}}$ against NOPE, the state-of-the-art tool for proving unrealizability of SyGuS problems [11]. 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 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$_{\mathcal{SL}}$ and NOPE, that iteration is the one that dominates the runtime. On average, it accounts for 60.4% of the running time for NAY$_{\mathcal{SL}}$ and 90.3% for NOPE, but only 8.3% for NAY$_{\text{Horn}}$. (For NAY$_{\text{Horn}}$, counterexample generation is the most costly step.) The LimitedConst benchmarks could be solved by all tools, and results are given in the supplementary material.

[5] All the experiments were performed on an Intel Core i7 4.00GHz CPU, with 32GB of RAM. We used version 1.8 of CVC4 and commit d37c50e of ESolver. The timeout for each individual NAY/ESolver call is set at 10 minutes.

[6] We discovered that three of the benchmarks from [11] were actually realizable (marked with *). Because these benchmarks were created by bounding the number of Plus operators, we further reduced the bound by one to make them unrealizable.

Findings. NAY$_{\mathcal{SL}}$ solved 70/132 benchmarks, with an average running time of 1.97s.NAY$_{\text{Horn}}$ 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$_{\mathcal{SL}}$ can solve 11 LimitedPlus benchmarks that NOPE cannot solve. These benchmarks involve large grammars, a known weakness of NOPE (see [11]). 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$_{\mathcal{SL}}$ 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$_{\mathcal{SL}}$ 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$_{\mathcal{SL}}$ depends heavily on the number of examples, NAY$_{\mathcal{SL}}$ only solves benchmarks that require at most 4 examples. §8.2 analyzes the effect of the number of examples on NAY$_{\mathcal{SL}}$'s performance. When NAY$_{\mathcal{SL}}$ 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$_{\text{Horn}}$ 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$_{\text{Horn}}$. NAY$_{\text{Horn}}$ 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$_{\text{Horn}}$ is on average 19 times faster than NOPE. On benchmarks for which NOPE took more than 2 seconds, NAY$_{\text{Horn}}$ is 82x faster than NOPE (computed as the geometric mean).

The reason we use random examples in Alg. 2 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$_{\mathcal{SL}}$ 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 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$_{\mathcal{SL}}$ 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$_{\mathcal{SL}}$ solved, ESolver is 87% faster than NAY$_{\mathcal{SL}}$ calculated as a geometric mean.

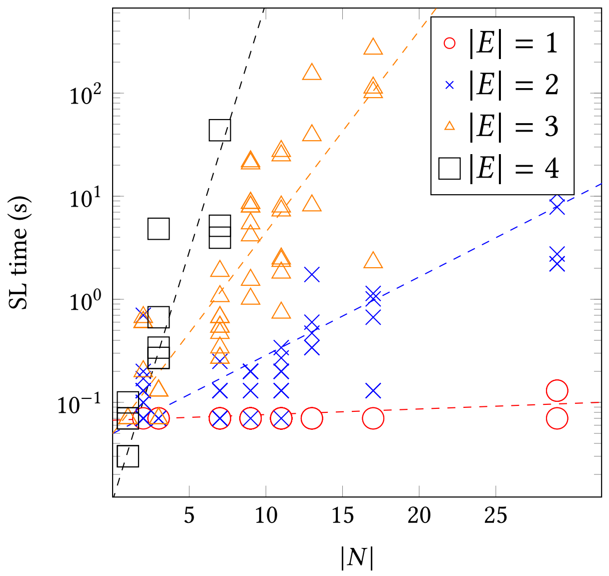

Figure 2. Time to compute semi-linear set vs. $|N|$.

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.*

8.2 The Cost of Proving Unrealizability

EQ 2. How does the size of the grammar and the number of examples affect the performance of different solvers?

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

NAY$_{\text{Horn}}$ and NOPE (shown in Fig. 3 and Fig. 4, 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$_{\mathcal{SL}}$ grows exponentially with respect to $|N|2^{|E|}$, and the running time of NAY$_{\text{Horn}}$ and NOPE grows exponentially with respect to $|E|$.

8.3 Effectiveness of Grammar Stratification

EQ 3. Is the stratification optimization from §7 effective?

Finding. Using stratification, NAY$_{\mathcal{SL}}$ can compute the semi-linear sets for 9 benchmarks for which NAY$_{\mathcal{SL}}$ 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.

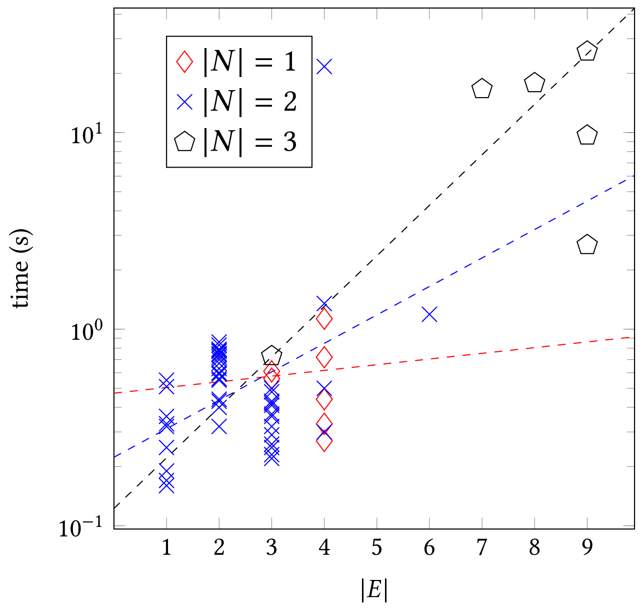

Figure 3. Running time of NAY$_{\text{Horn}}$ vs. number of examples.

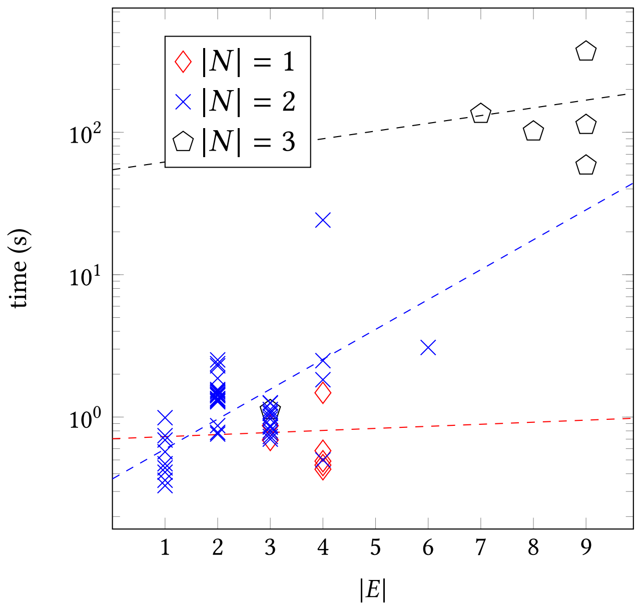

Figure 4. Running time of NOPE vs. number of examples.

9 Related Work

Unrealizability in SyGuS. Several SyGuS solvers compete in yearly SyGuS competitions [2], and can produce solutions to SyGuS problems when a solution exists. If the problem is unrealizable, these solvers only terminate if the language of the grammar is finite or contains finitely many functionally distinct programs, which is not the case in our benchmarks.

NOPE [11], the tool we compare against in §8, is the only tool that can prove unrealizability for non-trivial SyGuS problems. NOPE reduces the problem of proving unrealizability to one of proving unreachability in a recursive non-deterministic program, and uses off-the-shelf verifiers to solve the unreachability problem. Unlike NAY, NOPE does not provide any insights into how we can devise specialized techniques for solving unrealizability, because NOPE reduces a constrained SyGuS problem to a full-fledged program-reachability problem. In contrast, the approach presented in this paper gives a characterization of unrealizability in terms of solving a set of equations. Using the equation-solving framework, we provided the first *decision procedures* for LIA and CLIA SyGuS problems over examples. Moreover, the equation-based approach allows us to use known equation-solving techniques, such as Newton's method and constrained Horn clauses.

Unrealizability in Program Synthesis. For certain synthesis problems—e.g., reactive synthesis [4]—realizability is decidable. However, SyGuS is orthogonal to such problems.

Mechtaev et al. [16] propose to use unrealizability to prune irrelevant paths in symbolic-execution engines. The synthesis problems generated by Mechtaev et al. are not directly expressible in SyGuS. Moreover, these problems are decidable because they can be encoded as SMT formulas.

Abstractions in Program Synthesis. SYNGAR [22] 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 in which the set of values $n_\mathcal{G}(X)$ is based on predicate abstraction (see §4.3). SYNGAR's approach is tied to finite abstract domains, while our equational approach extends to infinite domains—e.g., semi-linear sets—because it does not specify how the equations must be solved.

Acknowledgments

Supported, in part, by a gift from Rajiv and Ritu Batra; by ONR under grants N00014-17-1-2889 and N00014-19-1-2318; by NSF under grants 1763871 and 1750965; and by a Facebook fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 1–8.

[2] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2016. SyGuS-Comp 2016: results and analysis. *arXiv preprint arXiv:1611.07627* (2016).

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 171–177.

[4] Roderick Bloem. 2015. Reactive Synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)* (Austin, Texas). 3–3.

[5] A. Bouajjani, J. Esparza, and T. Touili. 2003. A Generic Approach to the Static Analysis of Concurrent Programs with Procedures. In *Princ. of Prog. Lang.*

[6] Benjamin Caulfield, Markus N. Rabe, Sanjit A. Seshia, and Stavros Tripakis. 2015. What's Decidable about Syntax-Guided Synthesis? *arXiv preprint arXiv:1510.08393* (2015).

[7] P. Cousot and N. Halbwachs. 1978. Automatic Discovery of Linear Constraints Among Variables of a Program. In *Princ. of Prog. Lang.*

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. 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* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[9] Hassan Eldib, Meng Wu, and Chao Wang. 2016. Synthesis of Fault-Attack Countermeasures for Cryptographic Circuits. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 343–363.

[10] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2010. Newtonian program analysis. *J. ACM* 57, 6 (2010), 33:1–33:47.

[11] Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving Unrealizability for Syntax-Guided Synthesis. In *International Conference on Computer Aided Verification (CAV)*. Springer-Verlag.

[12] Qinheping Hu and Loris D'Antoni. 2017. Automatic program inversion using symbolic transducers. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*. 376–389.

[13] Qinheping Hu and Loris D'Antoni. 2018. Syntax-Guided Synthesis with Quantitative Syntactic Objectives. In *Computer Aided Verification - 30th International Conference, (CAV)*. 386–403.

[14] J.B. Kam and J.D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Inf.* 7, 3 (1977), 305–318.

[15] Eryk Kopczynski and Anthony Widjaja To. 2010. Parikh images of grammars: Complexity and applications. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 80–89.

[16] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. 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)*. 389–399.

[17] U. Möncke and R. Wilhelm. 1991. Grammar Flow Analysis. In *Attribute Grammars, Applications and Systems, (Int. Summer School SAGA)*. 151–186.

[18] Rohit J. Parikh. 1966. On Context-Free Languages. *J. ACM* 13, 4 (Oct. 1966), 570–581. https://doi.org/10.1145/321356.321364

[19] G. Ramalingam. 1996. *Bounded Incremental Computation*. Springer-Verlag.

[20] T. Reps, M. Sagiv, and G. Yorsh. 2004. Symbolic implementation of the best transformer. In *VMCAI*.

[21] M. Sharir and A. Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall.

[22] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program synthesis using abstraction refinement. *PACMPL* 2, POPL (2018), 63:1–63:30.