Guarantees in Program Synthesis

Qinheping Hu[1], Jason Breck[1], John Cyphert[1],
Loris D'Antoni[1], Thomas Reps[1,2]
[1] University of Wisconsin-Madison, Madison, USA [2] GrammaTech, Inc., USA

1 Introduction

Program synthesis is the classic problem of automatically finding a program implementation in some search space that satisfies a given correctness specification. Traditionally, program synthesis is viewed as a deductive theorem proving problem [5, 12, 15] with specifications written in logic. More recently, studies on program synthesis problems with input-output examples [6, 8, 11] or user-specified search space [1, 13] propel program synthesis to more practical fields.

Although in general program synthesis is an exceptionally challenging problem, synthesis even in a relatively small scale can significantly impact software development in the sense that it reduces programmers' efforts to produce programs with concise intuition but daunting detail. For example, let's say we are synthesizing a program f, that inverts two given program $g_1(x) := (E$ `(bvshr` x `#x06`)) and $g_1(x) := (E$ `(bvshl` x `#x02`)), where E is an encoding map function. The specification $\varphi := \forall x. f(g_1(x), g_2(x)) = x$ is straightforward. A reliable synthesis tool may save us from tedious work of writing and debugging an implementation of f such as (`bvor` (`bvshl` $(D\ y_1)$ `#x06`) (`bvshr` $(D\ y_1)$ `#x02`)) where D is the provided decoding map function.

Unfortunately, it is usually not enough to produce *any* correct solution. For many program synthesis problems, there are multiple correct solutions in the search space, but some of them are not usable because, for example, their sizes are too large to be read. At the same time, synthesizers are unpredictable–users have no way to prefer one correct solution over others. Thus, sometimes synthesizers may produce a correct but unusable solution. For the example we mentioned above, the implementation of f returned by the state-of-the-art synthesis tool CVC4 is a lookup table with *hundreds* of `ITE` operators, and hence, is unreadable. Besides, when the search space is infinite, most of the synthesizers can only output a solution or timeout on the given synthesis problem. That is, when a solver timeouts, users know nothing about whether the solving is not finished, or the given synthesis problem is *unrealizable*–no solution in the search space satisfies the correctness specification.

Therefore, besides correctness and efficiency, one may want ask two more questions about a synthesis solver.

Question 1. Can the solver provide a *good* solution when there are multiple ones?

Question 2. Can the solver provide any information when there is no solution?

To answer the above two questions, we introduce two types of guarantees in program synthesis: *quantitative objectives* and *the ability to answer unrealizable*. The quantitative objectives extend the dimension of specifications in program synthesis–users can desire not only correct but also, for example, more efficient, more readable or more probable solutions. The ability to answer unrealizable guarantees users to get more information from synthesizers–they will return not only a solution but also explain why the solution is optimal or why the synthesis problem is unrealizable.

For each type of guarantees, we introduce a published work to see how to introduce guarantees into program synthesis and how to solve the synthesis problems with guarantees. Besides, there are still many more interesting topics and questions worth to be explored in the future. We presented some of them as future directions at the end of each section.

2 Quantitative Objectives

Quantitative objectives provide a natural way to guarantee users more preferable solutions. In this section, we first show our previous work [10] addressing **Question 1** for SyGuS problems and then show some future directions about quantitative objectives in synthesis problems.

2.1 SyGuS with Quantitative Syntactic Objectives

The goal of this work is to extend syntax guided synthesis (SyGuS) to SyGuS with quantitative syntactic objectives (QSyGuS) [10], a unifying framework for describing SyGuS problems with syntactic quantitative objectives–e.g., find the minimal solution–and present an algorithm for solving synthesis problems expressed in this framework. We focus on syntactic objectives because they are the most common ones in practical applications of program synthesis. For example, in programming by examples it is desirable to produce small pro- grams with fewer constants because these programs are more likely to generalize to examples outside of the specification [7]. QSyGuS extends SyGuS in two ways. First, in QSyGuS the search space is represented using weighted grammars, which augment context-free grammars with the ability to assign weights to programs. Second, QSyGuS allows the user to specify constraints over the weight of the solution, including optimization objectives—e.g., find the program with the fewest `ITE`-operators.

We illustrate QSyGuS problems and an algorithm to solve QSyGuS problems by a simple example. We start with a

$$
\begin{array}{llll}
\text{Start} \to & \text{Start} + \text{Start}/\mathbf{0} & \text{BExpr} \to & \text{Start} > \text{Start}/\mathbf{0} \\
& \mid \texttt{ITE}(\text{BExpr}, \text{Start}, \text{Start})/\mathbf{1} & & \mid \neg\text{BExpr}/\mathbf{0} \\
& \mid x/\mathbf{0} \mid\ y/\mathbf{0} \mid\ 0/\mathbf{0} \mid\ 1/\mathbf{0} & & \mid \text{BExpr} \wedge \text{BExpr}/\mathbf{0}
\end{array}
$$

Figure 1. Weighted grammar that assigns weight $w \in \text{Nat}$ to a program where w is the number of ITE-operators.

Syntax-Guided Synthesis (SyGuS) problem in which no quantitative objective is provided. Recall that the goal of a SyGuS problem is to synthesize a function f of a given type that is accepted by a context-free grammar G, and such that $\forall x.\phi(f, x)$ holds (for a given Boolean constraint ϕ).

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

$$
\begin{array}{l}
\text{Start} \to \text{Start} + \text{Start} \mid \texttt{ITE}(\text{BExpr}, \text{Start}, \text{Start}) \mid x \mid y \mid 0 \mid 1 \\
\text{BExpr} \to \text{Start} > \text{Start} \mid \neg\text{BExpr} \mid \text{BExpr} \wedge \text{BExpr}
\end{array}
$$

The semantic constraint is given by the following formula.

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

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

$$
\begin{array}{lcl}
max_1(x, y) & = & \texttt{ITE}(x > y, x, y) \\
max_2(x, y) & = & \texttt{ITE}(x > y, x, \texttt{ITE}(y > x, y, x))
\end{array}
$$

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

Adding weights. In our formalism, QSyGuS, we augment context-free grammars to assign weights to programs in the search space. Concretely, we adopt weighted grammars [4], a well-studied formalism with many desirable properties. In a weighted grammar, each production is assigned a weight. For example, the weighted grammar shown in Figure 1 extends the one from the previous SyGuS example to assign to each program p a weight w where w is the number of ITE-operators in p. In this case, the weight is an integer and the weight of a grammar derivation is the sum of all the weights of the productions involved in the derivation. In the figure, we write $/w$ to assign weight w to a production. The functions max_1 and max_2 have weights 1 and 2 respectively.

Adding and solving quantitative objectives. Once we have a way to assign weights to programs, QSyGuS allows the user to specify quantitative objectives over the weights of the productions—e.g., only allow solutions with fewer than 2 ITE-operators. In our example, we could require the solution to be minimal with respect to the number of ITE-operators, i.e., minimize the first component of the paired weight. With these constraints only max_1 would be considered optimal solutions because there exists no solution with 0 ITE-operators.

Our tool QuaSi can automatically discover solutions in both these cases. Let's consider the last minimization objective. In this case, QuaSi first uses existing SyGuS solvers to synthesize an initial solution using the non-weighted version of the grammar. Let's say that the returned solution is, for example, max_2 of weight 2. QuaSi uses this solution to build a new SyGuS instance that only accepts programs with at most one ITE-operators. The idea behind our construction is to introduce new nonterminals in the grammar to keep track of the weight of the trees that can be produced from those nonterminals. For example, the following grammar only produce terms with no more than one ITE-operators

$$
\begin{array}{l}
\text{Z} \to \text{Start\_1} \mid \text{Start\_0} \qquad \text{Start\_0} \to x \mid y \mid 0 \mid 1 \\
\text{Start\_1} \to \texttt{ITE}(\text{BExpr\_0}, \text{Start\_0}, \text{Start\_0}) \mid x \mid y \mid 0 \mid 1 \\
\text{BExpr\_0} \to \text{Start\_0} > \text{Start\_0} \mid \neg\text{BExpr\_0} \mid \text{BExpr\_0} \wedge \text{BExpr\_0}
\end{array}
$$

Solving this SyGuS problem can, for example, result in the program max_1 of weight 1, which will require our solver to build yet another SyGuS instance. This approach is repeated and if it terminates, an optimal program is found.

2.2 Future Directions

Besides syntactic quantitative objectives, there are other types of quantitative objectives: quantitative semantic objectives and resource bound objectives.

Quantitative Semantic objectives. When we know a program synthesis problem is unrealizable, a question naturally arises: *how much of the correct specification we can satisfy on a best effort?* With such objectives, we can find a solution to satisfy, for example as many as input-output examples as possible. Such quantitative objectives can also be used to train a neural network. Another example application is approximation synthesis [2], which allow us to find an approximate solution instead of finding a solution to fulfill the correct specification with a longer solving time.

A natural way to formalize synthesis problem with semantic quantitative objectives is that, instead of only one correct specification, we allow a program synthesis problem to contain a set of weight-specification pairs (φ_i, w_i). And we define the semantic weight of a solution e as $\sum_i [\varphi_i(e) = true] w_i$. Similar to syntactic quantitative objectives we presented in QSyGuS, a syntactic objective could be a range of allowed weight of solutions or a requirement of optimizing the solutions' weight.

Resource-bounded program synthesis. Another kind of complicated quantitative objectives we interested in is bounding the resource used by solutions. Resource usage including solving time, memory usage and domain-specific resource metrics has, of course, large impact on the quality of synthesis solutions. This problem is challenging because both synthesis and resource analysis are undecidable in theory and expensive in practice.

3 Ability to Answer Unrealizable.

Determining whether there is a solution for the given program synthesis problem is challenging because the search space in program synthesis is usually infinite and exhaustive search will fail in such cases. In this section, we will first introduce our previews work [9] to answer **Question 2** and then show some future directions on the topic of proving unrealizability of synthesis problems.

3.1 From Unrealizability to Unreachability

In §2.1, we showed an optimization synthesis algorithm which iteratively construct SyGuS sub-problems that only accept solutions with cost than the current solution's cost. The soundness of this optimization synthesis depends on whether the synthesizers have the ability of answering unrealizable–a current solution is optimal if the synthesis problem of finding smaller cost is unrealizable. However, most of the state-of-the-art SyGuS solver can not prove the unrealizability of unrealizable SyGuS problems with infinite search space.

In this work, we have introduced a novel SyGuS technique with the ability to answer unrealizable. Our technique bases on the Counterexample-Guided Inductive Synthesis (CEGIS) framework and, in each CEGIS iteration, reduces a SyGuS sub-problem to a reachability problem that can be proved to be unrealizable by off-the-shelf program verification solvers.

We illustrate by examples our framework for establishing the unrealizability of a SyGuS problem.

Again consider the SyGuS problem to synthesize a function f that computes the maximum of two variables x and y, denoted by (ψ, G). The grammar G we provided only produces terms without ITE-operators.

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

In fact, this SyGuS problem is *unrealizable*—i.e., it does not admit a solution, because no expression generated by G meets the specification.[1].

Now we show that how the unrealizability of (ψ, G) can be proven using input examples: $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$.

Our method can be seen as a variant of Counter-Example-Guided Inductive Synthesis (CEGIS), in which the goal is to create a program P in which a certain assertion always holds. Until such a program is created, each round of the algorithm returns a counter-example, from which we extract an additional input example for the original SyGuS problem. On the i^{th} round, the current set of input examples E_i is used, together with the grammar—in this case G—and the specification of the desired behavior—ψ), to create a candidate program $P[G, E_i]$. The program $P[G, E_i]$ contains an assertion, and a standard program analyzer is used to check whether the assertion always holds. When the assertion holds, all possible paths falsify the specification, and hence the original SyGuS problem is unrealizable. Figure 2 shows an example of $P[G, E_1]$ with example $(0, 1)$, in which we use an external function nd() for nondeterministically shooting production rules and each rule is encoded to be a sequence

```
 int I_0;
 void Start(int x_0,int y_0){
  if(nd()){  // Encodes ``Start+ Start)''
    Start(x_0,y_0); int tempL_0=I_0;
    Start(x_0,y_0); int tempR_0=I_0;
    I_0 = tempL_0+tempR_0;}
  else if(nd()) I_0 = x_0;  // Encodes `` x''
  else if(nd()) I_0 = y_0;  // Encodes `` y''
  else if(nd()) I_0 = 1;    // Encodes ``1''
  else          I_0 = 0;    // Encodes `` 0''
}
bool spec(int x, int y, int f){
  return (f>=x && f>=y && (f==x || f==y))}
void main(){
  int x_0 = 0; int y_0 = 1;   // Example (0,1)
  Start(x_0,y_0);
  assert(!spec(x_0,y_0,I_0));
}
```

Figure 2. Program $P[G, E_1]$ created during the course of proving the unrealizability of (ψ, G) using the set of input examples $E_1 = \{(0, 1)\}$.

of statements that evaluates and returns the value of an expression trees as a global variable I_0. The multi-examples cases are similar.

3.2 Future Direction

Decidable fragment of SyGuS. We described an approach to probing unrealizability of SyGuS problems in §3.1 using a reduction from SyGuS problems to reachability problems. However, the evaluation result shows that our tool nope times out on about half of the benchmarks. To address those time-outing benchmarks, a future direction is to explore the decidable fragment of SyGuS problems. Decidable fragment of SyGuS has been studied in previous study [3] but our benchmarks for nope is not included in the studied domain in their work.

A preliminary idea is to first summary the grammar in the decidable fragment and then prove that the summary is disjoint from the specification. For example, we may summary the decision-tree like LIA grammar as a semi-linear set and then construct an SMT query to check if the specification is disjoint from the semi-linear set.

Synthesizing imperative programs. The technique used in §3.1 is not restricted in SyGuS framework. This proposed work aims to generalize the technique used in §3.1 to imperative programs. The synthesis of imperative programs has been studied using a deductive approach [14]. Our goal is to build an synthesizer with the ability to answer unreliable and to incorporate syntactic quantitative objectives.

The main challenge when using the encoding presented in §3.1 to encode imperative synthesis problems is the assignment statement. For SyGuS problems, we only care about the evaluated values of terms while, for imperative program program synthesis, we need to also consider program states, e.g., when we encode a rule $r : S \rightarrow \text{Concate}(\text{Assign}(x, E), S)$, the evaluation of E will influence the evaluation of S.

[1]Grammar G only generates terms equivalent to some linear function; however, the maximum function cannot be described by a linear function.

References

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

[2] James Bornholt, Emina Torlak, Luis Ceze, and Dan Grossman. 2015. Approximate Program Synthesis. (2015). https://homes.cs.washington.edu/~bornholt/papers/appsyn-wax15.pdf

[3] Benjamin Caulfield, Markus N. Rabe, Sanjit A. Seshia, and Stavros Tripakis. 2015. What's Decidable about Syntax-Guided Synthesis? *CoRR* abs/1510.08393 (2015). http://arxiv.org/abs/1510.08393

[4] Manfred Droste, Werner Kuich, and Heiko Vogler. 2009. *Handbook of Weighted Automata* (1st ed.). Springer Publishing Company, Incorporated.

[5] Cordell Green. 1981. Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*. Elsevier, 202–222.

[6] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–330.

[7] Sumit Gulwani. 2016. Programming by Examples: Applications, Algorithms, and Ambiguity Resolution. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*. 9–14. https://doi.org/10.1007/978-3-319-40229-1_2

[8] William R Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 317–328.

[9] Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. 2019. Proving Unrealizability for Syntax-Guided Synthesis. In To Appear *in Computer Aided Verification Conference, CAV*.

[10] Qinheping Hu and Loris D'Antoni. 2018. Syntax-Guided Synthesis with Quantitative Syntactic Objectives. In *International Conference on Computer Aided Verification*. Springer, 386–403.

[11] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 215–224.

[12] Zohar Manna and Richard J Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM* 14, 3 (1971), 151–165.

[13] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 281–294.

[14] Jamie Stark and Andrew Ireland. 1999. Towards automatic imperative program synthesis through proof planning. In *14th IEEE International Conference on Automated Software Engineering*. IEEE, 44–51.

[15] Richard J Waldinger and Richard CT Lee. 1969. PROW: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., 241–252.