Program Synthesis Project

Vivian Ding (vyd2), Alex Drake (amd459)

October 5, 2023

Contents

1 Prompt

Program synthesis is an emerging area, full of exciting problems and new applications. At the end of this course you will hand in a final project relating to program synthesis. To allow you to fully exercise your creativity, we have left this final project somewhat open-ended.

Your final project should include:

- A short one page proposal (due October 6)
- A written progress report (2-5 pages, due November 6)
- A written report in the style of a computer science conference paper (5-10 pages, due last day of class, December 4).
- A short oral presentation to the class, with slides (presented last 2 days of class)

Note that you do not have to have a "positive" result in order to receive full credit for the final project! It's fine to explore a new idea and discover that it doesn't work as you expected. However, if you get a "negative" result, you should still explain what you did and seek to understand why you got this "negative" result.

2 Motivation

One limiting factor in program synthesis is the domain-specific knowledge required to write a synthesizer for a particular target language (hereafter referred to as "target"). Specifications for the target can often be used to speed up search through the program space. For instance, deduction rules about the types and sizes of inputs and outputs to operators are used to prune the program space in top-down search. However, these specifications are not always easy to write, especially for users inexperienced with formal languages, and they must be written separately for every desired target for synthesis. As a result, a novice user is mostly restricted to synthesizers pre-built for a particular target.

We propose a synthesizer which generates (approximate) specifications to be used as input to another program synthesizer. This system ought to be general enough to be applicable to any library without a deep understanding of formal languages or algebraic specification, but produce enough additional information to produce significant speedup in search.

The lofty goal of this is to reduce the work of using a program synthesizer for a previously unsupported target to near zero. Instead of requiring the user to specify the target in detail, which requires some level of expertise, a synthesizer could take as input some existing, easy to find information such as documentation and use it to inform synthesis. Essentially, it could "learn" to generate programs in new languages or using new libraries with little assistance[1].

3 Problem statement

To accomplish this goal, we propose a two-step system. **Step 1** involves the synthesis of specifications on the target language. **Step 2** refers to the synthesis of programs using the output of Step 1. The workflow can be broken down as follows.

a. Starting from some **initial specification** of the target, such as the function implementation, natural language documentation, or example input/output pairs,

b. Generate information useful for synthesis, the **synthesized properties** (step 1),

c. Which can be fed to a synthesizer to generate programs in the **target** (step 2).

One might imagine various approaches to implementation, such as using NLP techniques to analyze natural language documentation. However, for step 1 of this project, we focus on an algorithm which uses an SMT solver to answer the following problem: **Given black-box access to a function f and expressive positive examples of input/output to f, attempt to find a precise conjunctive formula—expressed in a unifying language of properties $\mathcal{U}$—that holds under all calls to f.**

For instance, here are some examples of properties we seek to synthesize, expressed in the form *function invocation* $\implies$ *property*. Notice that, although they look similar, the first

[1] "Learn" is in quotations since we do not actually take an ML approach in this project, but that is a promising option for future work.

three examples use a list-like library as the target, while the latter two pertain to strings.

$$\begin{aligned} o = \texttt{append}(l, x) &\implies \texttt{len}(o) = \texttt{len}(l) + 1 \\ o = \texttt{reverse}(l) &\implies \texttt{len}(o) = \texttt{len}(l) \\ o = \texttt{replace}(l, i, x) &\implies \texttt{len}(o) = \texttt{len}(l) \end{aligned}$$

$$\begin{aligned} o = \texttt{substring}(s, i_1, i_2) &\implies \texttt{len}(o) \leq \texttt{len}(s) \\ o = \texttt{concat}(s_1, s_2) &\implies \texttt{len}(o) = \texttt{len}(s_1) + \texttt{len}(s_2) \end{aligned}$$

Notice that these properties are the most precise properties. For instance, it would be less expressive to have $o = \texttt{concat}(s_1, s_2) \implies \texttt{len}(o) \geq \texttt{len}(s_1) \wedge \texttt{len}(o) \geq \texttt{len}(s_2)$, knowing that `len` is always non-negative.

In addition to synthesizing these properties, we aim to make the interface to the user as simple as possible - all inputs must be information commonly found in documentation, or otherwise easy to write by a programmer unexperienced in formal programming languages.

4 Implementation

4.1 Step 1

The implementation of step 1 will be heavily inspired by the Spyro project[2]. However, we make some substantial changes.

4.1.1 Inputs

Spyro takes as input:

1. Query $\Phi = \bigwedge_{i=1}^{n} o^i = f^i(x_1^i, \ldots, x_m^i)$, where o^i is an output variable, each x_j^i is an input variable, and f^i is a function symbol,
2. Grammar of a DSL $\mathcal{L}$ in which the synthesizer is to express properties,
3. Semantics of function symbols in the query Φ ,
4. Semantics of function symbols in $\mathcal{L}$ (e.g. `len`).

It then generates a most-precise $\mathcal{L}$-conjunction implied by Φ (by making calls to an SMT solver and building up a list of positive and negative examples to guide synthesis).
Our tool will take as input:

1. Function $f(x_1, \ldots, x_n)$ and its type,
2. List of example inputs and outputs $(i_1^j, \ldots, i_n^j, o^j)$
3. Implementations of basic primitives used in $\mathcal{U}$, such as `len`.
4. List of functions (and their types) which can be used to mutate inputs to f.

The correspondence between Spyro and our tool is as follows.

1. The function f is used to produce a query $\Phi = (o = f(x_1, \ldots, x_n))$. This query is much simpler than what can be used in Spyro, but in §4.2 we describe why it is sufficient for basic synthesis.

[2]Synthesizing Specifications, Park et al. 2023, https://arxiv.org/pdf/2301.11117.pdf

2. Since our goal is to generate specifications for arbitrary target languages, we use a built-in unifying specification language $\mathcal{U}$ in place of $\mathcal{L}$. For each invocation of our tool, $\mathcal{U}$ is extended to include input and output variables as terminals.
3. In Spyro, the formal specifications of function symbols in Φ impose a set of valid models over the variables in Φ, denoted $[\![\varphi_\Phi]\!]$. The list of example inputs and outputs to our tool will be used as an underapproximation of $[\![\varphi_\Phi]\!]$. These examples—which can be easily pulled from documentation or simply running the function on a few chosen inputs—replace the need for users to supply formal semantics of function symbols.
4. The implementations of basic primitives used in $\mathcal{U}$, such as `len`, will be used in replacement of a formal semantics of function symbols in the DSL of properties. While Spyro passes the semantics of these functions to an SMT solver, we instead inflate the input to the SMT solver by calling the user-provided implementation on all variables in the query Φ. We can then pass the result of each computation as an additional formula to the solver.

Notice that our tool requires the user to supply dramatically less complicated input. While Spyro requires that the user supply the concrete semantics of all function symbols in the query, as well as the language of properties, our tool requires no formal semantics to be supplied. Writing the concrete semantics of any nontrivial function is typically at least mildly challenging for an experienced user, and likely near impossible for a novice programmer. The example input-outputs can be taken from documentation, which often (if good) includes examples for both common and edge cases, reducing load on the user.

Additionally, our tool does not place the responsibility of choosing a language of properties and interesting query upon the user. While Spyro certainly allows for more customization, and is therefore well-suited to the experienced user, our tool is more conducive to use by programmers not well-aquainted with formal programming language semantics.

The grammar for $\mathcal{U}$ is as follows[3]. Let x be any input or output variable, n any number (we restrict to integers for the scope of this project).

$$\begin{aligned} p &\to e\ cmp\ e \mid p \vee p \mid p \wedge p \mid \neg p \\ cmp &\to = \mid \leq \mid < \mid \geq \mid > \\ e &\to n \mid x \mid \texttt{len}(x) \mid e\ op\ e \\ op &\to\ +\ \mid\ -\ \mid\ \times\ \mid \div \end{aligned}$$

While `len` is not expressive enough to represent multi-dimensional arrays or other useful properties of data, we use this language as a simple proof-of-concept. We suspect that similar work can be done to dramatically broaden the scope of this tool by using a slightly more sophisticated notion of size.

4.1.2 Algorithm

We will use roughly the same algorithm as Spyro.

[3]We may need to omit division from the implementation, as it is not yet clear if it will work.

4.1.3 (Un)soundness

A property φ is sound with respect to Φ when $[\![\varphi_\Phi]\!] \subseteq [\![\varphi]\!]$, where $[\![\varphi]\!]$ is the set of models of the synthesized property φ. Spyro guarantees soundness with respect to Φ; however, our system does not.

This is because our tool underapproximates $[\![\varphi_\Phi]\!]$ using the set of example inputs and outputs for f. Let's denote this set of examples E. The algorithm used in Spyro guarantees soundness with respect to the input, which describes the set of valid models. Since Spyro requires the user to supply concrete semantics of all functions, the algorithm receives the "ground truth" set of valid models and is therefore entirely sound. On the other hand, with only E as input, the algorithm guarantees that for all generated properties φ, it holds that $E \subseteq [\![\varphi]\!]$. This clearly does not imply soundness with respect to Φ, since $E \subseteq [\![\varphi_\Phi]\!]$[4]. Without requiring the user to supply concrete semantics for all functions in Φ and $\mathcal{U}$, $[\![\varphi_\Phi]\!]$ can only be approximated. As a result, we sacrifice soundness for the sake of accessibility.

However, we claim (and hope) that this will be fine. While the search in step 2 would not satisfy completeness if unsound rules were used to prune the search space, we can instead use the synthesized properties to *guide* search. Further description of this is in §4.2.

We also claim (and hope) that our approximation of $[\![\varphi_\Phi]\!]$ is "decent". Since examples provided in function documentation often capture both common and edge cases, we predict that they are sufficiently expressive to approximate $[\![\varphi_\Phi]\!]$.

We can further reduce the gap between E and $[\![\varphi_\Phi]\!]$ by simply generating more examples. A proposed procedure for this is in §4.1.5.

Finally, we expect that omitting arbitrary constants from $\mathcal{U}$ or downweighting "uncommon" constants in the algorithm will reduce the likelihood of producing nonsensical/unsound rules such as false restrictions on length, as mentioned earlier[5]. If we omit arbitrary constants, we may require that values such as `int` be implemented as length 1.

4.1.4 Precision

Spyro uses $\overline{[\![\varphi_\Phi]\!]}$ to compute a set of negative examples ψ from which the function CheckPrecision may draw counterexamples. Since we do not have this, there are three options.

1. For each provided example $(i_1^j, \ldots, i_n^j, o^j)$, we add the set $\{(i_1^j, \ldots, i_n^j, o) \mid o \neq o^j\}$ to ψ. Intuitively, then CheckPrecision may only draw counterexamples for the candidate property from pairs that are certainly negative examples[6]. It is the easiest option to implement.
2. We can use the same negative examples as in the previous item, but extended to include more examples generated by the procedure described in §4.1.5.
3. We can extend the set of negative examples to include all input/output pairs not in E. This may gain us more precision, but at the cost of overfitting to the examples.

We believe it will take some fiddling to see which option works best in practice.

[4] This should also make sense intuitively, as we may "overfit" to the examples. For instance, let f be list `reverse`. If all user-provided examples contain lists of size at most 2, we may erroneously introduce the restriction that input to f may not be of size 6172. This property is sound with respect to the examples, but not with respect to the underlying function itself.

[5] See footnote 4.

[6] This does not buy us soundness.

4.1.5 Extending the example set

Since the set of examples E is a subset of the ground-truth semantics $[\![\varphi_\Phi]\!]$, we can improve the approximation by synthesizing more examples.

It would be difficult to do this without additional support from the user. We propose that the user specify a list of functions and their type and bound requirements (i.e. whether arguments such as indices should be within some range) that may be used to mutate inputs in the user-provided examples. This need not be actually implemented by the user; simply passing in an existing function, for example `List.append()` or `String.substring()`, would suffice. We claim this is still not too much additional work for the user; after all, whether a function will throw an index out-of-bounds exception can be reasonably expected to be in the documentation.

Given these functions, we can randomly mutate inputs from the provided examples to produce new examples of the correct type[7]. For parametric types (e.g. `List<List<Int>>`), we may need to consider applying these functions recursively, but this is still doable.

Since this step places extra burden upon the user and is nontrivial to implement, we will leave it out of our first draft.

4.1.6 Evaluation

We will evaluate the quality of the synthesized properties by inspection. We may try giving ChatGPT the same input and comparing it with our tool, mostly for fun. If we were to do this, we would test on "unusual" functions, such as interleaving 3 lists or swapping the first and last elements of the list, in order to avoid using an example it is already trained on.

4.2 Step 2

This section describes how properties synthesized in step 1 can be used to guide search. This approach is inspired by BUSTLE[8] and Write, Execute, Assess[9], which reweight partial programs (affecting the order in which they are explored) after assessing their likelihood.

We will use a modified top-down search, where the order of exploration of the program space is determined by properties in step 1. If a synthesized property implies that a function cannot be applied at a particular point in program search, rather than pruning the branch entirely, the step 2 search algorithm can simply decrease the weight of the function, exploring the space of programs using it later. This guarantees that our search is still complete despite being guided by unsound deduction rules.

In a basic approach to top-down synthesis, we use deduction rules to determine whether a subtree of the program space, rooted at a particular function invocation, should be explored. As a result, while our queries to the property synthesizer are much simpler than in Spyro, they are sufficient for this approach to synthesis; we only need rules for one invocation to an operator at a time anyways.

4.2.1 Evaluation

We will evaluate the correctness of the synthesized programs through automatic testing. Quality of the programs will be determined by inspection. If we have time, we will compare

[7] It is unclear how to mutate first-class functions, so let's ignore that for now.
[8] Odena et al. https://arxiv.org/pdf/2007.14381.pdf
[9] Ellis et al. https://arxiv.org/pdf/1906.04604.pdf

correctness, speed, and quality of generated programs to other implementations such as top-down with no deduction rules, our step 2 implementation with reweighting but using manually written deduction rules (this one will be interesting), top-down with pruning and manually written deduction rules (almost guaranteed to be faster than our implementation), top-down with pruning and our synthesized properties (likely to fail to find programs), and potentially others.

5 Plan

The project will consist of an implementation of our tool, a description of the implementation in the report, further discussion on soundness and preciseness of step 1, and evaluation of the implementation.

We will evaluate both steps of the system using various libraries, such as for List, String, Queue, Stack, or Heap/PriorityQueue interfaces, in order to show versatility even with only two interesting terminals in $\mathcal{U}$ (x and $len(x)$). Many modern program synthesizers are completely limited to targets as small as individual string libraries, so while small, this is general enough to be slightly interesting. We do not expect performance to be better than existing synthesizers, but it will be for the benefit of a much more general tool, capable of something no other synthesizers (to our knowledge) are capable of.

Some milestones we have set for ourselves are:

1. Implement step 1 without arbitrary constants. This means:
   (a) Implement $\mathcal{U}$ and the construction of queries.
   (b) Implement the Spyro algorithm.
   (c) Link to an SMT solver. This should complete the first draft.
   (d) If the output is horrible, try extending the example set.
2. Evaluate step 1 (by manual inspection).
3. If the output is horrible due to lack of arbitrary constants, implement arbitrary constants in step 1. Otherwise, leave this until the end as a possible extension.
4. Implement step 2, or, if there's not enough time,
5. Compare our implementation of step 1 with ChatGPT.
6. Evaluate step 2.

This seems like an ambitious project, and because step 1 alone is decently interesting, we won't be too unhappy with only completing up to milestone 2.

Final report page breakdown:

- 1-2 pages - Introduction: motivation, problem statement
- 5 pages - Step 1: definitions and inputs (1-2 pages), description of algorithm (1 page), soundness and precision discussions (1 page), extending the example set (0.5-1 page), evaluation (1 page)
- 2 pages - Step 2: implementation (1 page), evaluation (1 page)
- 1 page - Conclusion

We're thinking of calling this system Synthesis for Synthesis (S4S).