Semantics-Guided Synthesis

JINWOO KIM, University of Wisconsin-Madison, USA
QINHEPING HU, University of Wisconsin-Madison, USA
LORIS D'ANTONI, University of Wisconsin-Madison, USA
THOMAS REPS, University of Wisconsin-Madison, USA

This paper develops a new framework for program synthesis, called *semantics-guided synthesis* (SemGuS), that allows a user to provide both the syntax and the semantics for the constructs in the language. SemGuS accepts a recursively defined big-step semantics, which allows it, for example, to be used to specify and solve synthesis problems over an imperative programming language that may contain loops with unbounded behavior. The customizable nature of SemGuS also allows synthesis problems to be defined over a non-standard semantics, such as an abstract semantics. In addition to the SemGuS framework, we develop an algorithm for solving SemGuS problems that is capable of both synthesizing programs and proving unrealizability, by encoding a SemGuS problem as a proof search over Constrained Horn Clauses: in particular, our approach is the first that we are aware of that can prove unrealizabilty for synthesis problems that involve imperative programs with unbounded loops, over an infinite syntactic search space. We implemented the technique in a tool called MESSY, and applied it to SyGuS problems (i.e., over expressions), synthesis problems over an imperative programming language, and synthesis problems over regular expressions.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; **Programming logic**; • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Program Synthesis, Semantics-Guided Synthesis (SemGuS), Unrealizability

ACM Reference Format:
Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-Guided Synthesis. *Proc. ACM Program. Lang.* 5, POPL, Article 30 (January 2021), 32 pages. https://doi.org/10.1145/3434311

1 INTRODUCTION

Program synthesis refers to the task of finding a program within a given search space that meets a given behavioral specification (typically a logical formula or a set of input-output examples). Program synthesis has been studied from a variety of perspectives, which have led to great practical advances in specific domains [Feser et al. 2015; Gulwani 2011; Phothilimthana et al. 2019].

The proliferation of domain-specific synthesis tools has led to numerous attempts to build frameworks that allow one to define and solve synthesis problems in a general fashion. Tools such as Sketch [Solar-Lezama 2013] and Rosette [Torlak and Bodík 2014] have introduced the notion of a solver-aided language, which allows one to define a synthesis problem using a specialized language and then solve the specified problem using a constraint solver. To retain the ability to solve practical problems, these tools have restricted their languages in ways that enable the use of

Authors' addresses: Jinwoo Kim, University of Wisconsin-Madison, USA, pl@cs.wisc.edu; Qinheping Hu, University of Wisconsin-Madison, USA, qhu28@wisc.edu; Loris D'Antoni, University of Wisconsin-Madison, USA, loris@cs.wisc.edu; Thomas Reps, University of Wisconsin-Madison, USA, reps@cs.wisc.edu.

2475-1421/2021/1-ART30
https://doi.org/10.1145/3434311

constraint-based synthesis methods—e.g., Sketch and Rosette do not allow arbitrary search spaces involving programs of unbounded size.

While solver-aided languages made synthesis more "programmable", their mutual incompatibility and language restrictions led to a natural question: *Can we define synthesis problems in a language-agnostic way?* This question was partly answered by the framework of of *syntax-guided synthesis* (SyGuS) [Alur et al. 2013], which provides a logical framework for defining synthesis problems. In a SyGuS problem, the search space is described using a context-free grammar of terms from a given theory, and the behavioral specification is expressed using a formula in that same logical theory. The unified logical format offered by SyGuS spurred researchers to design synthesizers that could solve problems defined in the SyGuS format [Alur et al. 2017b; Reynolds et al. 2015], and these solvers compete annually in SyGuS competitions [Alur et al. 2017a]. However, SyGuS introduced its own limitation: namely, that the semantics of SyGuS problems are limited to those from a fixed theory, such as linear integer arithmetic (LIA) or bitvectors. This limitation has created a gap between the two approaches: solver-aided languages are unable to express SyGuS problems with infinite search spaces, while SyGuS cannot express problems with semantics outside of a supported theory, such as imperative programs containing loops (which could be modeled using tools like Sketch and Rosette).

The SemGuS Framework. In this paper, we bridge this gap and present a new synthesis framework, called *semantics-guided synthesis* (SemGuS), that attempts to encompass and generalize the two approaches. Like SyGuS, the goal of SemGuS is to provide a *general, logical framework* that expresses the *core computational problem* of program synthesis [Alur et al. 2013], without being tied to a specific solution or implementation. However, in addition to a syntactic search space and a behavioral specification, SemGuS also allows the user to define the *semantics* of constructs in the grammar in terms of a set of inference rules—hence the name "semantics-guided synthesis".

By a *framework*, we mean the conceptual underpinnings that allows one to build a tool to automate the creation of solutions for problems in some domain. The canonical example is the theory of parsing, which provides the underpinnings of the tool `yacc` [Johnson 1975], which automates the construction of parsers.

For example, consider the problem that `yacc` addresses.

- An instance of a parsing problem, Parse(L,s), has two parameters: L, a context-free language; and s, a string to be parsed. String s changes more frequently than language L.
- Context-free grammars are a formalism for specifying context-free languages.
- Create a tool that implements the following specification:
  - Input: a context-free grammar that describes language L.
  - Output: a parser, `yyparse()`, such that invoking `yyparse()` on s computes Parse(L,s).

Thus, a *framework for synthesis* should follow a similar scheme.

- An instance of a synthesis problem Synthesize($\mathcal{L}$, $[\![\cdot]\!]_{\mathcal{L}}$, φ) has three parameters: $\mathcal{L}$, a formal language; $[\![\cdot]\!]_{\mathcal{L}}$, a semantics to ascribe to $\mathcal{L}$; and φ, a behavioral specification for some desired member of $\mathcal{L}$. The behavioral specification φ changes more frequently than $\mathcal{L}$ and $[\![\cdot]\!]_{\mathcal{L}}$.
- Let F_{syntax} and $F_{\text{semantics}}$ be appropriate formalisms for specifying $\mathcal{L}$ and $[\![\cdot]\!]_{\mathcal{L}}$, respectively.
- Create a tool that implements the following specification:
  - Input: an F_{syntax} specification of a language's syntax, and an $F_{\text{semantics}}$ specification of the language's semantics.
  - Output: a function $\text{Synth}_{\mathcal{L},[\![\cdot]\!]_{\mathcal{L}}}(\cdot)$ such that $\text{Synth}_{\mathcal{L},[\![\cdot]\!]_{\mathcal{L}}}(\varphi)$ computes Synthesize($\mathcal{L}$, $[\![\cdot]\!]_{\mathcal{L}}$, φ).

As in SyGuS, the formalism F_{syntax} used in SemGuS is regular-tree grammars. In SemGuS, $F_{\text{semantics}}$ is Constrained Horn Clauses, which are a class of logical formulas that are expressive enough to define a recursive big-step semantics. (In contrast, SyGuS has no explicit formalism $F_{\text{semantics}}$; instead, it relies on a shallow embedding into some decidable theory.)

The flexibility of Constrained Horn Clauses allows SemGuS to address synthesis problems for imperative programming languages. In §2, we show how the semantics of assignments and while loops can be defined in SemGuS. The customizable aspect of the semantics also provides a natural way to define synthesis problems over an alternative semantics (see §4). In essence, SemGuS extends the "logical framework" of SyGuS towards semantics, resulting in a framework that is capable of defining SyGuS problems, as well as problems that currently require a solver-aided language.

Solving SemGuS Problems. Following the definition of the SemGuS framework, this paper develops a method for solving general SemGuS problems capable of producing *two-sided answers* to a problem: either *synthesizing a solution*, or proving that the problem is *unrealizable*, i.e., has no solution. Proving the unrealizability of synthesis problems has applications in synthesizing programs that are optimized with respect to some metric [Hu and D'Antoni 2018], and can be employed in tandem with general synthesis algorithms as well. However, existing program synthesizers are generally unable to prove unrealizability, and focus only on synthesizing terms.

Although SemGuS can be used for much more than imperative program synthesis, solving SemGuS problems over an imperative programming language illustrates many of the challenges in computing solutions to general SemGuS problems:

Reasoning while lacking a direct background theory. Unlike SyGuS, in which problems are defined over decidable theories, such as linear integer arithmetic, SemGuS over an imperative programming language must deal with factors such as state, and there is typically no decidable theory of the programming language involved.

Loops. Loops provide a double challenge in the context of program synthesis: each loop could have (i) an infinite number of syntactic elaborations (of the condition and the loop-body), each of which may execute for (ii) an arbitrary number of iterations. Thus, a synthesis algorithm must reason about *sets* of loop-body elaborations instead of *individual* ones—otherwise, the search space becomes intractable. Existing constraint-based methods often deal with loops by setting an unrolling bound, which is a factor that limits the kinds of synthesis problems they can define or solve: SemGuS explicitly avoids this approach.

In §2 and §5, we show that an entire SemGuS problem—syntax, semantics, and behavioral specification—can be encoded using CHCs, effectively reducing program synthesis into a proof-search problem that can be solved with off-the-shelf CHC solvers, such as Z3 [De Moura and Bjørner 2008].[1] If a proof for the specification exists within the CHC-encoded syntax and semantic rules, the SemGuS problem is realizable, and the proof identifies a specific term satisfying the specification. If, on the other hand, the solver can prove that the specification is unsatisfiable using the given rules, then the problem is unrealizable. SemGuS is semantics-guided not only in the sense that it accepts a semantics, but in this proof-search step as well: among the lemmas established during the proof search (by an external solver), some may involve the semantics supplied to SemGuS by the user.

Contributions. This paper makes the following contributions:

- The SemGuS framework, which allows the user to supply inference rules that specify the syntax and semantics of the target language. In particular, the SemGuS framework can be used to specify synthesis problems over an imperative programming language (§4).

[1]In general, the problem of finding a solution to a set of CHCs is undecidable.

$$\begin{array}{rcl} \mathit{Start} & ::= & \text{while } B \text{ do } S \; ❶ \\ B & ::= & E < E \; ❷ \\ S & ::= & S; S \; ❸ \;\mid x := E \; ❹ \;\mid y := E \; ❺ \\ E & ::= & x \; ❻ \;\mid y \; ❼ \;\mid E \,\&\, E \; ❽ \;\mid E \mid E \; ❾ \end{array}$$

Fig. 1. Example grammar G_{ex}.

- A constraint-based approach for solving SemGuS problems using CHCs (§2 and §5), capable of producing both a synthesized program for realizable problems, and a proof of unrealizability for unrealizable ones.
- Multiple instantiations of the framework—with different kinds of semantics—that express variant SemGuS problems whose solutions can sometimes be obtained more efficiently (§6).
- An implementation of a SemGuS solver using Z3 [De Moura and Bjørner 2008; Komuravelli et al. 2016], called MESSY. We instantiate MESSY to come with a variety of semantics out-of-the-box, allowing users to easily define and solve SemGuS problems. Moreover, MESSY is the first tool capable of both (i) solving synthesis problems, and (ii) proving unrealizability for imperative-language problems involving a search space with an infinite number of programs.

§3 provides background material. §9 discusses related work. §10 concludes. A longer version of this paper, containing proofs for the various theorems, is available on arXiv [Kim et al. 2020].

2 MOTIVATING EXAMPLE

Consider the problem of synthesizing an imperative program that stores the bitwise-xor of two variables x and y in the variable x, using only bitwise-and and bitwise-or operations and no auxiliary variables. We show how one can define this problem in SemGuS and prove it unrealizable.

2.1 Defining a SemGuS Problem

The first contribution of this paper is the SemGuS framework (§4). A SemGuS problem is defined using three components: (i) a search space given by a regular tree grammar G, (ii) a semantics for the grammar G, and (iii) a specification of the desired behavior of the program.

Supplying SemGuS with a grammar. In this example, the grammar G_{ex} in Figure 1 describes a language of single-loop programs that can contain an arbitrary number of assignments to x and y, but involve only bitwise-and and bitwise-or operations. In the figure, the numbers in the black circles are used as unique identifiers for each production. Note that SyGuS cannot describe the language $L(G_{ex})$ due to the presence of assignments and loops.

Supplying SemGuS with a semantics. The next component of a SemGuS problem is a semantics for terms in the language $L(G_{ex})$. There are many possible ways to define the formal semantics of an imperative language. For example, if we let Γ, Γ_1, and Γ_2 denote valuations of the variables x and y, Equation (1) defines a semantics that a user might give for the term "while b do s".

$$\frac{[\![b]\!](\Gamma, \text{true}) \quad [\![s]\!](\Gamma, \Gamma_1) \quad [\![\text{while } b \text{ do } s]\!](\Gamma_1, \Gamma_2)}{[\![\text{while } b \text{ do } s]\!](\Gamma, \Gamma_2)} \; sem^{WTrue} \tag{1}$$

Equation (1) is a common way to define program semantics, but it contains some ambiguities, such as the method of defining the semantic function $[\![\cdot]\!]$. SemGuS takes an extra level of formalization and requires that semantic rules such as sem^{WTrue} are expressed using *logical relations* and *Constrained Horn Clauses* (CHCs), which are implications that are defined over logical relations and a single logical constraint. As an example, we show how sem^{WTrue} can be expressed as a CHC in SemGuS.

In SemGuS, semantics can be specified in a compositional fashion by associating each *production* in the grammar with one or more semantic rules,[2] with the additional constraint that each rule must be expressible as a CHC. To express the semantics of terms, which are derived from each nonterminal in a production, we assume that each nonterminal N has a corresponding logical relation sem_N, which represents the behavior of the semantic function $[\![\cdot]\!]$ in Equation (1). We refer to these relations as *semantic relations*. For example, the expression $\mathsf{sem}_B(\langle\Gamma, b\rangle, v_b)$ corresponds to the premise $[\![b]\!](\Gamma, v_b)$: the semantic relation sem_B tells us that executing the term $b \in L(B)$ on incoming state Γ results in a value v_b.

$$\frac{\mathsf{sem}_B(\langle\Gamma, b\rangle, \mathsf{true}) \quad \mathsf{sem}_S(\langle\Gamma, s\rangle, \Gamma_1) \quad \mathsf{sem}_{Start}(\langle\Gamma_1, \mathsf{while}\ b\ \mathsf{do}\ s\rangle, \Gamma_2)}{\mathsf{sem}_{Start}(\langle\Gamma, \mathsf{while}\ b\ \mathsf{do}\ s\rangle, \Gamma_2)}\ \mathit{sem}^{\mathsf{WTrue}}_{Start \to \mathsf{while}\ B\ \mathsf{do}\ S}\ \text{❶} \qquad (2)$$

Equation (2) uses semantic relations to express the same semantics as Equation (1), and fits our criterion of using CHCs as semantic rules: the relations sem_B, sem_S, and sem_{Start} represent the semantics of terms, while the whole of Equation (2) can be read as a CHC.

SemGuS assumes the supplied semantics are of the form in Equation (2): that is, the semantic relations model the semantics of terms, and each semantic inference rule is a CHC.[3] This assumption is not restrictive, nor does it impose a complex format—CHCs are expressive enough to model a recursively defined big-step semantics. Rather than restricting the kinds of problems SemGuS supports, these restrictions mainly exist to formalize the meaning of the word "semantics".

Supplying SemGuS with a behavioral specification. The behavioral specification of SemGuS states what property the target program should satisfy. One can provide a logical formula that relates the input and output valuations of the program variables, or alternatively, provide a set of examples, which is the kind of specification our algorithm for solving SemGuS problems relies upon.[4]

For our example, suppose that the specification is given as the set of input valuations $[(6, 9), (44, 247), (14, 15)]$ (each representing the values of x and y, respectively), which produce the output values $[15, 219, 1]$ (each representing the final value of x). Call this example set E_{ex}. In §2.2, we show how our algorithm (implemented in MESSY) can synthesize a valid solution on a *subset* of E_{ex}, namely, $[(6, 9)]$ with output $[15]$, where bitwise-xor is equivalent to bitwise-or, i.e., this sub-problem is realizable. We then describe how our algorithm proves that *no* program in the language of $L(G_{ex})$ can compute the bitwise-xor for all the examples in E_{ex}, i.e., that the problem is unrealizable.

2.2 Solving SemGuS Problems

The second contribution of this paper is a procedure for solving SemGuS problems (§5). To solve a SemGuS problem, this paper utilizes two key ideas: (i) both the syntax and the semantics of a synthesis problem can be described using Constrained Horn Clauses, and, (ii) one can phrase the synthesis problem as a proof search over CHCs.

Syntax and Semantic Rules. Describing a grammar using CHCs is a straightforward process: taking the production $Start \longrightarrow \mathsf{while}\ B\ \mathsf{do}\ S$ ❶ as an example, the production states that one can obtain a valid term for the nonterminal $Start$ using valid terms for nonterminals B and S.

[2]The ability to define multiple semantic rules for a production is useful when defining semantics for productions such as $Start \longrightarrow \mathsf{while}\ b\ \mathsf{do}\ s$, which is commonly equipped with two rules that describe looping and loop termination.
[3]For clarity, we sometimes use the format from Equation (1) when introducing semantics for SemGuS to work with (§6).
[4]Given a logical specification, one can always generate a set of examples and add more examples as needed through a technique known as *counterexample-guided inductive synthesis* (CEGIS), which is applied in many synthesizers.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{syn}_E(x)} \quad \dfrac{}{\mathsf{syn}_E(y)}}{\mathsf{syn}_B(x < y)} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{syn}_E(x)} \quad \dfrac{}{\mathsf{syn}_E(y)}}{\mathsf{syn}_E(x \mid y)}}{\mathsf{syn}_S(x := x \mid y)}}{\mathsf{syn}_{Start}(t)} \quad \dfrac{\dfrac{(6 < 9) = \mathsf{true}}{\mathsf{sem}_B(\langle(6,9),\, x < y\rangle,\, \langle \mathsf{true}\rangle)} \quad \dfrac{\dfrac{6 \mid 9 = 15}{\mathsf{sem}_E(\langle(6,9),\, x \mid y\rangle,\, \langle 15\rangle)}}{\mathsf{sem}_S(\langle(6,9),\, x := x \mid y\rangle,\, \langle(15,9)\rangle)} \quad \dfrac{\dfrac{(15 < 9) = \mathsf{false}}{\mathsf{sem}_B(\langle(15,9),\, x < y\rangle,\, \langle \mathsf{false}\rangle)}}{\mathsf{sem}_{Start}(\langle(15,9),\, t\rangle,\, \langle(15,9)\rangle)}}{\mathsf{sem}_{Start}(\langle(6,9),\, t\rangle,\, \langle(15, y')\rangle)} \over \textit{Realizable}} \ \text{Query}$$

Fig. 2. The full proof tree for synthesizing the bitwise-xor of x and y from the input example $[(x,y)] = [(6,9)]$ with output $[15]$, using the grammar G_{ex}. The term $t = (\text{while } x < y \text{ do } x := x \mid y)$ satisfies the one example provided: the loop iterates once to set x to 15.

Equation (3) encodes this property as a CHC.

$$\frac{\mathsf{syn}_B(b) \quad \mathsf{syn}_S(s)}{\mathsf{syn}_{Start}(\text{while } b \text{ do } s)} \ \mathit{syntax}_{Start \to \text{while } B \text{ do } S} \quad ❶ \tag{3}$$

The logical relations syn_B, syn_S, and syn_{Start} in Equation (3) model whether the supplied arguments are valid terms that may be derived from the corresponding nonterminals B, S, and $Start$. We refer to relations such as syn_S as *syntax relations*, and rules such as Equation (3) as *syntax rules*.

§2.1 illustrated how the programming-language semantics can be expressed using CHCs; in tandem with the syntax rules, they represent the semantics of all possible programs in the language.

Specification Query. The final step to solving a SemGuS problem is to create a query that encodes the behavioral specification, asking whether any of the programs generated by the grammar is consistent with the specification on the set of input examples E. This question is posed via the Query rule below, which checks for the existence of a term t that satisfies the syntax rules and the semantic rules, each instantiated with input $e_i \in \mathsf{E}$ and corresponding output value o_i.[5]

$$\frac{\mathsf{syn}_{Start}(t) \quad \bigwedge_{e_i \in \mathsf{E}} \mathsf{sem}_{Start}(\langle e_i, t\rangle, o_i)}{\mathit{Realizable}} \ \text{Query} \tag{4}$$

Generally, one could choose to use symbolic variables for o_i instead of concrete output examples, by adding an additional premise $\bigwedge_{e_i \in \mathsf{E}} \psi(e_i, o_i)$ to ensure that the input-output pair e_i, o_i meets the specification ψ. In this section, we consider concrete output examples for ease of presentation.

Expressing the entire SemGuS problem as a set of inference rules and a query effectively reduces solving the SemGuS problem to a proof search to establish that *Realizable* holds using the given inference rules. If one can prove that the premises of Equation (4) hold, then the SemGuS problem is realizable, and the term t is a concrete answer to the problem. If there exists no proof for *Realizable* using the inference rules, then the SemGuS problem is unrealizable.

Synthesizing Programs. To see how a valid program is synthesized based on our construction, consider the problem of synthesizing a program that computes the bitwise-xor of x and y, specified using the singleton example set $[(x,y)] = [(6,9)]$ and output $[15]$. In this case, the CHC solver is responsible for finding a term t that satisfies the conjunction of the relations (and, as stated above, also corresponds to proving):

$$\mathsf{syn}_{Start}(t) \qquad \mathsf{sem}_{Start}(\langle(6,9), t\rangle, \langle(15, y')\rangle)$$

The sem_{Start} literal states that the final value of x should be 15, and, via free variable y', that we do not care about the final value of y. (The reason why the final value of y is also present in the relation is because nonterminal $Start$ represents a statement, and thus the relation sem_{Start} must track changes to both x and y.) For this input/output pair, bitwise-xor is indistinguishable

[5]In practice, we encode terms into an alternative representation because SMT solvers have difficulties when terms are expressed directly. This encoding is presented in §5.

from bitwise-or (|), making the problem realizable: the term t = (while $x < y$ do $x := x \mid y$) is a solution. Figure 2 shows how such a solution corresponds to a proof in our system of CHCs, where the syntax premise $\mathsf{syn}_{Start}(t)$ ensures that t is indeed a valid term, and the semantic premise $\mathsf{sem}_{Start}(\langle(6, 9), t\rangle, \langle(15, y')\rangle)$ ensures that the semantics of t matches the specification. Our tool MESSY (which is based on Z3 [De Moura and Bjørner 2008] and its CHC solver Spacer [Komuravelli et al. 2016]) succeeds in deriving the proof tree in Figure 2, from which the term t is then extracted.

Proving Unrealizability. To see how a SemGuS problem is proved unrealizable, recall our full example set E_{ex}, for which the solver must find some term t that satisfies the relations:

$$\begin{array}{ll} \mathsf{syn}_{Start}(t) & \mathsf{sem}_{Start}(\langle(6, 9), t\rangle, (15, y'_1)) \\ \mathsf{sem}_{Start}(\langle(44, 247), t\rangle, (219, y'_2)) & \mathsf{sem}_{Start}(\langle(14, 15), t\rangle, (1, y'_3)) \end{array}$$

Put another way, the solver must establish that there exists *no* term t that satisfies all four relations at once—i.e., that *Realizable* is *unsatisfiable*—to prove the problem unrealizable.

Note that our algorithm does not provide additional machinery to reason about loops. Instead, we rely on the CHC solver to discover lemmas about *sets of loops*—as opposed to single loops—to prune the search space. When attempting to find a proof tree for *Realizable* consistent with the examples in E_{ex}, the CHC solver Spacer eventually proves the following invariant for the third example, namely $(14, 15) \to 1$: "For all values of x that are reachable from the nonterminal *Start*, $x \,\&\, 4 = 4$ always holds". The lemma $x \,\&\, 4 = 4$ implies that in the theory of fixed-length bitvectors, the *third bit of x* must *always* be true when the loop terminates. This condition conflicts with the output 1 (in which the third bit is false), which shows that the third example can never be satisfied—which, in turn, implies that the synthesis problem is unrealizable! Note that this lemma is an invariant of the *nonterminal Start*—i.e., an invariant of *all* programs derivable from *Start*—not just some specific program derivable from *Start*.

One might be tempted to give an operational reading of the Query rule as following the paradigm of *generate and test*: $\mathsf{syn}_{Start}(t)$ generates t, which then must pass the tests $\mathsf{sem}_{Start}(\langle I_1, t\rangle, O_1) \ldots \mathsf{sem}_{Start}(\langle I_n, t\rangle, O_n)$. However, the ability of Spacer to prove lemmas of the sort discussed above means that MESSY is not merely enumerating and testing individual programs. On the contrary, the technique for solving SemGuS problems infers lemmas about the behavior of *multiple* programs in the language of the grammar, and uses them to prune the search space!

2.3 Instantiating SemGuS with Other Semantics

The procedure described in the previous section gives a general way to solve SemGuS problems, but also suffers from several limitations. For example, one might have to prove a large number of sem relations from the premise of the Query rule if there are a large number of input-output examples; or, because solving CHCs is still difficult in general, the problem may simply be too difficult to solve. As a third contribution, we show how, thanks to its generality, SemGuS can be supplied with alternative semantics to address some of these challenges (§6). As an example, here we show how to supply SemGuS with an *abstract semantics* to prove unrealizability more efficiently.

Consider again the problem of proving that synthesizing a bitwise-xor program from the grammar G_{ex} is unrealizable. As described in §2.2, the lemma used to prove this fact states that the third bit of x under the example $(14, 15) \to 1$ is always set to true, conflicting with the output 1. While we proved this problem unrealizable using a precise semantics, it is also possible to prove unrealizability using an abstract domain. For example, consider the abstract domain $\mathbb{B}_3$, which only tracks the value of the third bit of every variable, using the values true, false, and $\top$ (top), where $\top$ represents the scenario in which the third bit may be either true or false: i.e., the semantics may be imprecise. Then, one could supply an abstract semantics for a term $e_1 \,\&\, e_2$ (the bitwise-and of e_1 and e_2), created from the production $E \to E \,\&\, E$, as:

$$\frac{[\![e_1]\!]^\#(\Gamma^\#, v_1^\#) \quad [\![e_2]\!]^\#(\Gamma^\#, v_2^\#) \quad v^\# = (\text{if } (v_1^\# = \top \vee v_2^\# = \top) \text{ then } \top \text{ else } v_1^\# \mathbin{\&} v_2^\#)}{[\![e_1 \mathbin{\&} e_2]\!]^\#(\Gamma^\#, v^\#)} \text{And}^\# \tag{5}$$

The final premise in Equation (5) represents the abstract transformer of bitwise-and $\mathbb{B}_3$, which sends the computation to $\top$ if any of $v_1^\#$ or $v_2^\#$, the abstract values for v_1 and v_2, are $\top$, or computes the exact value otherwise. $\top$ can be generated in $\mathbb{B}_3$ by operators such as +, which always loses precision because it does not track carry bit values from the second position.

From a SemGuS point of view, an abstract semantics is merely a different semantics, which allows SemGuS problems with abstract semantics such as $\mathbb{B}_3$ to be solved using the same algorithm described in §2.2. Although $\mathbb{B}_3$ is more lightweight compared to the precise semantics discussed in §2.1, it is sufficient to prove the unrealizability of synthesizing bitwise-xor from G_{ex}—therefore resulting in a more efficient solving procedure.

In §6, we show how other semantics, such as an underapproximating one, can be supplied to the SemGuS framework, each with their advantages. These semantics illustrate one of the benefits of allowing a user to supply their own semantics in SemGuS—in addition to a wider range of definable problems, one can also describe specific strategies to optimize the synthesis problem at hand!

3 PRELIMINARIES

In this section, we provide some background information on concepts that we build upon for the rest of the paper. §3.1 provides background on Horn Clauses, which are used in §5 to define our procedure for solving SemGuS problems. §3.2 is about trees, regular tree grammars, and program semantics, which are required for our definition of the SemGuS problem in §4.

3.1 Constrained Horn Clauses

Constrained Horn Clauses (CHCs) are a class of logical rules that we use to formalize the concept of semantics, as well as use in our algorithm for solving SemGuS problems.

Definition 3.1 (Constrained Horn Clauses.). A *Constrained Horn Clause* is a first-order formula of the form $\forall \vec{x}, \vec{x_1}, \ldots, \vec{x_n}.(\phi \wedge R_1(\vec{x_1}) \wedge \cdots \wedge R_n(\vec{x_n}) \implies H(\vec{x}))$, where ϕ is a constraint over some background theory that may contain variables from $\vec{x}, \vec{x_1}, \ldots, \vec{x_n}$, and $R_1, \ldots, R_n$ and H are uninterpreted relations.

CHCs often allow first-order terms directly in their arguments, which can be viewed as a form of syntactic sugar with respect to Definition 3.1: for a first-order formula $f(x)$, one can rewrite $R(f(x))$ to $R(y)$ and add the constraint $f(x) = y$ to ϕ. For the remainder of the paper, we assume that CHCs allow first-order terms as their arguments.

Example 3.2. Equations (6) and (7) give an example of how the syntax and semantic rules from §2 can be interpreted as CHCs.

$$\forall b, s.\ \mathsf{syn}_B(b) \wedge \mathsf{syn}_S(s) \implies \mathsf{syn}_{Start}(\mathsf{while}\ b\ \mathsf{do}\ s) \tag{6}$$

$$\begin{aligned} \forall \Gamma, \Gamma_1, \Gamma_2, b, s.\ (v_b = true) \wedge \mathsf{sem}_B(\langle \Gamma, b \rangle, v_b) & \\ \wedge\ \mathsf{sem}_S(\langle \Gamma, s \rangle, \Gamma_1) \wedge \mathsf{sem}_{Start}(\langle \Gamma_1, \mathsf{while}\ b\ \mathsf{do}\ s \rangle, \Gamma_2) & \\ \implies \mathsf{sem}_{Start}(\langle \Gamma, \mathsf{while}\ b\ \mathsf{do}\ s \rangle, \Gamma_2) & \end{aligned} \tag{7}$$

Syntax and semantic relations such as syn_B or sem_{Start} are expressed as uninterpreted relations, while atomic semantic operations such as addition are represented using the constraint ϕ.

CHCs occur frequently in program verification, and many efficient algorithms for solving CHCs have been developed [Blanc et al. 2013; Komuravelli et al. 2016; McMillan and Rybalchenko 2013]. In terms of CHC solving, the proof search described in §2 is augmented with an extra CHC *Realizable* $\implies$ false, which asserts $\neg$*Realizable*. If there exists an interpretation of the syntax and semantic relations that can derive *Realizable*, then the system of CHCs is not valid—and, as shown in §2, one can extract a program from the proof tree for *Realizable*. If *Realizable* cannot be derived from any interpretation of the syntax and semantic relations, the system of CHCs is valid, which corresponds to unrealizability. In this paper, we use the (un)satisfiability of *Realizable* itself, instead of the validity of the augmented CHC problem, to illustrate our algorithm.

3.2 Trees, Tree Grammars, and Semantics

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., $\Sigma = \{+^{(2)}, c^{(0)}\}$. (For brevity, the superscript is often 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).

In this paper, 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 $a_{\sigma^{(i)}} = (\tau_0, \tau_1, \ldots, \tau_i)$, 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 3.3 (Regular tree grammar). A *typed regular tree grammar* (RTG) is a tuple $G = (N, \Sigma, S, a, \delta)$, where N is a finite set of non-terminal symbols of arity 0; Σ is a ranked alphabet; $S \in N$ is an initial nonterminal; a is a type assignment that gives types for members of $\Sigma \cup N$; and δ is a finite set of productions of the form $A_0 \longrightarrow \sigma^{(i)}(A_1, ..., A_i)$, where for $1 \leq j \leq i$, each $A_j \in N$ is a nonterminal such that if $a_{\sigma^{(i)}} = (\tau_0, \tau_1, ..., \tau_i)$ then $a_{A_j} = \tau_j$.

Given a tree $t \in T_{\Sigma \cup N}$, applying a production $r = A \longrightarrow \beta$ to t produces the tree t' resulting from replacing the leftmost 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.

Figure 1 from §2 shows an example of a typed regular tree grammar. For readability, the grammar does not contain explicit symbols—e.g., the production $Start \longrightarrow$ while B do S should be more correctly stated as a production $Start \longrightarrow$ while(B, S), where while is a binary symbol. We will use the former notation for readability, and assume that all expressions are well-typed.

We note that terms can be represented using trees of productions, which makes it easier to distinguish terms created by different productions with identical operators: we use this representation in our tool MESSY.

Example 3.4. Recall the grammar G_{ex} from Figure 1, where each production is labeled with a unique identifier ⓝ. The term "while $x < x$ do $x := y$" can be represented using the tree Tree❶(Tree❷(❻, ❻), Tree❹(❼)). The first child tree Tree❷(❻, ❻) represents the condition "$x < x$", while the second child tree Tree❹(❼) represents the assignment "$x := y$".

When defining a SemGuS problem, one has to provide a semantics for the productions in the RTG. The semantic definitions are allowed to use terms from a theory $\mathcal{T}$ (e.g., linear integer arithmetic).

Definition 3.5 (Production-based semantics). Given an RTG $(N, \Sigma, S, a, \delta)$ and a theory $\mathcal{T}$, a *semantics for the grammar* is a function $[\![\cdot]\!]$ that maps every production $A_0 \to \sigma^{(i)}(A_1, ..., A_i)$ of type $a_{\sigma^{(i)}} = (\tau_0, \tau_1, ..., \tau_i)$ to a set of Constrained Horn Clauses of the form $\phi \wedge \mathsf{sem}_{A_1}(\Gamma_1, t_{A_1}, \Upsilon_1) \wedge \cdots \mathsf{sem}_{A_i}(\Gamma_i, t_{A_i}, \Upsilon_i) \implies \mathsf{sem}_{A_0}(\Gamma_0, t_{A_0}, \Upsilon_0)$, where $\mathsf{sem}_{A_0}, \mathsf{sem}_{A_1}, \cdots \mathsf{sem}_{A_i}$ are uninterpreted relations, $\Gamma_0, \Gamma_1, \cdots \Gamma_i$ are variables that represent state, t_{A_k} is a variable that represents a term $t \in L(A_k)$, $\Upsilon_0, \Upsilon_1, \cdots \Upsilon_i$ are variables of type $\tau_0, \tau_1, \cdots \tau_i$, and ϕ is a constraint within the theory $\mathcal{T}$.

The function $[\![\cdot]\!]$ can be lifted to trees as follows: for every subtree t' of t, if $t' = \sigma^{(i)}(t_1, ..., t_i)$, then $[\![t']\!] = [\![\sigma^{(i)}]\!]([\![t_1]\!], \ldots, [\![t_i]\!])$. In practice, the type signatures for Γ and Υ are nearly unrestricted—one has great flexibility in defining for each production (i) the kind of program state that is tracked in Γ, as well as (ii) the kind of value returned in Υ. This approach allows one to use arbitrary CHCs to define the semantics, as long as it contains the term t as an argument.

Example 3.6. The semantics for the statement nonterminal S, from Figure 1 and Equation (2), uses the product type $\mathsf{Int} \times \mathsf{Int}$ for both Γ and Υ.

As is common in many semantic definitions, Def. 3.5 defines the semantics of terms in the grammar inductively. This ability to *equip the grammar with customized semantics* is the defining characteristic that distinguishes SemGuS from SyGuS. In SyGuS, the underlying theory—e.g., LIA—is what corresponds to the specified semantics. In SemGuS, the semantics can be any Constrained Horn Clause defined over the relations $\mathsf{sem}_{A_0}, \mathsf{sem}_{A_1}, \cdots \mathsf{sem}_{A_i}$.

Example 3.7. The big-step semantics of simple imperative languages can be expressed using rules like the one illustrated in Equation (2), which inductively defines the semantic of the production $Start \to$ while B do S through the semantic relations for nonterminals B and S.

4 SEMANTICS-GUIDED SYNTHESIS AND ITS PROPERTIES

We now provide a formal definition of the Semantics-Guided Synthesis problem:

Definition 4.1 (SemGuS). A *SemGuS problem* over a theory $\mathcal{T}$ is a tuple $sem = (G, \forall x.\psi(x, f(x)))$, where G is a regular tree grammar with a production-based semantics $[\![\cdot]\!]$, and $\forall x.\psi(x, [\![f]\!](x))$ is a Boolean formula over the theory $\mathcal{T}$ that specifies the desired behavior of f, where f is a free second-order variable. A **solution** to the SemGuS problem sem is a term $s \in L(G)$ such that $\forall x.\psi(x, [\![s]\!](x))$ holds. We say that sem is **realizable** if a solution exists and **unrealizable** otherwise.

Example 4.2. The problem of synthesizing a program for bitwise-xor described in §2 can be written as a SemGuS problem $sem = (G_{ex}, \forall x, y. f(x, y) = x \oplus y)$ (with $\oplus$ denoting bitwise-xor), where G_{ex} is equipped with a semantics that contains the rule given in Equation (2).

Example 4.2 gives an example of a SemGuS problem where the grammar is equipped with a semantics that one would normally expect for imperative programs. Definition 4.1, which defines SemGuS problems, shows that SemGuS can be instantiated with different kinds of semantics, as long as the semantics satisfies the definition of a production-based semantics (Definition 3.5). This feature allows SemGuS problems to be instantiated with a semantics that is *approximate* with respect to some original semantics. An approximate semantics can be used to efficiently compute one-sided answers to the original problem—either synthesis or unrealizability—depending on the relation between the approximating and the original semantics.

4.1 Unrealizability of SemGuS Problems with Overapproximating Abstract Semantics

In this section, we see how an *overapproximating semantics* can be used to prove *unrealizability*. An overapproximating semantics overapproximates the set of reachable states with respect to an original program semantics; in essence, they are an *abstract semantics* [Cousot and Cousot 1992], and we use the latter term for the rest of the paper. More specifically, we show that if a SemGuS problem $sem = (G, \psi(x, f(x)))$ is unrealizable when G is equipped with an abstract semantics, then sem is unrealizable when equipped with the original semantics as well.

Definition 4.3. For a grammar G equipped with a semantics $[\![\cdot]\!]$, we say $[\![\cdot]\!]^\#$ is an *abstract semantics* for G with respect to $[\![\cdot]\!]$ if there exists an abstraction function α and a concretization function γ, such that for all $t \in L(G)$, if $[\![t]\!](\Gamma, v)$ holds, then $[\![t]\!]^\#(\alpha(\Gamma), \alpha(v))$ holds, and $\Gamma \in \gamma(\alpha(\Gamma)), v \in \gamma(\alpha(v))$, i.e., α and γ form a Galois connection.

In SemGuS, an abstract semantics $[\![\cdot]\!]^\#$ overapproximates the set of values that are obtainable by synthesizing a term from the grammar, again with respect to the original semantics $[\![\cdot]\!]$. Because the set of values is overapproximated, a term synthesized using the abstract semantics may not satisfy the specification when executed with the standard semantics. However, by showing the desired output is *absent* from the set of obtainable values, one can prove *unrealizability* in a sound manner!

Theorem 4.4 (Soundness of Abstract Semantics for Unrealizability). *For a SemGuS problem $sem = (G, \forall x.\psi(x, f(x)))$, if sem is unrealizable when G is equipped with an abstract semantics $[\![\cdot]\!]^\#$, then sem is also unrealizable when G is equipped with $[\![\cdot]\!]$.*

Equipping a SemGuS problem with an abstract semantics still results in a SemGuS problem, which can be solved using the procedure described in §5. Much like how abstract semantics are used for efficient program verification, an abstract semantics can sometimes be used to prove the unrealizability of a SemGuS problem with the original semantics in a much more efficient manner.

4.2 Solving Realizable SemGuS Problems with Underapproximating Semantics

In this section, we show that an *underapproximating semantics*, can be used to synthesize solutions to *realizable* SemGuS problems.

Definition 4.5. For a grammar G equipped with a semantics $[\![\cdot]\!]$, we say $[\![\cdot]\!]^\flat$ *underapproximates* $[\![\cdot]\!]$ on G, or that $[\![\cdot]\!]^\flat$ is an *underapproximating semantics* for G with respect to $[\![\cdot]\!]$, if for every term $t \in L(G)$, every state Γ, and every value v on which $[\![\cdot]\!]^\flat$ *is defined*, $[\![t]\!]^\flat(\Gamma, v) = [\![t]\!](\Gamma, v)$.

Intuitively, an underapproximating semantics is defined as a subset of the original semantics. Outside of the subset upon which it is defined, an underapproximating semantics is undefined, which does not mean that a term can evaluate to any value, but rather that a term *cannot* evaluate to any value. More precisely, one cannot prove any theorems about the relation $[\![t]\!]^\flat(\Gamma, v)$ if $[\![\cdot]\!]^\flat$ is undefined on t, Γ, and v. Instead, an underapproximate semantics is precise on the subset upon which it is defined, i.e., $[\![t]\!]^\flat(\Gamma, v) = [\![t]\!](\Gamma, v)$ if $[\![\cdot]\!]^\flat$ is defined on t, Γ, and v.
In SemGuS, an underapproximating semantics corresponds to a problem where synthesized terms only have meaning if their semantics is defined on the input-output examples. For the subset of terms for which the semantics is defined, the semantics is exact, which allows underapproximating semantics to be used for program synthesis. Because there may be an answer to the problem outside the defined subset, an underapproximating semantics cannot be used for unrealizability.

Theorem 4.6 (Soundness of Underapproximating Semantics for Synthesis). *For a SemGuS problem $sem = (G, \forall x.\psi(x, f(x)))$, if sem is realizable with solution t when G is equipped with an underapproximating semantics $[\![\cdot]\!]^\flat$, then t is also a solution for sem when G is equipped with $[\![\cdot]\!]$.*

An underapproximate semantics indirectly restricts the search space for program synthesis. This restriction is not necessarily related to the grammar supplied to a SemGuS problem, but may have a semantic meaning—for example, a bound on the number of possible loop iterations.

As is the case with an abstract semantics, SemGuS can be supplied with an underapproximate semantics to yield a relatively more efficient procedure for program synthesis, as illustrated in §6.3.

5 SOLVING SEMANTICS-GUIDED SYNTHESIS PROBLEMS VIA CONSTRAINED HORN CLAUSES

This section presents a general procedure for encoding SemGuS problems so that they can be solved by answering a query over Constrained Horn Clauses, which in turn can be solved by an off-the-shelf CHC solver.

§5.1 describes how general SemGuS problems can be solved by solving SemGuS-with-examples problems in tandem with counterexample-guided inductive synthesis; it also states the correctness of our solving procedure. §5.2 presents a method for using *flattened representations* of terms as opposed to trees, to avoid the use of algebraic datatypes in SMT solvers.

5.1 Solving SemGuS Problems with Counterexample-Guided Inductive Synthesis

Counterexample-guided inductive synthesis (CEGIS) is a widely implemented algorithm in program synthesizers. The core idea of CEGIS is that instead of searching for a term that satisfies the specification for the entire input space, the synthesizer searches for a solution that satisfies the specification on a finite set of examples E. A verifier then attempts to prove that the solution is also correct on the universally quantified specification; if not, a counterexample is added to the set of examples. The algorithm then repeats. The main advantage of CEGIS is that it eliminates the universal quantifier over the space of program inputs, yielding a simpler problem.

The algorithm sketched in §2, as well as the one presented in §5.2, is designed to solve *SemGuS-with-examples* problems, which are SemGuS problems where the specification is given in terms of a set of examples E, and has the form $\bigwedge_{x \in \mathsf{E}} \psi(x, [\![f]\!](x))$. To solve general SemGuS problems, the SemGuS-with-examples algorithm can then be embedded within a CEGIS loop, where the specification is given in terms of the set of counterexamples accumulated by CEGIS.

The general idea of using CHCs to describe the syntax and semantics of a SemGuS-with-examples problem $sem = (G, \forall x \in \mathsf{E}.\psi(x, f(x)))$ has already been described in §2: Equation (3) and Equation (2) show how the syntax and the semantics of a production $Start \to \mathsf{while}\ B\ \mathsf{do}\ S$ can be written as CHCs, and it is straightforward to describe other productions in this manner as well.

The final query that describes the specification can be formally written as the following rule.

$$\frac{\mathsf{syn}_{Start}(t) \quad \bigwedge_{\mathsf{e}_i \in \mathsf{E}} \mathsf{sem}_{Start}(\langle \mathsf{e}_i, t\rangle, o_i) \quad \bigwedge_{\mathsf{e}_i \in \mathsf{E}} \psi(\mathsf{e}_i, o_i)}{Realizable}\ \mathsf{Query} \tag{8}$$

Realizable is the final theorem that shows whether the given SemGuS-with-examples problem is realizable or not. If the CHC solver finds a proof for *Realizable*, then the problem is *realizable* and the program t is a solution. If the solver can establish that *Realizable* is *unsatisfiable*, then the problem is *unrealizable*. The correctness of our algorithm can be stated as the following theorem:

Theorem 5.1 (Soundness and Completeness). *Consider a SemGuS-with-examples problem* $sem = (G, \forall x \in \mathsf{E}.\psi(x, f(x)))$, *equipped with semantic rules* $\mathcal{R}_{sem}$, *a specification set* E, *and the* Query *rule (Equation (8)). Let the CHC form of G be* $\mathcal{R}_{syn}$. *Then, Realizable is a theorem over* $\mathcal{R}_{sem}$ *and* $\mathcal{R}_{syn}$ *if and only if the SemGuS-with-examples problem sem is realizable. Moreover, if Realizable is a theorem, then the value of t in the* Query *rule satisfies* $t \in L(G)$ *and* $\forall x \in \mathsf{E}.\ \psi(x, [\![t]\!](x))$.

Theorem 5.1 can be proved by proving the correctness of the syntax rules via structural induction.

As shown in prior work [Solar-Lezama 2013], the CEGIS algorithm is often powerful enough for program synthesis, where a term synthesized for the given examples generalizes to the entire space of possible inputs. Prior work on unrealizability [Hu et al. 2019, 2020] also shows that CEGIS is often powerful enough to prove that a synthesis problem is unrealizable—i.e., the problem does not admit a solution even when only a finite number of examples are considered.

Example 5.2. The problem of synthesizing a program for bitwise-xor described in §2 can be written as a SemGuS-with-examples problem $sem = (G_{ex}, \forall_{x,y \in \mathsf{E}_{ex}} f(x, y) = x \oplus y)$, where $\mathsf{E}_{ex} = [(6, 9), (44, 247), (14, 15)]$. As seen in §2, E_{ex} is sufficient to prove that sem is unrealizable.

In particular, for a SemGuS problem sem, the CEGIS algorithm is sound but incomplete for unrealizability [Hu et al. 2019]. As discussed in §8, CEGIS is still able to synthesize solutions to, or prove unrealizability of, many SemGuS problems. However, this procedure is incomplete.

Theorem 5.3 (CEGIS for unrealizability [Hu et al. 2019]). *Let* sem_{E} *be a SemGuS-with-examples problem identical to sem, but where the specification is given over the input examples* E. *If* sem_{E} *is unrealizable, then sem is unrealizable as well. However, there exists an unrealizable SemGuS problem sem for which* sem_{E} *is realizable for any finite set of examples* E.

5.2 Using Flattened Representations of Terms to Solve SemGuS Problems

While it is possible to solve SemGuS-with-examples problems using terms encoded as trees using the scheme given in §3.2, current solvers sometimes fail to return an answer depending on how well they can handle trees encoded as algebraic datatypes. In this section, we show how to alleviate this problem by using a *flattened representation* of terms, which we refer to as a listing.[6] The idea is that a term t can be encoded using a *pre-order listing* $\mathcal{L}_t$ of the productions applied to derive t.

Example 5.4. Consider once more the term t = while $(x < x)$ do $x := x$ from Example 3.4, constructed from the grammar G_{ex} in Figure 1. The pre-order listing of productions applied to derive t is [❶, ❷, ❻, ❻, ❹, ❻], where $Start \longrightarrow$ while B do S ❶ is the first production applied to the nonterminal $Start$, the next production $B \longrightarrow E < E$ ❷ is applied next, and the remaining productions are applied in left-to-right order as well.

Following the list representation of terms, the next step is to modify the syntax relations and rules to operate over lists. Equation (9) describes the syntax rule generated using a flattened representation of terms for the production $A_0 \longrightarrow \sigma(A_1, \cdots, A_i)$ ⓝ .

$$\frac{\mathsf{syn}_{A_i}(\mathcal{L}_{in}, \mathcal{L}_i) \quad \mathsf{syn}_{A_{i-1}}(\mathcal{L}_i, \mathcal{L}_{i-1}) \cdots \mathsf{syn}_{A_1}(\mathcal{L}_2, \mathcal{L}_1)}{\mathsf{syn}_{A_0}(\mathcal{L}_{in}, \text{ⓝ} :: \mathcal{L}_1)}\ syntax^{\mathsf{List}}_{A_0 \rightarrow \sigma(A_1, \cdots, A_i)\ \text{ⓝ}} \tag{9}$$

There are several things to notice about Equation (9). First, the syntax relation syn_N now ranges over two listings (term representations) as opposed to a single term, where the first listing may be interpreted as an incoming listing and the second an outgoing listing. Here, the relations should evaluate to true if and only if the outgoing listing is equivalent to the pre-order representation of the term concatenated to the incoming listing.

Second, the outgoing listing of a nonterminal is passed as the incoming listing of the next nonterminal in *right-to-left order*, followed by prepending the number of the production to the head of the listing. This algorithm effectively creates a pre-order representation of a term by performing a post-order traversal, appending each production encountered to the head of the listing.

Example 5.5. Consider Equation (3) from §2, which describes the syntax rule for the production $Start \longrightarrow$ while B do S ❶ . Using a list representation of terms, the rule would be modified to:

$$\frac{\mathsf{syn}_S(\mathcal{L}_{in}, \mathcal{L}_2) \quad \mathsf{syn}_B(\mathcal{L}_2, \mathcal{L}_1)}{\mathsf{syn}_{Start}(\mathcal{L}_{in}, \text{❶} :: \mathcal{L}_1)}\ syntax^{\mathsf{List}}_{Start \rightarrow \text{while } B \text{ do } S\ \text{❶}} \tag{10}$$

[6]Listings may be implemented as lists or arrays in an SMT solver.

Equation (10) traverses the nonterminals B, S in right-to-left order, then prepends the identifier $\textcircled{1}$ to the head of the list $\mathcal{L}_1$.

Having encoded a pre-order representation of a term, the semantic rules must interpret this representation accordingly as well. The semantic relations now also range over 4 elements: an incoming listing $\mathcal{L}_{in}$ and an incoming state Γ, followed by an outgoing listing $\mathcal{L}_{out}$ and a resulting value v. They should evaluate to true if and only if for the list $\mathcal{L}_t$ such that $\mathcal{L}_{in} = \mathcal{L}_t + +\mathcal{L}_{out}$, $[\![t]\!](\Gamma, v)$ also evaluates to true for the corresponding term t.

Keeping that in mind, a semantic rule that uses a flattened representation of terms for the production $A_0 \rightarrow \sigma(A_1, \cdots, A_i)$ $\textcircled{n}$, equipped with the semantics $\phi \wedge \mathtt{sem}_{A_1}(\langle \Gamma_1, t_1 \rangle, v_1), \cdots, \mathtt{sem}_{A_i}(\langle \Gamma_i, t_i \rangle, v_i) \implies \mathtt{sem}_{A_0}(\langle \Gamma, t \rangle, v_0)$ is described in Equation (11).

$$\frac{\phi \quad \mathtt{sem}_{A_1}(\langle \Gamma_1, \mathcal{L}_1 \rangle, \langle v_1, \mathcal{L}_2 \rangle) \cdots \mathtt{sem}_{A_i}(\langle \Gamma_i, \mathcal{L}_i \rangle, \langle v_i, \mathcal{L}_{out} \rangle)}{\mathtt{sem}_{A_0}(\langle \Gamma, \textcircled{n} :: \mathcal{L}_1 \rangle, \langle v_0, \mathcal{L}_{out} \rangle)} \; sem^{\mathsf{List}}_{A_0 \rightarrow \sigma(A_1, \cdots, A_i)} \; \textcircled{n} \tag{11}$$

Because the syntax rules have encoded terms as a pre-order listing, the semantic rules are free to interpret the current production by checking the head of the list, then compute values for subterms in *left-to-right* order. The actual semantics of the production remains encoded in ϕ.

Example 5.6. Consider Equation (2) from §2, which describes the semantic rule for the production $Start \rightarrow \mathsf{while}\ B\ \mathsf{do}\ S$ $\textcircled{1}$. Using a list representation of terms, the rule would be modified to:

$$\frac{\mathtt{sem}_B(\langle \Gamma, \mathcal{L}_1 \rangle, \langle \mathsf{true}, \mathcal{L}_2 \rangle) \;\; \mathtt{sem}_S(\langle \Gamma, \mathcal{L}_2 \rangle, \langle \Gamma_1, \mathcal{L}_{out} \rangle) \;\; \mathtt{sem}_{Start}(\langle \Gamma_1, \textcircled{1} :: \mathcal{L}_1 \rangle, \langle \Gamma_2, \mathcal{L}_{out} \rangle)}{\mathtt{sem}_{Start}(\langle \Gamma, \textcircled{1} :: \mathcal{L}_1 \rangle, \langle \Gamma_2, \mathcal{L}_{out} \rangle)} \tag{12}$$

The list $\textcircled{1} :: \mathcal{L}_{in}$ represents the entire term for while B do S in preorder—the tailing list $\mathcal{L}_{out}$ represents the part that comes after while B do S.

Theorem 5.7 (Correctness of Listings). *Let $\mathcal{R}^{\mathsf{List}}_{sem}$ be a set of semantic rules using a flattened representation of terms, created from the set of semantic rules $\mathcal{R}_{sem}$. For any nonterminal N, $\mathtt{sem}_N(\langle \Gamma, \mathcal{L}_{in} \rangle, \langle v, \mathcal{L}_{out} \rangle)$ is a theorem of $\mathcal{R}^{\mathsf{List}}_{sem}$ iff $\mathtt{sem}_N(\langle \Gamma, t \rangle, v)$ is a theorem of $\mathcal{R}_{sem}$, and $\mathcal{L}_{in} = \mathcal{L}_t \mathcal{L}_{out}$ (i.e., $\mathtt{concat}(\mathcal{L}_t, \mathcal{L}_{out})$) where $\mathcal{L}_t$ is the pre-order listing of a term $t \in L(N)$.*

Theorem 5.7 states the correctness of the flattened term representations, and can be proved using induction on the height of the derivation tree. The specification query is similar to the one given in Equation (8), except that the new syntactic and semantic relations are used in place of the old ones.

6 INSTANTIATING SEMGUS WITH VARIOUS SEMANTICS

We now proceed to showcase the capabilities of the SemGuS framework by instantiating it with a variety of semantics to solve imperative program-synthesis problems. In §6.1, §6.2, and §6.3, we are concerned with various different semantics for the imperative programming language G_{impv}, from Figure 3. Values in G_{impv} range over integers, bitvectors, Boolean values, and arrays. G_{impv} contains most common imperative structures, such as assignments, branches and loops. Imperative grammars that use the same operators but different productions can be viewed as being derived from G_{impv}, which means that the techniques introduced in this section are applicable to *any* imperative grammar, as long as they use a subset of the operators in G_{impv}.

In §6.1, we discuss how to instantiate an imperative SemGuS problem with an alternative exact semantics. This semantics, called a *vectorized semantics*, sidesteps the problem of having to consider multiple examples separately. In §6.2, we show how SemGuS can be instantiated with an *abstract semantics* to prove the unrealizablity of a synthesis problem, and in §6.3, how an *underapproximating semantics* can be used to more efficiently compute solutions for a realizable problem.

Stmt	S	$::=$	$x := E \mid x := C \mid arr[E] := E \mid S; S \mid$ if B then S else $S \mid$ while B do S
BVExpr	C	$::=$	$x \mid \bar{0} \mid \bar{1} \mid C \mathbin{\&} C \mid (C \mid C) \mid\ !C \mid C + C \mid$ if B then C else C
IntExpr	E	$::=$	$x \mid 0 \mid 1 \mid x \mid E + E \mid$ if B then E else $E \mid arr[E]$
BoolExpr	B	$::=$	true $\mid$ false $\mid \neg B \mid B \wedge B \mid E < E \mid C < C$

Fig. 3. The general imperative grammar G_{Impv} that we are interested in.

Finally, in §6.4, we sketch how SemGuS can be instantiated with a semantics for regular expressions to create a tool to synthesize regexes from positive and negative examples.

6.1 Instantiating SemGuS with an Alternative Exact Semantics

A straightforward way of instantiating a SemGuS problem is to supply SemGuS with a standard semantics, as discussed in §2 and §5. For example, the three rules in Figure 4a are standard semantic rules that define the semantics of the terms "$x := e$" and "while b do s". These semantics operate over a single state, and compute exact values for all terms in the program.

However, this straightforward approach induces a substantial drawback in the Query rule in Equation (8). In each premise of the Query rule, the solver must re-derive proof trees for each example, even though they are all structurally similar due to sharing the same term representation.

To mitigate this inefficiency, we develop a different exact semantics, called the *vectorized semantics*, and show that SemGuS can be instantiated with this semantics as well. The vectorized semantics modifies the semantics of standard imperative programs to accommodate and execute multiple examples simultaneously in the form of vectors. This idea allows us to merge the examples, as well as the semantic premises $\mathsf{sem}_N(\langle \mathsf{T}, \mathsf{e}_1\rangle, o_1), \cdots, \mathsf{sem}_N(\langle \mathsf{T}, \mathsf{e}_n\rangle, o_n)$ of the Query rule, into a single semantic premise $\mathsf{sem}_N(\langle \mathsf{T}, \vec{\mathsf{e}}\rangle, \vec{o})$, where $\vec{\mathsf{e}}$ and $\vec{o}$ represent the vectorized input-output examples.

The main challenge in defining a vectorized semantics is that, in the presence of loops and conditionals, different examples can cause a given loop to run a different number of times, and can take different branches of an if-statement. Here, we note that SemGuS is not the cause of these challenges, nor does it require the vectorized semantics; rather, SemGuS is what provides us with the possibility of defining different semantics that are better suited to solving the task at hand.

The three rules in Figure 4b present the big-step semantics for the terms $x := e$ and while b do s, the terms that are most relevant to overcoming these challenges. The most interesting rule here is $\mathsf{WTrue_E}$. This rule states that as long as one of the examples in the vector makes the guard b true, the body of the loop should be entered. However, only the variable valuations that make the guard true are updated in the loop-body s (the $\textsc{proj}(\vec{\Gamma}, \vec{v}_b)$ operator sets all valuations for which the guard is false to the special value $\bot$). The whole process is repeated (using the projected vector of valuations) until all entries of $\vec{v}_b$ are $\bot$, as stated in $\mathsf{WFalse_E}$. Finally, the vector of valuations in the bottom of the rule contains the $\textsc{merge}$ of valuations for which the guard was false, and valuations $\vec{\Gamma}_2$ that resulted from running the loop on the valuations $\textsc{proj}(\vec{\Gamma}, \vec{v}_b)$ for which the guard was true.

When supplying vectorized semantics to a SemGuS-with-examples problem, one should supply a *single* vectorized example that contains all the examples from the original example set. Aside from this difference in how examples should be supplied, the vectorized semantics can be treated just like any other semantics, meaning that the CHC-based solving procedure from §2 and §5 still holds. Moreover, as stated at the start of this section, the vectorized semantics illustrated above can be generated automatically for all subgrammars of G_{impv}, which allows it to be used as a general optimization for solving imperative SemGuS problems (as our tool MESSY does).

$$\frac{[\![e]\!](\Gamma, v) \quad \Gamma_r = \Gamma[x \mapsto v]}{[\![x := e]\!](\Gamma, \Gamma_r)}\ \text{Assign} \qquad \frac{[\![b]\!](\Gamma, v_b) \quad v_b = \text{false}}{[\![\text{while } b \text{ do } s]\!](\Gamma, \Gamma)}\ \text{WFalse}$$

$$\frac{[\![b]\!](\Gamma, v_b) \quad v_b = \text{true} \quad [\![s]\!](\Gamma, \Gamma_1) \quad [\![\text{while } b \text{ do } s]\!](\Gamma_1, \Gamma_2)}{[\![\text{while } b \text{ do } s]\!](\Gamma, \Gamma_2)}\ \text{WTrue}$$

(a) Standard semantic rules for the terms $x := e$ and while b do s in $L(G_{Impv})$, where the semantic function is denoted by $[\![\cdot]\!]$.

$$\frac{[\![e]\!]_{\mathrm{E}}(\vec{\Gamma}, \vec{v}) \quad \forall i.\ \vec{\Gamma}_r[i] = \vec{\Gamma}[i][x \mapsto \vec{v}[i]]}{[\![x := e]\!]_{\mathrm{E}}(\vec{\Gamma}, \vec{\Gamma}_r)}\ \text{Assign}_{\mathrm{E}} \qquad \frac{[\![b]\!]_{\mathrm{E}}(\vec{\Gamma}, v_b) \quad \forall i.\ \vec{v}_b[i] = \text{false}}{[\![\text{while } b \text{ do } s]\!]_{\mathrm{E}}(\vec{\Gamma}, \vec{\Gamma})}\ \text{WFalse}_{\mathrm{E}}$$

$$\frac{[\![b]\!]_{\mathrm{E}}(\vec{\Gamma}, \vec{v}_b) \quad \exists i.\ \vec{v}_b[i] = \text{true} \quad [\![s]\!]_{\mathrm{E}}(\textsc{proj}(\vec{\Gamma}, \vec{v}_b), \vec{\Gamma}_1) \quad [\![\text{while } b \text{ do } s]\!]_{\mathrm{E}}(\vec{\Gamma}_1, \vec{\Gamma}_2)}{[\![\text{while } b \text{ do } s]\!]_{\mathrm{E}}(\vec{\Gamma}, \textsc{merge}(\textsc{proj}(\vec{\Gamma}, \neg\vec{v}_b), \textsc{proj}(\vec{\Gamma}_2, \vec{v}_b)))}\ \text{WTrue}_{\mathrm{E}}$$

$$\textsc{proj}(\vec{\Gamma}, \vec{v}_b) \triangleq [\text{if } \vec{v}_b[0] \text{ then } \vec{\Gamma}[0] \text{ else } \bot, \cdots, \text{if } \vec{v}_b[n-1] \text{ then } \vec{\Gamma}[n-1] \text{ else } \bot]$$

$$\textsc{merge}(\textsc{proj}(\vec{\Gamma}, \neg\vec{v}_b), \textsc{proj}(\vec{\Gamma}', \vec{v}_b)) \triangleq$$
$$[\text{if } \vec{v}_b[0] \text{ then } \vec{\Gamma}[0] \text{ else } \vec{\Gamma}'[0], \cdots, \text{if } \vec{v}_b[n-1] \text{ then } \vec{\Gamma}[n-1] \text{ else } \vec{\Gamma}'[n-1]]$$

(b) Sample vectorized semantic rules for the terms $x := e$ and while b do s in $L(G_{Impv})$, where the (vectorized) semantic function is denoted by $[\![\cdot]\!]_{\mathrm{E}}$. $\bot$ is a special state that ignores all computation performed.

Fig. 4. Standard and vectorized semantics for the terms $x := e$ and while b do s.

6.2 Using Abstract Semantics in SemGuS to Prove Unrealizability

In this section, we show how the grammar G_{impv} can be instantiated with an abstract semantics to prove the unrealizability of SemGuS problems, following the idea introduced in §4.1.

There are many abstract semantics with which one can equip a language. Here, we use the abstract domain $\mathbb{B}_i$ presented in §2.3 as an example, which tracks only the i-th bit of a variable using three values: true, false and $\top$ (the join of true and false).

Example 6.1. Recall Equation (5), which represents the abstract semantics for a term e_1 & e_2 from G_{ex} of §2, using the abstract domain $\mathbb{B}_3$. The right-hand side of the final premise describes the abstract semantic function $[\![\&]\!]^{\#}$ for the operator &, which sends the computation to $\top$ if any of $v_1^{\#}$ or $v_2^{\#}$ are $\top$, and computes the exact value otherwise. Note how the semantic relations, as well as the structure of the semantic rule, remain unchanged—from the viewpoint of SemGuS, an abstract semantics expressed using CHCs is merely a different semantics supplied to SemGuS, for which one can apply the same solving procedure as given in §2 and §5.

Different abstract domains have different degrees of efficiency and precision in SemGuS. To see why, consider how one would deal with branches using the abstract domain described above. This particular abstract domain cannot handle comparisons well because it only tracks a single bit, and thus it is almost always the case that one does not know which branch to take in an if-statement. There are two possible approaches in this situation—one may just choose to assign $\top$ to the result of the branch, or one may try and execute both branches and assign their join to the result. This problem arises for both if-then-else statements and loops. As an example, two different rules for loop iteration are described in Example 6.2.

Example 6.2. Equations (13) and (14) present different abstract semantics for the term while b do s from G_{ex} of §2, using the abstract domain $\mathbb{B}_3$, which tracks only the third bit of each variable.

$$\frac{[\![b]\!]^\#(\Gamma^\#, v_b^\#) \quad [\![s]\!]^\#(\Gamma^\#, \Gamma_1^\#) \quad [\![\text{while } b \text{ do } s]\!]^\#(\Gamma_1^\#, \Gamma_2^\#) \quad \Gamma_r^\# = \top}{[\![\text{while } b \text{ do } s]\!]^\#(\Gamma^\#, \Gamma_r^\#)} \;\text{WTrue}^\#_{\text{Havoc}} \tag{13}$$

$$\frac{[\![b]\!]^\#(\Gamma^\#, v_b^\#) \quad [\![s]\!]^\#(\Gamma^\#, \Gamma_1^\#) \quad [\![\text{while } b \text{ do } s]\!]^\#(\Gamma_1^\#, \Gamma_2^\#) \quad \Gamma_r^\# = \textsc{join}(\Gamma^\#, \Gamma_2^\#)}{[\![\text{while } b \text{ do } s]\!]^\#(\Gamma^\#, \Gamma_r^\#)} \;\text{WTrue}^\#_{\text{Join}} \tag{14}$$

In both scenarios, the value of v_b will be $\top$ because knowing only the third bit does not give us enough information to resolve a condition of the term "$e < e$" from G_{ex}. In this situation, the rule $\text{WTrue}_{\text{Havoc}}$ simply gives up and assigns $\top$ to the resulting value Γ_r. On the other hand,the rule $\text{WTrue}_{\text{Join}}$ attempts to preserve some precision by assigning the join of when the condition evaluates to true (Γ_2, as the loop iterates in this case) and when the condition evaluates to false (Γ, as the loop body does not execute). If both Γ and Γ_2 contain $x^\# = \text{true}$, then $\text{WTrue}_{\text{join}}$ is capable of inferring that the result of while b do s also has $x^\# = \text{true}$, while $\text{WTrue}_{\text{Havoc}}$ cannot.

The semantics expressed by $\text{WTrue}_{\text{Join}}$ is more precise and more expensive than the first option. For the example in §2, an abstract semantics using $\text{WTrue}_{\text{Havoc}}$ will fail to prove unrealizability of synthesizing bitwise-xor, because it cannot resolve the branch of the loop. On the other hand, the added precision from $\text{WTrue}_{\text{Join}}$ succeeds in proving unrealizability, showing how different abstract domains can solve different SemGuS problems.

In general, there are many different abstract domains that one could use for a SemGuS problem, as well as automated methods [Wang et al. 2018a, 2017] to discover them. An interesting line of future work would be to design an algorithm for extracting abstract domains from SemGuS proofs (like we did in our selection of the domain $\mathbb{B}_i$) to solve other SemGuS problems more efficiently.

6.3 Using Underapproximating Semantics in SemGuS for Program Synthesis

In this section, we demonstrate how SemGuS can be equipped with an *underapproximating semantics* to perform program synthesis, following the idea from §4.2. Example 6.3 shows an underapproximating semantics that sets a bound on the number of times each loop may be executed, as in bounded model checking [Clarke et al. 2003]. The change to the semantics is simple—one simply adds a bound to the state and decreases the bound by one each time a loop iteration is performed.

Example 6.3. Equations (15) and (16) present an underapproximating semantics for the term while b do s, where the number of loop iterations is bounded by a fresh variable i.

$$\frac{[\![b]\!]^\flat(\langle\Gamma, i\rangle, \text{true}) \quad i > 0 \quad [\![s]\!]^\flat(\langle\Gamma, i\rangle, \langle\Gamma', i\rangle) \quad [\![\text{while } b \text{ do } s]\!]^\flat(\langle\Gamma', i{-}1\rangle, \langle\Gamma_r, i{-}1\rangle)}{[\![\text{while } b \text{ do } s]\!]^\flat(\langle\Gamma, i\rangle, \langle\Gamma_r, i\rangle)} \;\text{WTrue}^\flat \tag{15}$$

$$\frac{[\![b]\!]^\flat(\langle\Gamma, i\rangle, \text{false}) \quad i > 0}{[\![\text{while } b \text{ do } s]\!]^\flat(\langle\Gamma, i\rangle, \langle\Gamma, i\rangle)} \;\text{WFalse}^\flat \tag{16}$$

One can see how these rules are underapproximating by considering why one is unable to build a proof tree for a loop that must execute more iterations than the unrolling bound. For example, let the unrolling bound be $i = 1$. To prove that $[\![\text{while } b \text{ do } s]\!]^\flat(\langle\Gamma, 1\rangle, \langle\Gamma_r, 1\rangle)$, i.e., the conclusion with $i = 1$, one would also require a proof for the final premise in the rule, namely $[\![\text{while } b \text{ do } s]\!]^\flat(\langle\Gamma', 0\rangle, \langle\Gamma_r, 0\rangle)$. However, a proof of $[\![\text{while } b \text{ do } s]\!]^\flat(\langle\Gamma', 0\rangle, \langle\Gamma_r, 0\rangle)$ requires that $0 > 0$ due to the third premise $i > 0$, which is unsatisfiable. Thus, nothing can be proved about $[\![\text{while } b \text{ do } s]\!]^\flat(\langle\Gamma', 0\rangle, \langle\Gamma_r, 0\rangle)$—corresponding to the fact that $[\![\text{while } b \text{ do } s]\!]^\flat(\langle\Gamma', 0\rangle, \langle\Gamma_r, 0\rangle)$, and any relations that rely on this premise, are *undefined.*

In contrast, the semantics described by Equation (15) match exactly the standard semantics of a while loop for a loop that executes fewer iterations than the unrolling bound.

The constraints that make a semantics underapproximating—for example, $i > 0$ in Example 6.3—can be encoded in the constraint element ϕ of a CHC.

6.4 Instantiating SemGuS to Synthesize Regular Expressions

In this section, we move away from imperative programs and show how one can use the SemGuS framework to solve problems from a different domain—namely, regular expressions. For convenience, in this section we revert to expressing semantics as CHCs (as done in §2). We assume a standard grammar for regular expressions.

$$\mathit{Regex} \ni R ::= c \quad | \quad \epsilon \quad | \quad \phi \quad | \quad (R \mid R) \quad | \quad R \cdot R \quad | \quad R^*$$

Encoding semantics for regular expressions via CHCs poses an interesting challenge, in that the semantics often involves nondeterminism, either when dealing with Kleene star, or finding matching substrings for concatenation. Generally, nondeterminism is naturally dealt with when using only positive examples, where, for example, the *existence* of a proof of the relation $\mathsf{sem}_R(\langle \omega, r\rangle, \mathsf{true})$, where ω is the string to match and r a regex, ensures that there *exists* a run of r that accepts ω. However, nondeterminism mixes poorly with negative examples, which specify strings that a regex should reject: the existence of a proof of the relation $\mathsf{sem}_R(\langle \omega, r\rangle, \mathsf{false})$ merely states that there exists a run of r that rejects ω, whereas the semantics of regular expressions dictate that *no* run of r should accept ω for ω to be rejected.

Fortunately, the expressiveness of CHCs and SemGuS allows us to develop an alternative, deterministic semantics for regexes. Given an input string ω of length n, the semantics is expressed in terms of $(n+1)$-by-$(n+1)$ upper-triangular matrices of Boolean variables: in matrix X, an element $X[i, j]$ indicates whether the considered regex matches the substring $\omega[i, j]$ (as presented in [Pan et al. 2019]). Then, given regexes r_1 and r_2 that return the matrices X_1 and X_2, respectively, the semantics of concatenation is Boolean matrix multiplication—i.e., $X_1 \cdot X_2$—where the element-multiplication operation is logical-and, and element-addition is logical-or. For the regex r^*, the entire set of substrings of ω matched by r^* can be computed by taking $X^0 + X^1 + \cdots X^n$, where X is the matrix for substring acceptance by r, and the + operator denotes pointwise logical-or of the operand matrices. The + operator is also the interpretation of alternation in regular expressions.

This approach gives us a deterministic semantics for regular expressions, but another interesting challenge lies in how these semantics should be concretely embedded in an SMT solver. One could simply choose to encode the semantics directly using the theory of strings; some SMT solvers also directly support regular expressions as part of their theories. However, it is also the case that strings are poorly supported at best, especially in CHC solvers—in our experiments, Z3 would often throw segmentation faults when asked to solve CHCs that contained strings.

For this reason, we again exploit the generality of SemGuS, and develop a more solver-friendly semantics by encoding strings as sequences of integers. The base cases for regular expressions—matching single characters, ϵ, and ϕ—can be encoded using equality between integers, and the rest of the semantics is compositional. Figure 5 shows the final rule for the term r^*: the input string is encoded as a sequence of integers $s_0, s_1, \cdots, s_n$. The output is a matrix of Boolean variables X as described above: the output argument of Star_ϵ shows the substring matrix that matches ϵ (i.e., the identity matrix I). The input parameter k in Star iterates through Star to compute X^k, along with the outputs of sem_R which show how concatenation is computed by matrix multiplication. A regular expression r accepts ω iff in X_r, the matrix for r, $X_r[0, n] = \mathsf{true}$.

7 IMPLEMENTION AND OPTIMIZATIONS

In this section, we describe our implementation of MESSY, a solver for SemGuS problems, as well as some optimizations that were applied in MESSY.

$$\frac{}{\mathsf{sem}_R(\langle (s_0, s_1, \cdots, s_n), 0, r^* \rangle, I)} \ \text{Star}_\epsilon$$

$$\frac{\mathsf{sem}_R(\langle (s_0, s_1, \cdots, s_n), k, r \rangle, X) \quad \mathsf{sem}_R(\langle (s_0, s_1, \cdots, s_n), k-1, r^* \rangle, X_{k-1})}{\mathsf{sem}_R(\langle (s_0, s_1, \cdots, s_n), k, r^* \rangle, X \cdot X_{k-1} + X_{k-1})} \ \text{Star}$$

Fig. 5. Semantics for the regular expression r^*, where $(s_0, s_1, \cdots s_n)$ represents a sequence of integers that encodes the input string.

7.1 Implementing MESSY

At a high level, MESSY accepts SemGuS problems and encodes them as CHCs using the encoding in §5. It then passes the CHCs to Z3 [De Moura and Bjørner 2008], which performs the actual proof search and produces an answer. The output from Z3 is either UNSAT, which means the problem is unrealizable, or a proof for *Realizable* from Equation (8) using the inference rules from the SemGuS problem: in this case, MESSY can extract a solution to the SemGuS problem from the proof.[7] We note that the capability of MESSY to synthesize programs also allows it to perform CEGIS for both program synthesis and unrealizability, which is unsupported in previous work on proving unrealizability [Hu et al. 2019, 2020].

We report here that Z3 itself varied in performance depending on whether particular internal flags were enabled.[8] While enabling these flags are the default setting for Z3 and result in better performance, they also made it difficult to recover the term representation from the output of Z3 (which is required to synthesize a term). Thus, during our evaluation in §8, we disabled the flags; MESSY can also be configured to run with the flags enabled.

In §5, we looked at different ways of translating SemGuS problems into CHCs depending on whether trees or listings are used to represent terms. MESSY supports three configurations for representing terms—a configuration that uses algebraic datatypes to model trees, and two configurations that respectively use lists and arrays to encode listings. In addition, MESSY also implements a SemGuS-specific optimization called the *fused semantics* , described in §7.2. For regular expressions, MESSY simply implements the semantics in §6.4 with terms encoded as arrays.

We also implemented an enumerative solver that uses the same CHC formalism, but replaces the syntax relations with concrete term representations (encoded using arrays) instead. This solver enumerates terms and checks whether they are correct using the given semantics—i.e., for each enumerated term the CHCs are then again passed to Z3, which will verify whether the given term is correct or not. The enumerative solver can be treated as a baseline alternative solver for SemGuS problems that plugs into MESSY (which is responsible for generating CHC scripts for Z3 to solve).

As presented throughout our paper, MESSY supports various SemGuS problems defined over many different domains and term representations. Implementing a new domain for SemGuS problems consists of two steps: a theoretical step where one develops a semantics for the domain on paper, then an implementation step to actually implement the domain. The generality of SemGuS often results in the first step requiring much more thought and effort: as shown in §8, different ways of encoding semantics can lead to big differences in performance, and one must also consider the correctness of the semantics with respect to CHC solving, as discussed in §6.4.

On the other hand, once the first step is completed, or if one is able to use a standard semantics, expressing the semantics as concrete semantics is a routine task. For example, the vectorized semantics detailed in §6.1 totals around 350 lines of code in our implementation; the regex semantics from §6.4 totals around 100 lines,[9] which mostly implement pattern matching on terms and encoding

[7]Z3 may also time out, or produce an error for various reasons, for example when dealing with algebraic datatypes.
[8]The particular flags are fp.xform.slice, fp.xform.inline_linear, and fp.xform.inline_eager.
[9]The semantics were written in a DSL we designed to interface with the SMT-LIB CHC format for easy development.

SMT queries. Moreover, given a library of various semantics for operators, defining a new SemGuS problem is just as easy as defining a SyGuS problem.

7.2 Optimizing Imperative SemGuS Problems with Fused Semantics

MESSY offers an optimization that utilizes a slightly different method of encoding syntax and semantic rules: instead of building a term using the syntax rules and propagating it through the semantic rules separately, one can also think of a scheme where the semantics of a term is executed on-the-fly while the term is being constructed. We refer to this kind of encoding as the *fused semantics.* Fused semantics are *different* from supplying SemGuS with a different semantics, because they are *derived* from an original semantics that SemGuS is supplied with. Instead, one may think of them as an *optimization* for SemGuS problems over subgrammars of G_{impv}.

The key idea for fused semantics is to modify the syntax relations so that they can check semantics as well as the syntactic structure, and modify the syntax rules accordingly as well. Thus, a syntax relation is now defined over three inputs—a term t, an input state Γ, and an output value v. The relation should evaluate to true if and only if t is a valid term, and $[\![t]\!](\Gamma, v)$ is also true. Generally, the syntax rule for a production $A_0 \rightarrow \sigma(A_1, \cdots, A_i)$ $\textcircled{n}$, again equipped with the semantics $\phi \wedge \mathsf{sem}_{A_1}(\langle\Gamma_1, t_1\rangle, v_1), \cdots, \mathsf{sem}_{A_i}(\langle\Gamma_i, t_i\rangle, v_i) \implies \mathsf{sem}_{A_0}(\langle\Gamma, t\rangle, v_0)$, can be generated in the form of Equation (17): the structure of Equation (17) matches *exactly* the structure of the supplied semantics (using the listing representation of terms).

$$\frac{\phi \quad \mathsf{syn}^{\mathsf{fused}}_{A_1}(\langle\Gamma_1, \mathcal{L}_{in}\rangle, \langle v_1, \mathcal{L}_{mid}\rangle) \cdots \mathsf{syn}^{\mathsf{fused}}_{A_i}(\langle\Gamma_i, \mathcal{L}_{mid}\rangle, \langle v_i, \mathcal{L}_{out}\rangle)}{\mathsf{syn}^{\mathsf{fused}}_{A_0}(\langle\Gamma, \mathcal{L}_{in}\rangle, \langle v, \textcircled{n} :: \mathcal{L}_{out}\rangle)} \; syntax^{\mathsf{fused}}_{A_0 \rightarrow \sigma(A_1, \cdots, A_i)} \; \textcircled{n} \tag{17}$$

One may ask how the fused semantics is different from a set of CHCs where the syntax relations have been eliminated, and the ordinary semantic rules implicitly check the syntax of a term while computing its semantics as well. They are indeed similar, but there is a subtle difference in the order of the arguments: the semantics from Equation (17) have $\mathcal{L}_{in}$ as the first argument, and $\mathcal{L}_{out}$ as the second; whereas ordinary semantic rules have the list order reversed.[10] Intuitively, the specification of the fused semantics constructs a term while computing the semantics on the fly, while the specification of the ordinary semantics can only compute the semantics for a fully constructed term.

The new encoding presented in Equation (17) is enough to allow *only the syntax rules* to describe both the syntax and semantics of terms within a SemGuS problem, provided that the grammar does not contain productions with while loops. However, productions that contain loops, such as $N \rightarrow \mathsf{while}\ B\ \mathsf{do}\ S$ $\textcircled{n}$, require a separate procedure because there must be a guarantee that the *same loop body* is synthesized for each iteration. To ensure that the same loop body is synthesized, one can either impose an additional constraint that states that each synthesized loop body must be identical, or more simply, one can apply the semantic relations described from §5 instead.

$$\frac{\mathsf{syn}^{\mathsf{fused}}_{B}(\langle\Gamma, b\rangle, \mathsf{true}) \quad \mathsf{syn}^{\mathsf{fused}}_{S}(\langle\Gamma, s\rangle, \Gamma_1) \quad \mathsf{sem}_{N}(\langle\Gamma_1, \mathsf{while}\ b\ \mathsf{do}\ s\rangle, \Gamma_2)}{\mathsf{syn}^{\mathsf{fused}}_{N}(\langle\Gamma, \mathsf{while}\ b\ \mathsf{do}\ s\rangle, \Gamma_2)} \; syntax^{\mathsf{fused}}_{N \rightarrow \mathsf{while}\ B\ \mathsf{do}\ S} \; \textcircled{n} \tag{18}$$

Consider the rule given in Equation (18). Note that the first two relations from the premise are syntax relations that both synthesize a term and execute its semantics. In contrast, the third relation is a semantic relation, which is defined identically to the semantic relations in §2. The semantic relations do not suffer from the problem of having to synthesize the same loop body over multiple

[10]This subtle difference also has a difference in performance—syntax-relation-less CHCs perform similarly to SemGuS problems with syntax relations, which perform worse compared to the fused semantics as shown in §8.

iterations. The idea here is that the syntax relations synthesize the loop body on the first iteration, then pass the representation to the semantic relations for subsequent iterations.

Finally, multiple sem_N premises in the Query rule must be rewritten as $\text{syn}_N^{\text{fused}}$ as well; when using non-vectorized semantics, this approach raises the same problem of potentially synthesizing different solutions for each example, and necessitates a constraint to ensure that all generated representations are identical. Although it is possible to use fused semantics for non-vectorized semantics in this way, MESSY employs the fused semantics as an optimization to vectorized semantics only, which does not suffer from this problem, because there is only a single vector of examples.

8 EVALUATION

In this section, we evaluate the feasibility of our algorithm to solve SemGuS problems through our implementation MESSY. Specifically, we investigate the following four issues:

Q1: We evaluate the effectiveness of MESSY on SyGuS benchmarks.
Q2: We evaluate the effectiveness of MESSY on imperative program-synthesis problems.
Q3: We evaluate the effectiveness of the optimizations discussed in §5 and §7.
Q4: We evaluate the effectiveness of approximate semantics supplied to MESSY.
Q5: We evaluate the effectiveness of MESSY on regular-expression benchmarks.

Overall, our evaluation is tilted towards proving unrealizability compared to synthesizing programs. This is because because there already exist many program synthesizers that incorporate multiple years of engineering effort [Barrett et al. 2011; Reynolds et al. 2015; Solar-Lezama 2013]; it is beyond the scope of this paper and MESSY to directly compete with these synthesizers.

8.1 Benchmarks

We performed our evaluation using three sets of benchmarks.

The first set consists of 132 unrealizable variants of the 60 LIA (Linear Integer Arithmetic) benchmarks from the LIA SyGuS competition track. These benchmarks were generated by Hu et al. [Hu et al. 2019] and have been used as benchmarks for unrealizability in previous work [Hu et al. 2019, 2020]. These benchmarks originate from SyGuS problems where the goal is to prove that a synthesized solution is optimal with respect to some metric [Hu and D'Antoni 2018] (e.g., minimizing the number of If-Then-Else operators). One can prove optimality by proving that a solution with a lower score is unrealizable.

In each of the benchmarks, the grammar that specifies the search space is recursive, and hence generates infinitely many LIA terms. These benchmarks are unrealizable because they contain grammars that restrict how many times a certain operator (e.g., plus or if-then-else) can appear in the solution. To see how effective MESSY is as a synthesizer, we also test MESSY on the 60 original LIA SyGuS benchmarks in §8.2. These benchmarks have a completely unrestricted grammar, as opposed to the 132 unrealizable variants generated from them.

The second set consists of 289 imperative SemGuS problems defined over various fragments of the imperative grammar G_{impv}. Out of these, 36 benchmarks were created by hand from common imperative programming questions, such as synthesizing a Fibonacci function or swapping variables using bitwise-xor. The remaining 253 benchmarks were derived by using the 30 benchmarks employed in a previous paper on synthesizing imperative programs via enumeration [So and Oh 2017] as a template. Out of the 30 templates, we ignored 7 that contained division, on which Z3 would return an error, and derived 11 benchmarks from each of the 23 remaining templates for a total of 253 benchmarks. The 23 base templates consist largely of two categories: those that compute a function over a range of numbers 1 to n using a loop (such as factorials or sums), and those that

Table 1. Number of solved benchmarks for various configurations of MESSY, alongside results for Nay, ESolver, CVC4, and SIMPL. ✗ indicates cases where the tool is non-applicable. SIMPL could only be evaluated on 23 realizable imperative benchmarks, because SIMPL cannot accept a grammar.

	Solver	SyGuS		Imperative		Regex
		Realizable	Unrealizable	Realizable	Unrealizable	Realizable
	Nay	✗	70	✗	✗	✗
	ESolver	6	✗	✗	✗	✗
	CVC4	59	✗	✗	✗	✗
	SIMPL	✗	✗	23*	✗	✗
	AlphaRegex	✗	✗	✗	✗	25
MESSY	Total	4	66	8	112	5
	Fused Trees	3	66	5	31	✗
	Fused Lists	4	66	5	62	✗
	Fused Arrays	2	64	6	91	✗
	Vectorized Arrays	2	66	5	56	✗
	Individual	0	56	3	10	0
	Abstract	✗	18	✗	37	✗
	Underapproximate	✗	✗	6	✗	✗
	Enumerative	✗	✗	2	✗	5
	Total benchmarks	60	132	67	222	25

compute a function over an array, again using a loop to iterate (such as finding the maximum element of an array, or adding two arrays together). To derive our benchmarks, we first instantiated SemGuS with the problem specification and the unbounded grammar G_{impv} with a restriction on the number of loops: the grammar in this case replicates the templates used to specify the search space from [So and Oh 2017]. Then, various restrictions were imposed on the grammar, such as limiting the number of statements allowed, or limiting the kinds of expressions that can occur as the loop condition. Out of the 11 benchmarks generated from each template, 2 were designed to be realizable, and 9 to be unrealizable. We developed our own set of benchmarks this way because the unrealizability of imperative programs is a previously unstudied field.

The final set of benchmarks consists of the 25 regular-expression problems from textbooks on automata theory, where the specification is given as a set of positive and negative examples, averaging around 10 examples total [Lee et al. 2016].

Each benchmark was given 10 minutes to complete on a machine with a 2.6GHz Intel Xeon processor with 32GB of RAM, with version 4.8.9 of Z3 as the external CHC solver. We note that the front-end processing step, to encode a SemGuS problem into CHCs, took less than 3 minutes for all of our benchmarks and configurations combined; the 10-minute timeout was separate from the front-end processing step, and devoted entirely to CHC solving.

Table 1 summarizes the numbers of solved benchmarks for various configurations of MESSY we tested, as well as comparisons for Nay [Hu et al. 2020], ESolver [Alur et al. 2017b], CVC4 [Reynolds et al. 2015], SIMPL [So and Oh 2017], and AlphaRegex [Lee et al. 2016].

8.2 Q1: Evaluating MESSY on SyGuS Benchmarks

In this section, we evaluate the effectiveness of MESSY on SyGuS benchmarks by comparing it against Nay, the state-of-the-art tool for proving unrealizablity for SyGuS problems, and against the SyGuS synthesizers CVC4, which was run using the default settings from SyGuS-COMP 2019 [Reynolds et al. 2019], and ESolver. It is worth noting that both CVC4 and ESolver solve synthesis problems directly, while MESSY relies on an external CEGIS loop to generate examples. We did not test the enumerative SemGuS solver described in §7 for SyGuS benchmarks, because ESolver already serves as a basic enumerator.

Like MESSY, Nay checks the (un)realizability of a SyGuS problem when the specification is given as a set of examples; unlike MESSY, however, Nay cannot synthesize a solution for realizable problems. We used the set of 132 unrealizable SyGuS benchmarks described in §8.1 for evaluation. We report that MESSY or Nay solves a problem if it can solve a problem using any of its configurations—for MESSY, this encompasses the three different term representations, as well as the different semantics that SemGuS can be instantiated with.

We implemented a CEGIS algorithm for MESSY, and compared it against the CEGIS algorithm of Nay. Because Nay is incapable of synthesizing an answer to realizable a SyGuS problem, the CEGIS loop of Nay relies on an external synthesizer ESolver to produce a term. Because MESSY and Nay rely on different methods to produce counterexamples, their CEGIS iterations may differ.

With the standard CEGIS algorithm, MESSY can prove 61/132 benchmarks unrealizable, while Nay can do so for 65/132. Nay also provides a modified "random" variant of CEGIS that allows random examples to be added to the set of counterexamples throughout the CEGIS loop. Using this technique, Nay can prove unrealizability for an additional 5 benchmarks. We ran MESSY on the same set of examples produced using this technique; it was able to prove unrealizability for the same 5 benchmarks as well, for a total of 66/132 benchmarks. There are 67 benchmarks where both solvers timed out—stuck at some iteration of the CEGIS loop. On 17 of them, MESSY can complete more iterations of the CEGIS loop than Nay (avg. 6.1 for MESSY vs. 4.8 for Nay). On 13 of them, Nay can progress further (avg. 2.2 for MESSY vs. 3.1 for Nay). The two solvers were stuck at the same iteration on the rest of the benchmarks.

Next, we compared the abilities of MESSY as a SyGuS synthesizer on the 60 original LIA benchmarks upon which the unrealizable benchmarks were derived. MESSY solved 4/60 benchmarks. This number is comparable to ESolver, which solved 6/60 benchmarks and was the winner of the first SyGuS competition in 2014. Moreover, MESSY solved one benchmark that ESolver could not solve. While MESSY is not competitive with current SyGuS solvers, such as CVC4 [Reynolds et al. 2015], which solved 59/60 benchmarks, the fact that its performance is already comparable with an early version of a SyGuS solver is encouraging, and one might hope that more efficient algorithms for synthesizing solutions to SemGuS problems are possible in the future.[11]

MESSY was efficient at proving unrealizability: 56 out of the 66 benchmarks solved were solved in under 10 seconds, and MESSY also has comparable runtimes with Nay. For synthesizing programs, ESolver solved all solvable benchmarks in under two minutes each, while MESSY required 6 minutes for two of the solved benchmarks.

To answer Q1: MESSY *is quite effective on unrealizable SyGuS problems, to a degree that is comparable with Nay, and can also synthesize solutions for realizable SyGuS problems*: in particular, MESSY is more general than previous tools as it can solve non-SyGuS problems, and can produce two-sided answers to synthesis problem.

8.3 Q2: Evaluating MESSY on Imperative Synthesis Benchmarks

In this section, we evaluate the effectiveness of MESSY by seeing how well it can deal with imperative synthesis benchmarks. We consider SemGuS-with-examples problems, as opposed to ordinary SemGuS problems, due to the challenges of checking whether an imperative program

[11]We also note that the LIA SyGuS benchmarks have an entirely free grammar and are single-invocation, which allows CVC4 to use a specialized method involving quantifier elimination to synthesize programs.

$$\begin{array}{lcc} Start & ::= & \text{while } B \text{ do } S \\ S & ::= & x := E \mid y := E \mid S; S \\ E & ::= & x \mid y \mid E + E \mid E - E \\ B & ::= & E < E \end{array}$$

(a) A grammar for a benchmark where the goal is to synthesize a program adding integers from 1 to n.

$$\begin{array}{lcc} Start & ::= & \text{while } B \text{ do } S \\ S & ::= & x := E \mid S; S \\ E & ::= & x \mid y \mid E + E \mid 0 \mid 1 \mid -1 \\ B & ::= & E < E \end{array}$$

(b) A grammar for a benchmark where the goal is to synthesize a program resulting in $x == y$.

Fig. 6. Grammars for example benchmarks that were proved unrealizable.

satisfies a specification or not (which makes it difficult to implement a CEGIS loop). In principle, one could implement a CEGIS loop using an external verifier.

Out of our 289 imperative benchmarks, a total of 67 were designed to be realizable, while the remaining 222 were designed to be unrealizable. As shown in Table 1, MESSY solved 8/67 realizable benchmarks, and 112/222 unrealizable benchmarks, for a total of 120 benchmarks solved.

Out of the 120 solved benchmarks, 10 benchmarks—2 realizable, and 8 unrealizable—were those with infinite syntactic search spaces and also contained the possibility of an infinite loop. MESSY also solved 15 benchmarks that did not contain loops, but nevertheless had infinite syntactic search spaces. Overall, MESSY had more success with proving unrealizability than synthesizing programs for both SyGuS and imperative benchmarks, especially when a simple lemma proved sufficient for showing unrealizablity of the whole SemGuS problem (such as the example given in §2).

One benchmark proved unrealizable in a similar manner uses the grammar shown in Figure 6a, where the specification was to compute the sum of integers from 1 to x and store it in y. Here, Z3 inferred the lemma that for the input example $[(x, y)] = [(2, 0)]$, the values of x and y remain even regardless of the program being considered, because the constant 1 is not present in the grammar—and thus cannot reach the correct output $y = 3$.

An example of a lemma that makes use of multiple input examples comes from a benchmark that uses the grammar shown in Figure 6b, where the goal was to set x to be equal to y by modifying x in a while loop. This problem is unrealizable because the value of x can increase or decrease as the loop iterates, but cannot do both (based on what input is given). When given the input examples $[(x, y)] = [(12, 20), (20, 12)]$, Z3 infers the lemma that ultimately when the loop terminates, "$x > 12$ or $x < 20$". If $x > 12$, the second example is unsatisfiable, and if $x < 20$, the first example is unsatisfiable—thus Z3 is able to prove the whole synthesis problem as unrealizable.

Another reason why MESSY performs better on unrealizable problems is in part due to how the generated CHCs are dealt with internally in Z3—as described above, Z3 proves unrealizability by discovering a lemma that conflicts with the specification. For realizable problems, however, Z3 in the worst case must conduct a search over all possible concrete terms from a possibly infinite search space, in a process similar to generate-and-test. The authors are unsure of whether Z3 is capable of discovering lemmas that can be used to prune the search space for realizable benchmarks; regardless of the answer, the results suggest that program execution expressed as CHC solving introduces overhead that is large enough to make synthesis relatively more difficult compared to proving unrealizability.

The baseline enumerative solver for SemGuS succeeded in solving only 2 realizable benchmarks. Interestingly, the runtimes for enumerative scripts (each of which verifies a candidate term) displayed a high amount of variance, with some scripts terminating in under a second, and others exceeding the timeout of 10 minutes. There was no clear pattern behind the variation, except that terms containing loops tended to time out more compared to those that did not contain loops. This result is in line with the hypothesis about overhead in program execution expressed as CHC solving above—it seems that the internal algorithms of a CHC solver suffer for the relatively simpler task of checking whether a proof tree is correct.

In contrast, SIMPL [So and Oh 2017], whose benchmarks we use in our evaluation of MESSY, employs a strategy of performing static analysis in tandem with enumeration; SIMPL also employs heuristics that prefer smaller programs, and directly executes candidates to see if the specification is met. This enumerative approach makes SIMPL perform better as a synthesizer: SIMPL solves the full set of 23 realizable benchmarks upon which our benchmarks are based, while MESSY can solve none. However, SIMPL is incapable of proving unrealizability, because it is based on enumeration. One also cannot express a syntactic search space in SIMPL outside of simple templates, which prevented us from running the rest of our realizable benchmarks on SIMPL.

MESSY took less than 10 seconds to solve 82/120 benchmarks, and the other 12 benchmarks required more than a minute to complete. Whether a benchmark contained an unbounded loop or an infinite search space seemed to have little correlation with the runtimes: there were finite-search space benchmarks that took over a minute to complete, and benchmarks with both unbounded loops and infinite search spaces that took less than a second to complete. This phenomenon suggests the importance of discovered lemmas in solving SemGuS problems: given a strong lemma, a SemGuS problem can be solved quickly, even with infinite loops and search spaces. On the other hand, without a lemma, the problem can take a long time to solve even if the search space is finite.

To answer Q2: MESSY *is capable of solving SemGuS problems with infinite search spaces and imperative semantics, especially if the given problem is unrealizable.* Notably, MESSY is the first tool that can prove unrealizablity for imperative synthesis problems.

8.4 Q3: Evaluating Optimized Methods for Solving SemGuS Problems

In this section, we compare the effectiveness of the various optimizations we described in §5 and §7 for solving SemGuS problems. Specifically, we investigate the following two issues:

(1) We assess the effectiveness of the flattened term representation from §5.2, by comparing the performance of MESSY configured to use trees, lists, and arrays as the term representation.
(2) We assess the effectiveness of vectorized and fused semantics, by comparing the performance of MESSY on (i) individual semantics, (ii) vectorized but non-fused semantics, and (iii) vectorized and fused semantics.

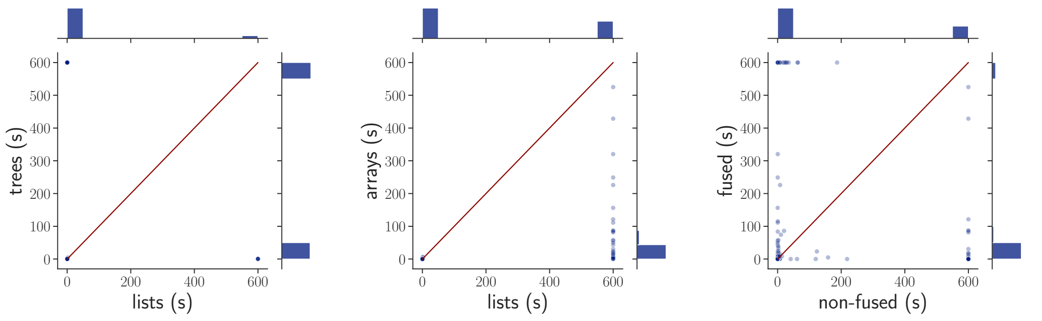
Effectiveness of Flattened Term Representations. To evaluate the effectiveness of the three term representations, we supplied SemGuS with vectorized semantics and enabled the fused-semantic-optimization, the configuration that yielded the overall best results in our evaluation. In this section, we say that a particular term representation "solved" a benchmark if Z3 was able to solve the CHCs produced by encoding the SemGuS problem using the given term representation.

The 'Fused Trees', 'Fused Lists', and 'Fused Arrays' rows of Table 1 summarize the results for the different term representations: for SyGuS benchmarks, all three representations were similar. For imperative benchmarks, the array representation is clearly better compared to the list and tree representations: in particular, the tree representation could only solve one benchmark that the array representation could not solve, while the list representation solved a strict subset of the benchmarks solved by the array representation.

Figures 7a and 7b compare the performances of the list versus tree representations, and the list versus array representations, on imperative benchmarks solved by at least one of the representations.

As mentioned in §5.2, we suspect the differences between the different term representations is due to the fact that support for algebraic datatypes in Z3 still remains relatively limited.[12] When

[12]In our correspondence of the authors of Z3 and Spacer, they mentioned that using inductive datatypes with the Horn Clause solver was highly experimental.



(a) Runtime for list representations vs. tree representations, for imperative benchmarks solved by at least one configuration.

(b) Runtime for list representations vs. array representations, for imperative benchmarks solved by at least one configuration.

(c) Runtime for the non-fused vs. fused vectorized semantics, for benchmarks solved by at least one configuration.

Fig. 7. Runtime comparisons for various configurations of MESSY. Bar graphs on the outer axes show the distribution of the data points. 600 seconds indicates a timeout.

the tree or list representations were used, Z3 terminated with the error "stuck on a lemma" far more often than when the array representation was used.

To answer part (1) **of Q3:** *Flattened term representations are indeed effective, especially when using arrays to avoid the use of algebraic datatypes altogether.*

Effectiveness of the Vectorized and Fused Semantics. To evaluate the effectiveness of the vectorized and fused semantics optimizations, we compared them against each other, and against individual semantics that are not vectorized nor fused (corresponding to the "standard" form of semantics mentioned in §6). We note that while the vectorized semantics is actually a different semantics that SemGuS can be supplied with, MESSY can automatically vectorize the semantics for any subgrammar of G_{impv} as an optimization; thus we treat it as an optimization for our evaluation.

Individual semantics vs. Non-fused vectorized semantics. Using a list representation of terms, the individual semantics could solve 56 unrealizable SyGuS benchmarks, a strict subset of the 66 solved by the vectorized semantics. For imperative benchmarks, the individual semantics could only solve 3 realizable and 10 unrealizable benchmarks, compared to 5 realizable and 62 unrealizable when using the vectorized semantics.

Non-fused vectorized semantics vs. Fused vectorized semantics. The "Fused Arrays" and "Non-Fused Arrays" rows of Table 1 describe the number of solved benchmarks: again, performance for the SyGuS benchmarks was similar. For imperative benchmarks, the non-fused vectorized array semantics solved 5 realizable and 56 unrealizable benchmarks, compared to 6 realizable and 91 unrealizable for the fused vectorized array semantics.

Figure 7c compares the results of the non-fused vectorized versus the fused vectorized semantics using array representations: while the fused semantics solve more benchmarks, the graph suggests that the fused semantics are not strictly better than the non-fused semantics. In particular, there are 11 unrealizable benchmarks that only the non-fused semantics can solve.

When using a list representation, the difference becomes less pronounced—the non-fused vectorized semantics can solve 4 realizable and 58 unrealizable benchmarks, compared to 5 realizable and 62 unrealizable for fused vectorized list semantics. We think the reason is again the limited support in Z3 for algebraic datatypes, which remains the main bottleneck when using list representations.

To answer part (2) **of Q3:** *The fused vectorized semantics is effective as an optimization, especially for imperative benchmarks, but there exist some benchmarks for which the non-fused vectorized semantics performs better.* Both are consistently better than the individual semantics.

8.5 Q4: Evaluating SemGuS and MESSY with Approximate Semantics

We evaluated how well MESSY performs when it is instantiated using an approximate semantics to produce one-sided answers to either synthesis or unrealizability. In this section, we focus on identifying the number of *new* benchmarks that an approximate semantics can solve compared to the standard version of the semantics. This approach is motivated by the fact that the nature of an approximate semantics can sometimes change the kind of answer that can be obtained—for example, using an abstract semantics for an unrealizable problem might make it realizable—and thus a direct performance comparison makes little sense.

Both an abstract and underapproximating semantics were implemented using arrays as the term representation, with the fused-semantics and vectorizing optimizations, which displayed the best overall performance in §8.4. The new benchmarks solved are ones that the exact, fused and vectorized array semantics was unable to solve.

Abstract semantics for unrealizability. To test the capabilities of abstract semantics, we implemented five variants of the abstract domain $\mathbb{B}_3$ from §6.2, where each domain tracked the first, second, third, fourth, and fifth bit of variables, respectively. This choice was driven by the fact that most of the input examples for our imperative synthesis benchmarks were small, between 0 to 31.

The “Abstract” row of Table 1 describes the number of benchmarks solved using the abstract semantics. The abstract semantics did not make a difference for the SyGuS benchmarks; all benchmarks that were solved by the abstract semantics were also solvable by the exact semantics. However, for the imperative benchmarks, the abstract semantics was able to solve 17 unrealizable benchmarks that the exact semantics could not solve. Using abstract semantics yielded faster runtimes: the abstract semantics timed out for at most 15 benchmarks regardless of the variant of abstract domain, compared to over 200 for the exact semantics (although realizability results in the abstract semantics have no meaning).

One reason that the abstract semantics failed to make a difference on the SyGuS benchmarks could be that the SyGuS benchmarks themselves used inputs with very small values, often between 0 and 7: the abstract semantics were able to prove some SyGuS benchmarks as unrealizable using the lower bits, but nothing new. In addition, the abstract domains that we used do not work well in the presence of addition, because carry bits often render a result to be $\top$. All of our SyGuS benchmarks contain addition, while some imperative benchmarks do not.

Underapproximating semantics for program synthesis. For the underapproximating semantics, we implemented the technique of bounding the number of loop-unrollings from §6.3, and experimented with loop bounds of 10, 50, and 100. We only compare the imperative benchmarks here, because the SyGuS benchmarks do not contain loops. The “Underapproximate” row of Table 1 describes the number of benchmarks solved using this semantics. The bound semantics was able to synthesize one more program compared to the non-bound semantics. Interestingly, MESSY succeeded in synthesizing the program (to compute the factorial function using a while loop) when the bound was set to 100, but not when the bound was 10 or 50. The small difference in performance may be due to the fact that MESSY generally performs worse as a synthesizer than a tool for proving unrealizability. The results also tell us that synthesizing imperative programs is difficult, even without the presence of infinite loops: it could be because unrolled loops still pose a significant burden when trying to compute the semantics of an imperative program, especially because our approach must ultimately prove that a candidate term satisfies the specification using semantics encoded as CHCs (which is likely to be slower compared to direct execution).

To answer Q4: *Abstract semantics allows* MESSY *to solve many more unrealizable* SemGuS *problems compared to using only exact semantics. The bound underapproximating semantics did allow* MESSY *to solve more realizable* SemGuS *problems, but the improvement was small.*

8.6 Q5: Evaluating SemGuS and MESSY on Regular Expressions

Finally, we evaluated the effectiveness of MESSY on regular-expression benchmarks. The final column of Table 1 displays the results: the CHC-solving approach failed to solve any of the benchmarks within 10 minutes, while the enumerative approach described in §7 solved 5 benchmarks. This is an interesting case where the enumerative approach outperformed the CHC-solving approach.

The most likely hypothesis for this behavior is that the iterative nature of the Star rule in Figure 5 (due to the unrolling parameter k), as well as the behavior of strings themselves, mixes poorly with the overarching principle of finding a powerful lemma or invariant. In addition, we did not impose restrictions on the grammar for regular expressions, which made all benchmarks realizable.

The enumerative solver, while succeeding in solving 5 benchmarks, displayed behavior similar to what it displayed when solving imperative benchmarks: the variance between termination times was high, without a clear pattern behind the variance. In contrast, AlphaRegex [Lee et al. 2016], from which we took our benchmarks, reports being able to solve all 25 benchmarks; we believe this difference is due to the overhead in using a CHC solver instead of an automaton to check regexes, as well as the additional enumeration optimizations that AlphaRegex employs.

To answer Q5: *Regular-expression synthesis problems can be expressed in MESSY.* MESSY could solve some of the benchmarks using enumeration, but none of the benchmarks using the CHC-solving method. The most likely explanation for this behavior is the difficulty of finding good lemmas for regex problems, upon which MESSY—more specifically, Z3 and Spacer—heavily relies.

9 RELATED WORK

General Synthesis Frameworks. Sketch [Solar-Lezama 2013] and Rosette [Torlak and Bodík 2014] are both solver-aided languages, where one specifies a synthesis problem using a domain-specific language, which is translated into an SMT problem. FlashMeta [Polozov and Gulwani 2015] is a synthesis framework that allows one to specify the semantics of operators in the language using witness functions, which roughly correspond to the "inverse" semantics of operators. In these tools, the way synthesis problems are defined is directly tied with how they are solved: one needs to develop non-standard inverse semantics for FlashMeta, or phrase the synthesis problem within the language of Sketch or Rosette, which are requirements imposed by their particular synthesis algorithms. Due to these reasons, these tools disallow defining (and therefore solving) synthesis problems involving infinite search spaces.

The first attempt to unify these frameworks into a logical one was provided by SyGuS [Alur et al. 2013]. However, SyGuS is not general enough as it cannot express synthesis problems over arbitrary syntactic constructs that do not lie inside a decidable SMT theory. SemGuS, on the other hand, provides a *logical* way to *define* synthesis problems with *custom* semantics. Moreover, the solving procedure for SemGuS is *motivated by the definition*, not the other way around.[13]

Customizing Semantics in SyGuS. SyGuS allows one to provide semantics for user-defined terms, but the support is limited to functions/operators that can be used in the grammar. Concretely, if we consider our formalization of semantics (Definition 3.5), the degree of customization available

[13]One may argue that semantics expressed as a CHC is a restriction, but as stated in §1, they are more of a formalization. One may also assume a different surface syntax, such as the format in Equation (1); a translation to CHCs is straightforward.

in SyGuS is limited to customizing the constraint ϕ to a formula expressible in the background theory.[14] This limitation prevents SyGuS from expressing imperative program-synthesis problems; SemGuS eases this restriction by allowing one to replace ϕ with any first-order formula, as well as introduce new relations or arguments for the relations.

CVC4 also has a setting where one can encode a grammar via algebraic datatypes, and build an interpretation of these datatypes in terms of a background theory T into an SMT solver [Reynolds et al. 2015]. This feature is designed to extend the SMT-based synthesis approach of CVC4 towards grammars, and like SyGuS, is restricted to what one can express within the theory T. Semantics in SemGuS and MESSY are expressed using CHCs, which allow us to express a wider variety of semantics, and the solving procedure is a proof search over CHCs as opposed to solving a universally quantified SMT formula (as in CVC4).

Synthesis for Imperative Programs. There have been attempts at designing synthesizers specifically for imperative programs. Existing tools require the user to provide templates that specify most of the program [Solar-Lezama 2013; Srivastava et al. 2010]; the tools then resort to various constraint-solving techniques to complete missing parts of the template, which often do not contain loops [Srivastava et al. 2010].

SIMPL [So and Oh 2017] can synthesize imperative programs from input-output examples and a template that specifies most of the program. SIMPL employs a simple *enumeration-based* strategy, and uses abstract interpretation to rule out templates that will not result in a solution. Because SIMPL is based on enumeration, it performs well as a synthesizer. However, unlike MESSY, SIMPL cannot restrict the terms allowed in a program and it cannot establish that a problem is unrealizable.

Unrealizability. Nope [Hu et al. 2019] and Nay [Hu et al. 2020] are, to the best of our knowledge, the only two tools that can prove unrealizability for SyGuS benchmarks in which the grammar can generate infinitely many terms. Because Nay consistently outperforms Nope, we only compare against Nay in our evaluation. MESSY can solve synthesis problems over any specified language, including imperative languages, whereas both Nope and Nay can only solve SyGuS problems. One variant of Nay also uses Constrained Horn Clauses, which are used to encode the problem of solving a set of equations that describes the sets of possible outputs of the program. In MESSY, the constraints are used for describing both the syntax and the semantics of the programs in the search space. Because of the syntactic constraints, MESSY can extract the synthesized program when the problem is realizable, which Nay is unable to do.

There exist other tools that are capable of proving unrealizability in limited situations, such as CVC4 [Reynolds et al. 2015] or DryadSynth [Alur et al. 2017a]. However, CVC4 can only prove unrealizability when the grammar is completely unrestricted [Hu et al. 2019, 2020]. DryadSynth does not accept a grammar as part of its specification; MESSY is the only tool that can perform synthesis and unrealizabilty for general SemGuS problems.

The Use of Semantics in Program Synthesis. *Synthesis using abstraction refinement* (SYNGAR) [Wang et al. 2018b] uses predicate abstraction to prune the search space of a synthesis-from-examples problem. SYNGAR builds a tree automaton representing all trees in the search space that are correct with respect to an abstract semantics expressed using predicate abstraction. SYNGAR can be viewed as a special case of SemGuS in which predicate abstraction is used to *overapproximate* the semantics of terms in the programming language. SYNGAR's approach is tied to the use of an *abstract semantics* that can be expressed using a finite abstract domain, whereas

[14]To be precise, this formula is further limited to a formula of the form $v_0 = f(v_1, \cdots, v_i)$, where f is a non-recursive function expressible in the background theory. Notably, this prevents expressing relations between $v_1, \cdots, v_i$, the results of nonterminals in the RHS.

our approach extends to infinite domains. In particular, with our approach, one can express the *concrete semantics* of a programming language.

FlashMeta [Polozov and Gulwani 2015], is also a way of using semantics in program synthesis.

The Use of Horn Clauses in Program Synthesis. In Inductive Logic Programming (ILP) [Lavrač and Džeroski 1994; Muggleton 1991; Quinlan 1990], given background knowledge, typically in the form of Horn Clauses, the goal of ILP is to learn the defining formula for a logical relation that agrees with a given classification of input examples. Both ILP and our framework use Horn Clauses to specify background knowledge—which for our algorithm consists of the syntax and semantics of the target programming language. However, the respective goals for the output answer are different: (i) In ILP, the goal is to create a Horn-Clause program as the answer. (ii) In our algorithm for SemGuS, the goal is to create a program in the language that has been specified via the background knowledge. Whether ILP techniques can be adapted to SemGuS is left for future work.

10 CONCLUSION AND FUTURE WORK

This paper develops SemGuS, a new framework for program synthesis that allows one to specify both the syntax and the semantics of a synthesis problem. SemGuS can be used for specifying synthesis problems over an imperative programming language; it also allows one to work with a variety of different semantics that may be better suited to solve a synthesis problem efficiently. The paper also presents a general procedure for solving SemGuS problems capable of both program synthesis and proving unrealizability, and an implementation MESSY to solve SemGuS problems.

SemGuS opens many future directions of work.

Inferring Lemmas for SemGuS. As mentioned in §2, our procedure for solving SemGuS problems relies on an external CHC solver to infer lemmas over sets of programs in the syntactic search space, using the semantics of terms. While we have relied on an external solver (Z3) to perform this inference for us, it is also unclear to what degree CHC solvers are capable of discovering lemmas to prune a syntactic search space. An algorithm to explicitly infer lemmas and prune parts of the search space would be especially useful in enhancing our solving algorithm as a synthesizer, allowing MESSY to compete with state-of-the-art synthesizers as well.

SemGuS with Abstract Domains. As discussed in §6.3, SemGuS provides a natural framework for discovering and composing abstract domains for program synthesis. Coupled with the fact that lemmas play an important role in solving SemGuS problems, a system for inferring lemmas efficiently using abstract domains, and vice versa, has the potential to make SemGuS solvers more efficient. We have already witnessed this in our paper: the abstract domain $\mathbb{B}_i$ in §6.2 was inspired by the lemma from §2.2, which in turn proved the benchmark from Figure 6b unrealizable.

Specialized Solvers for SemGuS. One limitation of our approach was that the synthesis procedure was quite inefficient, especially for regular expressions. Since SemGuS now gives us a way to specify synthesis problems, is it possible to develop solvers that are specialized for specific SemGuS scenarios? An example of such an approach could be a better enumerative solver, which does not rely on CHC solving but employs a separate algorithm to check the validity of proof trees.

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 1420866, 1763871, and 1750965; by the Wisconsin Alumni Research Foundation; by a Facebook fellowship; and by a grant from the Korea Foundation for Advanced Studies. We also thank our shepherd, Nadia Polikarpova, for making numerous suggestions to improve the paper. Opinions, findings, or conclusions expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

REFERENCES

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), 2013.* IEEE, 1–8.

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017a. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438* (2017).

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017b. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 319–336.

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.* Springer, 171–177.

Régis Blanc, Ashutosh Gupta, Laura Kovács, and Bernhard Kragl. 2013. Tree interpolation in vampire. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning.* Springer, 173–181.

Edmund Clarke, Daniel Kroening, and Karen Yorav. 2003. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451).* IEEE, 368–371.

Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation frameworks. *Journal of logic and computation* 2, 4 (1992), 511–547.

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. http://dl.acm.org/citation.cfm?id=1792734.1792766

John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

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.* Springer, 335–352.

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 Conference on Programming Language Design and Implementation.* 1128–1142.

Qinheping Hu and Loris D'Antoni. 2018. Syntax-guided synthesis with quantitative syntactic objectives. In *International Conference on Computer Aided Verification.* Springer, 386–403.

S.C. Johnson. 1975. *YACC: Yet Another Compiler-Compiler.* Technical Report Comp. Sci. Tech. Rep. 32. Bell Laboratories.

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2020. Semantics-Guided Synthesis. *arXiv preprint arXiv:2008.09836* (2020).

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design* 48, 3 (2016), 175–205.

N. Lavrač and S. Džeroski. 1994. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood.

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing regular expressions from examples for introductory automata assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.* 70–80.

Kenneth L McMillan and Andrey Rybalchenko. 2013. Solving constrained Horn clauses using interpolation. *Tech. Rep. MSR-TR-2013-6* (2013).

S. Muggleton. 1991. Inductive logic programming. *New Generation Comp.* 8, 4 (1991), 295–317.

Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic repair of regular expressions. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. Swizzle inventor: data movement synthesis for GPU kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* 65–78.

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* 107–126.

J.R. Quinlan. 1990. Learning logical definitions from Relations. *Mach. Learn.* 5 (1990), 239–266.

Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. CVC4SY for SyGuS-COMP 2019. *arXiv preprint arXiv:1907.10175* (2019).

Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. 2015. Counterexample-guided quantifier instantiation for synthesis in SMT. In *International Conference on Computer Aided Verification*. Springer, 198–216.

Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*. 364–381. https://doi.org/10.1007/978-3-319-66706-5_18

Armando Solar-Lezama. 2013. Program sketching. *STTT* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 313–326. https://doi.org/10.1145/1706299.1706337

Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 530–541. https://doi.org/10.1145/2594291.2594340

Xinyu Wang, Greg Anderson, Isil Dillig, and Kenneth L McMillan. 2018a. Learning Abstractions for Program Synthesis. In *International Conference on Computer Aided Verification*. Springer, 407–426.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018b. Program Synthesis Using Abstraction Refinement. *PACMPL* 2, POPL (2018), 63:1–63:30.