POPL: G: Synthesis and Unrealizability of Imperative Programs via Semantics-Guided Synthesis

Jinwoo Kim
University of Wisconsin-Madison
USA
pl@cs.wisc.edu

Abstract

We study the problem of synthesizing imperative programs and its unrealizability, using a framework in which a user provides both the syntax and the semantics for operators in the language. The synthesis framework that we define is "semantics-guided": among the lemmas established by an underlying SMT solver during synthesis, some involve the semantics supplied by the client. We develop an algorithm to solve semantics-guided synthesis problems, implement our proposed technique in a tool called SIP, and apply it to both SyGuS problems (i.e., over expressions) and synthesis problems over an imperative programming language.

1 Problem and Motivation

Program synthesis refers to the task of finding a program within a given search space that meets a given specification, which is most often a logical formula or a set of input-output examples. A large subset of program-synthesis problems have been studied under the framework of *syntax-guided synthesis* (SyGuS) [1], in which a problem instance consists of (i) a regular-tree grammar describing the search space of programs, and (ii) a logical formula acting as the specification.

While existing SyGuS-based program synthesizers such as CVC4 [3] and EUSolver [2] have been quite successful, they also suffer from some limitations. One of these limitations is due to the SyGuS framework itself, which does not permit one to customize the semantics of a language at will, limiting its applicability to program-synthesis problems over a small number of theories, such as linear integer arithmetic (LIA). Consequently, this limitation has made it difficult to use SyGuS solvers to synthesize imperative programs, i.e. programs that contain state and loops.

Another limitation of existing solvers is their inability to prove the *unrealizability* of a synthesis problem, which refers to the situation in which a program meeting the specification does not exist within the given search space. Proving the unrealizability of synthesis problems has applications in synthesizing programs that are optimized with respect to some metric [7], and can be employed in tandem with general synthesis algorithms as well.

In this paper, we present a new synthesis framework, which can be used (among other things) to specify synthesis problems over an imperative programming language. Like SyGuS, our framework allows a user to specify a search space in the form of a grammar and a behavioral specification in the form of a logical formula. However, the framework also allows the user to define the *semantics* of operators in the grammar (which might contain assignment statements and loop constructs) in terms of a set of inference rules. Because of the inclusion of the semantic-specification component, we call this framework *semantics-guided synthesis* (SemGuS).

Following the definition of the SemGuS framework, we develop a procedure to solve SemGuS problems that is capable of both producing a synthesized program for realizable problems, and a proof of unrealizability for unrealizable ones. We then show how SemGuS and the solving procedure can be used to solve imperative synthesis problems that possibly involve an infinite search space containing loops.

2 Background and Related Work

Related Work. Imperative program synthesis has been an active area of research the past few years [8, 10–12]. Although there have been many approaches in the area, most approaches to imperative program synthesis employ a form of enumeration or focus on specific finite domains. These approaches, while effective in their respective domains, (i) often fail on general problems outside the target domain, and (ii) lack methods to prove unrealizability, which requires reasoning about *all* possible terms in the search space (as opposed to finding a single answer).

There has also been some successful recent work on proving the unrealizability of SyGuS problems [5, 6]. However, these tools work only on SyGuS problems, and are also limited to unrealizability in that they cannot produce an answer for realizable problems.

Our approach of solving SemGuS problems is the first that we are aware of that can prove that a synthesis problem for an imperative programming language is unrealizable, and

ACM Student Research Competition Grand Finals, 2020

is also capable of synthesizing imperative programs. SemGuS (and our algorithm) is also not limited to imperative synthesis problems, and can be used for regular SyGuS problems or other scenarios involving customized operators and semantics as well.

Challenges. SemGuS problems, especially those related to imperative program synthesis, pose several unique challenges in computing—or proving unrealizability of—their solutions.

- **Reasoning about imperative programs.** Unlike traditional SyGuS problems, which are typically defined over decidable theories such as linear integer arithmetic (LIA), imperative programs must deal with state, and typically lack a decidable theory for reasoning about programs and program fragments.
- **Loops.** Loops provide a double challenge in the context of program synthesis: each loop could have (i) an infinite number of elaborations (of the condition and the loop-body), each of which may have (ii) an arbitrary number of iterations. Thus, a synthesis algorithm must have some way of reasoning about *sets* of possible elaborations instead of reasoning about *individual* loop elaborations—otherwise, the search space becomes intractable.
- **Proving unrealizability.** To prove the unrealizability of a synthesis problem, one must show that *all* programs in a possibly infinite search space fail to satisfy the specification. This rules out methods such as enumeration, and, like loops, requires an algorithm capable of effectively reasoning about sets of possible elaborations.

The Solution. Our key idea is to reduce the SemGuS problem into a proof-search problem over constrained Horn clauses (CHCs), which are used to describe both the syntax and semantics of programs in the search space, as well as the specification. The reduction of a SemGuS problem into CHCs plays a central role in making the problem computationally tractable, thanks to recent advances in CHC solving [9].

If a proof exists, using the given CHCs, that the specification can be satisified, the problem is realizable—and an answer may be extracted from the proof. If, on the other hand, the solver can prove that the specification cannot be proved using the given rules, the problem is unrealizable.

Contributions. To summarize, this paper makes the following contributions:

- The extension of SyGuS to SemGuS by allowing 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.
- A constraint-based approach for solving SemGuS problems using constrained Horn clauses, together with an optimized instantiation of this approach for solving SemGuS problems over an imperative programming language.
- An implementation of a SemGuS solver using Z3 [4, 9], called SIP (§4). We find that SIP performs comparably to the two other tools that are capable of proving unrealizability for SyGuS problems, while adding the ability to synthesize a program for problems that are realizable (which the other tools cannot do).

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

Figure 1. Example grammar G_{ex}.

3 Uniqueness of the Approach

In this section, we describe the core idea behind SemGuS problems, alongside an illustrative example.

Consider the unrealizable problem of computing the bitwise-xor of two variables x and y using only bitwise-and and bitwise-or operations and no auxiliary variables, where the result is stored in variable x. We show how one can formulate this as a SemGuS problem, and how our tool SIP will automatically prove that the problem is indeed unrealizable.

3.1 The SemGuS Framework

A SemGuS problem consists of three components - (i) a search space given by a regular tree grammar G, (ii) a semantics for the grammar G, and (iii) a specification ψ of the desired behavior of the program.

In our example, the grammar G_{ex} in Figure 1 acts as the first component, which 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 the language $L(G_{ex})$ is not expressible using SyGuS due to the presence of assignments and loops.

The next component of a SemGuS problem is a semantics for the given language, which in this case is $L(G_{ex})$. We assume that when specifying a SemGuS problem, the user provides a semantic rule for each production in the grammar. This aspect distinguishes SemGuS from existing synthesis frameworks like SyGuS.

For example, the user might define (among others) rules like the following, for a production involving the while operator:

$$
\frac{\begin{array}{l}[\![b]\!](\Gamma, v_b) \quad v_b = \mathsf{true} \\ [\![s]\!](\Gamma, \Gamma') \quad [\![\mathsf{while}\ b\ \mathsf{do}\ s]\!](\Gamma', \Gamma_r)\end{array}}{[\![\mathsf{while}\ b\ \mathsf{do}\ s]\!](\Gamma, \Gamma_r)}\ \mathsf{WTrue} \tag{1}
$$

where Γ, Γ', and Γ_r represent state, i.e. valuations of the variables x and y. Such rules can be defined for any SMT solver by representing each judgment $[\![t]\!](\Gamma_1, \Gamma_2)$ as a logical relation and universally quantifying all variables in the derivation. Such a universally quantified statement is called a constrained Horn clause. In the rest of the paper, we often use "SMT solver" to mean a solver that can solve problems involving constrained Horn clauses.

Finally, one needs to provide a specification of the intended behavior of the program to be synthesized, which can either be a logical formula over program variables, or a finite set of input/output examples. Our algorithm for solving SemGuS problems requires that the specification be given as a set of input/output examples.

In our example, suppose that the set of input valuations is $(x, y) = [(44, 247), (6, 9), (14, 15)]$, with corresponding output values $[15, 219, 1]$. Calling this example set E_{ex}, we first show how our algorithm and SIP can synthesize a valid solution on a subset of examples $(x, y) = [(6, 9)]$. We then describe how our algorithm proves that the bitwise-xor function cannot be expressed by any program in $L(G_{ex})$ using the examples in E_{ex}.

3.2 Solving SemGuS Problems

The key idea behind our approach to SemGuS is to encode the grammar and its semantics using two sets of inference rules. More precisely, each *production* in the grammar is translated into two kinds of constraint rules: *syntax rules*, which "describe" the set of syntactically valid programs accepted by the grammar, and *semantic rules*, which "describe" the semantics of a term in the language of the grammar. Conjoined with the behavioral specification of the synthesis problem, these rules describe the set of programs in the search space whose semantics is consistent with the behavior specification. These rules are then translated into universally quantified constrained Horn clauses, which, thanks to recent advances in SMT solving, makes the problem practically feasible.

Syntax Rules. The purpose of the syntax inference rules is to describe the set of candidate terms in the (syntactic) search space. In SemGuS, we are interested in working with a broad target language G possibly containing terms outside of the language of an SMT solver, which necessiates representing terms of $L(G)$ as explicit data values of the SMT solver.

We choose to encode these (flattened) terms using lists, due to the inefficiency of algebraic datatypes in SMT solvers. Each term t accepted by G can be described by the list of productions in its leftmost derivation (i.e., a pre-order listing); our syntax inference rules describe all and only the lists that correspond to valid terms in $L(G)$.

For example, Equation 2 shows the syntax rule that corresponds to the production $Start \to \text{while } B \text{ do } S$ ❶ in G_{ex}.

$$\frac{\mathcal{L}_{out} = \mathcal{L}_B + + \overbrace{\mathcal{L}_S + + \mathcal{L}_{in}}^{\mathcal{L}_{mid}} \quad \mathsf{syn}_B(\mathcal{L}_{mid}, \mathcal{L}_{out}) \quad \mathsf{syn}_S(\mathcal{L}_{in}, \mathcal{L}_{mid})}{\mathsf{syn}_{Start}(\mathcal{L}_{in}, ❶ :: \mathcal{L}_{out})}\ syntax_{Start \to \text{while } B \text{ do } S\ ❶} \tag{2}$$

In reading the syntax rules, the intuition to keep in mind is that a pre-order listing can be produced by performing a *right-to-left post-order traversal* of t, *pre-pending* each item to the list. In this construction, a judgment $\mathsf{syn}_N(\mathcal{L}_{in}, \mathcal{L}_{out})$ for nonterminal N holds for two lists $\mathcal{L}_{in}$ and $\mathcal{L}_{out}$ if and only if $\mathcal{L}_{out} = \mathcal{L}_N + + \mathcal{L}_{in}$, and $\mathcal{L}_N$ is the pre-order listing of rules that generate a legal term from nonterminal N. An operational reading of the rules in Equation 2 is that the list is generated by recursive calls—in right-to-left order—to the right-hand-side nonterminals, followed by pre-pending the number of the production to the head of $\mathcal{L}_{out}$. The sublist $\mathcal{L}_{in}$ in $\mathcal{L}_N + + \mathcal{L}_{in}$ records the pre-order contribution from the right context of the term in which list $\mathcal{L}_{in}$ is being produced. These rules are produced for all productions in the grammar; together, they specify the possible space of programs.

Semantic Rules. Once a representation $\mathcal{L}$ of a program is obtained from the syntax rules, the semantic rules represent the semantics of executing the term $t_{\mathcal{L}}$ corresponding to $\mathcal{L}$. Because we only have the list representation $\mathcal{L}$ (and not the actual term), we will have to "unroll" the derivation tree of the term—using $\mathcal{L}$—and execute it according to its semantics on-the-fly. Our judgments are therefore of the form $\mathsf{sem}_S(\langle \Gamma, \mathcal{L} \rangle, \langle \Gamma_1, \mathcal{L}_1 \rangle)$, taking both a state-valuation Γ and a term representation $\mathcal{L}$, and will be such that $\mathcal{L} = \mathcal{L}_S + + \mathcal{L}_1$, and $[\![t_{\mathcal{L}'}]\!](\Gamma, \Gamma_1)$.

For example, the following is the semantic rule for the production $Start \to \text{while } B \text{ do } S$ ❶ using the semantics given in Equation 1.

$$\frac{\mathsf{sem}_B(\langle \Gamma, \mathcal{L} \rangle, \langle b, \mathcal{L}_1 \rangle) \quad b = true \quad \mathsf{sem}_S(\langle \Gamma, \mathcal{L}_1 \rangle, \langle \Gamma', \mathcal{L}_2 \rangle) \quad \mathsf{sem}_{Start}(\langle \Gamma', ❶ :: \mathcal{L} \rangle, \langle \Gamma_r, \mathcal{L}_3 \rangle)}{\mathsf{sem}_{Start}(\langle \Gamma, ❶ :: \mathcal{L} \rangle, \langle \Gamma_r, \mathcal{L}_3 \rangle)}\ sem^{\text{WTrue}}_{Start \to \text{while } B \text{ do } S\ ❶} \tag{3}$$

There are several things to note about this rule. First, it unpacks the list, which is "consumed" by the nonterminals B and S—in essence, inverting the creation of the list performed by the syntax rules. Second, $\mathcal{L}_2$ represents the portion of the program that follows the while loop, and hence plays no direct role inside the loop. Third, the rule executes the program specified by the list according to the semantics; in particular, $\mathsf{sem}_{Start}(\langle \Gamma', ❶ :: \mathcal{L} \rangle, \langle \Gamma_r, \mathcal{L}_3 \rangle)$ represents the next iteration of the loop, and again uses the list ❶ :: $\mathcal{L}$ (i.e.,

the representation of loop-production ❶, together with the representation of the loop-body $\mathcal{L}$).

Specification Query. The syntax rules define the set of programs generated by the grammar, and the semantic rules specify their semantics. The final step is to ask whether any of the programs are consistent with the set of input-output examples—i.e., *whether there exists a list*—a representation of a program—that, when interpreted according to the semantic rules, satisfies the specification. This question is posed via the Query rule below, which checks the existence of a list $\mathcal{L}$ that satisfies the syntax rules and the semantic rules instantiated with input/output pairs I_i, O_i.

$$\frac{\mathsf{syn}_{Start}(nil, \mathcal{L}) \quad \mathsf{sem}_{Start}(\langle I_1, \mathcal{L}\rangle, \langle O_1, nil\rangle) \quad \cdots \mathsf{sem}_{Start}(\langle I_n, \mathcal{L}\rangle, \langle O_n, nil\rangle)}{Realizable}\ \text{Query}$$

Synthesizing Programs. To see how a valid program is synthesized based on our construction, take our problem of synthesizing the bitwise-xor, specified using the input/output pair $(x, y) = [(6, 9)]$. In this case, the Horn-clause solver is responsible for finding a list $\mathcal{L}$ that satisfies the conjunction of the literals:

$$\mathsf{syn}_{Start}(nil, \mathcal{L}),\ \mathsf{sem}_{Start}(\langle(6, 9), \mathcal{L}\rangle, \langle 15, nil\rangle)$$

For the given input/output pair, bitwise-xor is indistinguishable from bitwise-or, making the problem realizable. The list [❼, ❶, ❸, ❹, ❷, ❻, ❹, ❸] satisfies both the literals in this case, which is the pre-order listing of the term while $x < y$ do $x := x \mid\mid y$ in $L(G_{ex})$—and our tool (which is based on Spacer [9] and Z3) succeeds in finding the solution.

Proving Unrealizability. To see how a SemGuS problem is proved as unrealizable, recall our full example set E_{ex}, in which case the solver would have to find some $\mathcal{L}$ that satisfies the conjunction of the literals:

$$\mathsf{syn}_{Start}(nil, \mathcal{L}),\ \mathsf{sem}_{Start}(\langle(44, 247), \mathcal{L}\rangle, \langle 219, nil\rangle),\ \mathsf{sem}_{Start}(\langle(6, 9), \mathcal{L}\rangle, \langle 15, nil\rangle),\ \mathsf{sem}_{Start}(\langle(14, 15), \mathcal{L}\rangle, \langle 1, nil\rangle)$$

Put another way, if the solver can establish that Query is *unsatisfiable*—i.e. there exists no $\mathcal{L}$ that satisfies all four literals at once—then the problem is *unrealizable.*

One thing to note about our algorithm is that it provides no additional machinery to reason about loops. Instead, as mentioned in §1, we rely on the Horn-clause solver to discover lemmas about *sets of loops*—as opposed to single loops—to prune the search space.

In our example, when proving that no program in $L(G_{ex})$ is consistent with the examples in E_{ex}, Spacer infers a lemma that states that for the third example, namely, $(14, 15) \rightarrow 1$, the third bit of whatever value is assigned to x when the loop terminates must be set to 1. This condition conflicts with the output 1 (in which the third bit is 0), 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* loops derivable from *Start*—not just some specific loop derivable from *Start*.

One might be tempted to give an operational reading of the Query rule as following the paradigm of *generate and test*: $\mathsf{syn}_{Start}(nil, \mathcal{L})$ generates list $\mathcal{L}$, which then must pass the tests $\mathsf{sem}_{Start}(\langle I_1, \mathcal{L}\rangle, \langle O_1, nil\rangle)$ ... $\mathsf{sem}_{Start}(\langle I_n, \mathcal{L}\rangle, \langle O_n, nil\rangle)$. However, the ability of Spacer to prove lemmas of the sort discussed above means that the system is not merely enumerating and testing individual lists. On the contrary, the constraint-based 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!

3.3 Instantiating the SemGuS Framework to Optimize Imperative Program Synthesis

Now that we have illustrated our core algorithm to solve SemGuS problems, we show how we can supply the framework with *vectorized semantics* to optimize solving SemGuS problems for imperative programs.

The Query rule presented above contains an inefficiency in that each semantic rule must be "re-executed" multiple times. In essence, even though the individual premises of the rules have similar structures, the rules are disjoint, which makes it hard for the solver to discover this commonality.

This problem can be circumvented by supplying the SemGuS framework with *vectorized semantics*, which are semantics that operate in parallel over *multiple* examples in the form of vectors. This idea allows us to merge premises of the form $\mathsf{sem}_{Start}(\langle I_i, \mathcal{L}\rangle, \langle O_i, nil\rangle)$ in the Query rule, into a single rule $\mathsf{sem}_{Start}(\langle \overline{I}, \mathcal{L}\rangle, \langle \overline{O}, nil\rangle)$ where $\overline{I}$ and $\overline{O}$ are vectors of variable valuations (one for each example).

Vectorized semantics are quite non-standard because of the way branch statements and loops operate. For example, consider the semantics for while loops described in Equation 1. The corresponding vectorized rule is described in Equation 4, where the rule $\text{WTrue}_{\mathsf{E}}$ must consider issues such as examples terminating on different iterations. This issue necessitates the introduction of the VProj operator, which "projects" a terminated example to an empty state $\perp$ that ignores all subsequent computations, and the VJoin operator, which restores the projected states back to their original values.

$$\frac{[\![b]\!]_{\mathsf{E}}(\overline{\Gamma}, \overline{v}_b) \quad \exists i.\ \overline{v}_b[i] = \mathsf{true} \quad [\![s]\!]_{\mathsf{E}}(\textsc{proj}(\overline{\Gamma}, \overline{v}_b), \overline{\Gamma}_1) \quad [\![\mathsf{while}\ b\ \mathsf{do}\ s]\!]_{\mathsf{E}}(\overline{\Gamma}_1, \overline{\Gamma}_2)}{[\![\mathsf{while}\ b\ \mathsf{do}\ s]\!]_{\mathsf{E}}(\overline{\Gamma}, \textsc{merge}(\textsc{proj}(\overline{\Gamma}, \neg\overline{v}_b), \textsc{proj}(\overline{\Gamma}_2, \overline{v}_b)))}\ \text{WTrue}_{\mathsf{E}} \quad (4)$$

In §4, we also give results for *fused semantics*, which assimilate the syntax rules into the semantic rules so that semantics can be executed on-the-fly while building a term in the search space. The vectorized and fused semantics give a good example of the possibilities enabled by customizing semantics in SemGuS—not only are we able to express constructs such as loops, we can also express optimization strategies for the problem!

4 Results and Contribution

Based on our algorithm described in §3, we implemented a tool SIP that is capable of synthesizing solutions to, and proving unrealizability of, SemGuS problems. A third possibility is that SIP times out. [1] We configured SIP to support a small, general imperative language that contains branches, and loops, and supports integers, Booleans, arrays, and bitvectors as datatypes. SIP is responsible for translating a SemGuS problem into CHCs, upon which we invoke Z3 (more specifically, Spacer) to solve the proof-search problem.

Although it is possible to use SIP for program synthesis, SIP was primarily designed as, and performs better, as an unrealizability prover (partly because there are already many effective program synthesizers)—thus, many of the benchmarks used in the evaluation are unrealizable as well.

We evaluated SIP on two sets of benchmarks—a set of SyGuS benchmarks (§4.1), a set of imperative synthesis benchmarks (§4.2), and also compared the vectorized and fused semantics against the individual semantics (§4.3).

4.1 Effectiveness of SIP on SyGuS Benchmarks

The first set of benchmarks consists of 132 variants of the 60 LIA (Linear Integer Arithmetic) benchmarks from the LIA SyGuS competition track. These benchmarks, which are all unrealizable, were previously generated by Hu et al. [5], and have been used as benchmarks for unrealizability in previous work [5, 6].

We compared SIP against Nay, the state-of-the-art tool for proving unrealizability of SyGuS problems. [2] SIP can prove 61/132 benchmarks unrealizable, while Nay can prove 65/132 benchmarks unrealizable. These results suggest that SIP has comparable performance with Nay—which is remarkable, because SIP is both more general than Nay and also enables program synthesis.

4.2 Effectiveness of SIP on Imperative Benchmarks

The second set consists of 288 imperative SemGuS problems defined over the various theories we support. Out of these, 35 benchmarks were created by hand from common imperative programming questions, such as the Fibonacci function or swapping variables using bitwise-xor. The remaining 253 benchmarks were derived from the 30 benchmarks described in a previous paper on synthesis of imperative programs via enumeration [11], by restricting the grammars in various ways—e.g. by only allowing certain loop conditions.

SIP succesfully terminated on 131 out of the 288 imperative benchmarks (17 were realizable and 114 were unrealizable). Out of the 131 benchmarks, 14 benchmarks—4 of which were unrealizable—were ones that contained both infinite grammars and potentially unbounded loops, indicating that SIP was successful in dealing with these challenges. SIP also terminated on 11 unrealizable benchmarks that did not contain loops, but nevertheless had an infinite grammar.

To summarize, SIP is capable to some extent of dealing with problems that contain both infinite search spaces and loops, making it the first tool that can prove unrealizability for SemGuS problems over imperative grammars.

4.3 Comparison of Various Semantics for SIP

Finally, we compared the results from the vectorized, fused, and individual semantics over all benchmarks to assess the degree of optimization enabled by custom semantics.

The individual semantics performed consistently worse compared to the vectorized and fused semantics. This situation was most pronounced in the imperative benchmarks, where the individual semantics solved only 13/288 benchmarks successfully compared to 63/288 for both the fused and separated semantics.

Comparing the vectorized and fused semantics, results were more interesting—the fused semantics had the edge on realizable benchmarks, solving a strict superset of 61 more benchmarks compared to the vectorized semantics. However, this trend seemed to be reversed for unrealizable benchmarks, where the vectorized semantics could solve 24 unrealizable benchmarks that failed using fused semantics.

To summarize this section: the various semantics one can introduce through SemGuS can have a profound effect on efficiency, and even on the kinds of synthesis problems that can be solved, opening exciting new possibilities.

5 Conclusion

We have developed a new framework called SemGuS for program synthesis, described a procedure for solving SemGuS problems, and showed how SemGuS can be instantiated to express and solve imperative synthesis problems. Through our evaluation, we also see that our algorithm for solving SemGuS problems is capable of competing with state-of-the-art unrealizability provers, as well as dealing with complex synthesis problems containing loops. Finally, we show that the customizable semantics of SemGuS allows one to equip SemGuS with interesting optimizations or evaluation strategies as well, which we expect will raise many interesting research questions.

[1] We performed experiments on a machine with a 2.3GHz processor and 160GB of RAM, with the timeout as 10 minutes.

[2] In §4.1 and §4.2, we report as SIP terminating if it terminates on any of its configurations as detailed in §4.3.

References

[1] Alur, R., Bodik, R., Juniwal, G., Martin, M. M., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013* (2013), IEEE, pp. 1–8.

[2] Alur, R., Černý, P., and Radhakrishna, A. Synthesis through unification. In *Computer Aided Verification* (Cham, 2015), D. Kroening and C. S. Păsăreanu, Eds., Springer International Publishing, pp. 163–179.

[3] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. Cvc4. In *International Conference on Computer Aided Verification* (2011), Springer, pp. 171–177.

[4] De Moura, L., and Bjørner, N. 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* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, pp. 337–340.

[5] Hu, Q., Breck, J., Cyphert, J., D'Antoni, L., and Reps, T. Proving unrealizability for syntax-guided synthesis. *arXiv preprint arXiv:1905.05800* (2019).

[6] Hu, Q., Cyphert, J., D'Antoni, L., and Reps, T. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems.

[7] Hu, Q., and D'Antoni, L. Syntax-guided synthesis with quantitative syntactic objectives. In *International Conference on Computer Aided Verification* (2018), Springer, pp. 386–403.

[8] Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering* (2010), vol. 1, IEEE, pp. 215–224.

[9] Komuravelli, A., Gurfinkel, A., and Chaki, S. Smt-based model checking for recursive programs. *Formal Methods in System Design 48*, 3 (2016), 175–205.

[10] Shi, K., Steinhardt, J., and Liang, P. Frangel: component-based synthesis with control structures. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1–29.

[11] So, S., and Oh, H. 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* (2017), pp. 364–381.

[12] Solar-Lezama, A. Program sketching. *STTT 15*, 5-6 (2013), 475–495.