At the End of Synthesis: Narrowing Program Candidates

David Shriver, Sebastian Elbaum
Computer Science and Engineering Department
University of Nebraska
Lincoln, NE, USA
{dshriver, elbaum}@cse.unl.edu

Kathryn T. Stolee
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
ktstolee@ncsu.edu

***Abstract*—Program synthesis is succeeding in supporting the generation of programs within increasingly complex domains. The use of weaker specifications, such as those consisting of input/output examples or test cases, has helped to fuel the success of program synthesis by lowering adoption barriers. Yet, employing weaker specifications has the side effect of generating a potentially large number of candidate programs. This was not a problem for simpler and smaller program domains, but it is becoming evident that differentiating among many synthesized programs is a challenge that needs addressing. We sketch an approach to mitigate this challenge, requiring less effort from the user while automatically identifying inputs that can differentiate clusters of synthesized programs. The approach has the potential to more cost-effectively narrow the space of candidate programs in a range of synthesis applications.**

***Keywords*-program synthesis, pruning, input generation**

I. Introduction

Inductive program synthesis techniques semi-automatically derive a set of programs that satisfy a specification. There are a myriad of synthesis approaches ranging from analytical to logical [1]. There are also many forms of specifications ranging from logic formula to input/output examples. These specifications are known to be incomplete in that they only describe part of the desired program behavior, leading to solutions that are only sound in relation to the incomplete specifications, but unsound in general. Yet, inductive synthesis has achieved remarkable success in the last few years for domains including spreadsheet transformations [2], string manipulation [3], program repair [4]–[6], number formatting [7], parser synthesis [8], bit-streaming programs [9], and source code snippet synthesis [10], [11].

These recent successes can be attributed to at least three factors. First, technical advances in solvers and analysis algorithms, and additional computation power at the reach of many users, have made synthesis faster and more powerful. Second, the domains on which synthesis is being performed have been strategically chosen; programs in those domains can be encoded with relatively simple grammars. Third, the application of synthesis in the targeted domains requires simple specifications, either from the end-user or from test cases, and often in the form of input and output example pairs.

We expect for the first factor to continue its current evolution as computational power and solver-analysis speed continue to increase. For the second factor, although many of the "lower hanging fruit" domains have been identified, the additional power and speed enables targeting richer domains. We are not as optimistic, however, about the third factor.

As synthesizers create larger and more complex code, the space of potential solutions for a weak specification will tend to increase. This intensifies the burden on the client of the synthesizer, whether it's a user selecting a program [2], [3], a program repair engine creating a patch [4], [5], or a code search tool retrieving ranked matches [10], [11]. Independent of the client, lowering this burden implies having more complete and precise user specifications (i.e., more input/output examples). We conjecture, however, that requesting users for more useful examples to refine the space of synthesized programs is rapidly becoming the next bottleneck as synthesis scales to more complex problems.

In this work we focus on one dimension of this challenge, the one occurring at the end of the synthesis process when the specifications provided are deemed enough to generate many candidates that need to be narrowed down to a winner.

Our first insight to address this problem is: *narrowing the set of synthesized program candidates is (in part) an input generation problem.* That is, we want to flip the problem from one of synthesizing programs, to one of synthesizing inputs, which is something that automated input generation techniques can do really well. Now, we do not want just any inputs. We need inputs that can differentiate among candidate programs and help us narrow them quickly, which, as we show, requires some adaptation. At this point we can leverage our second insight: *in most cases it is easier for users to derive the output for a given input, than to find a differentiating input and compute its corresponding output.* As a result, we can reduce the load on users by requesting from them just the corresponding expected output to an automatically generated differentiating input.

Next, we will illustrate the challenge of refining input/output example specifications, outline an approach that builds on our key insights, and show a preliminary implementation within the data wrangling domain.

II. Motivation

To explain the extent of the problem, we employ a use case from the Zipfian Academy, a group that teaches how to analyze large data sets[1]. In this use case, a fictional scientist wants to

[1]http://nbviewer.ipython.org/github/Jay-Oh-eN/happy-healthy-hungry/blob/master/h3.ipynb

TABLE I: Inputs and Output Example

(a) Business Information Table (input 1)

bus_id	name	address	city	state	latitude	longitude	phone
16441	“HAWAIIAN DRIVE”	“2600 SAN BRUNO AVE”	“SFO”	“CA”	NA	-122.404101	NA
61073	“FAT ANGEL”	“1740 O’ FARRELL ST ”	“SFO”	“CA”	0.0	-122.433243	NA
66793	“CP - ROOM D14”	“ CANDLESTICK PARK ”	“SFO”	“CA”	37.712613	-122.387477	NA
1747	“NARA SUSHI”	“1515 POLK ST ”	“SFO”	“CA”	37.790716	NA	NA
509	“CAFE BAKERY”	“1365 NORIEGA ST ”	“SFO”	“CA”	37.754090	0.0	NA

(b) Inspection Table (input 2)

bus_id	Score	date
509	85	20130506
1747	93	20121204
16441	94	20130424
61073	98	20130422
66793	100	20130112

(c) Output Table

bus_id	name	address	Score	date	latitude	longitude
66793	“CP - ROOM D14”	“ CANDLESTICK PARK ”	100	20130112	37.712613	-122.387477

analyze San Francisco restaurant inspection data to understand the “cleanliness of the city”. The data is available from the city of San Francisco’s OpenData project. The challenge for the scientist is that the data needs some wrangling (e.g., merging, formatting, filtering) before it can be analyzed. For example, the scientist needs to join Table Ia, containing business information, and Table Ib, containing inspection data, and then filter rows with invalid latitude and longitude values to obtain the output shown in Table Ic. Zipfian Academy teaches how to write a series of python scripts that include calls to libraries that (1) merge two tables, (2) select the rows containing valid data, and (3) select the columns needed for the visualizations.

An alternative approach to writing such scripts that requires less user expertise involves the semi-automated synthesis of a program that performs the multi-step data wrangling.

We have built a small synthesis engine that generates data wrangling programs from tabular-form input/output examples, where the programs can consist of *union, join, filter rows, filter columns, compare columns, sort,* and *unique* operations, similar to that of Guo, et al. [12]. To illustrate, we describe the semantics of three operations used by synthesized candidate programs for the example in Table I (described next).

- *Join* takes two tables (`tab0`, `tab1`) and two column indices (`col0`, `col1`) and returns a single table containing each row of `tab0` combined with each row of `tab1` if `tab0.col0.val = tab1.col1.val`.
- *Filter rows* takes a table, column index, and list of values, and removes rows that contain any of the given values in the specified column.
- *Select* takes a table and a list of column indices as input and removes all columns that are not in the provided list.

The synthesis engine consumes input/output examples and systematically explores valid combinations of operations to generate programs that satisfy the examples. In our evaluation, this synthesis engine generates 4,418 satisfying programs for the example in Table I, two of which are shown in Figure 1. At this point, the user has to make a choice.

First, the user can select a program hoping that it meets the desired specification. Although all of the generated programs satisfy the constraints imposed by the example, the likelihood of selecting the right one is low as only two meet the user expectations (2/4,418 = 0.05%). Recent advances in ranking and navigating the space of synthesized programs [6], [13], [14] can mitigate this challenge, but increasingly that is not enough even for simple domains like tabular data wrangling due to the large space of synthesized programs with subtle but meaningful differences.

$t1 \leftarrow join(input1, input2, col0, col0)$
$t2 \leftarrow filter(t1, col5, [0.0, NA])$
$t3 \leftarrow filter(t2, col6, [0.0, NA])$
$output \leftarrow select(t3, [0, 1, 2, 9, 10, 5, 6])$

(a) A correct program

$t1 \leftarrow join(input1, input2, col0, col0)$
$t2 \leftarrow filter(t1, col5, [NA])$
$t3 \leftarrow filter(t2, col6, [0.0, -122.433243, NA])$
$output \leftarrow select(t3, [8, 1, 2, 9, 10, 5, 6])$

(b) An incorrect program that filters out the wrong rows

Fig. 1: Two programs generated during synthesis that meet the input/ouput example specification from Table I.

Second, the user can grab a program and tailor it to meet the desired specification. The likelihood of success in this case depends on the edit distance between the program selected and the desired program, and the ability of the user to tailor that program. We argue that as the targeted synthesis domains get more complex and reach more users, the distances will become larger and it is not clear that users should be expected to know the underlying programming language any better. Again, better navigation among the synthesized programs could help mitigate but not solve this challenge.

Third, the user can provide further input/output examples, strengthening the specification. The synthesis process can then leverage these input/output examples to refine the set of programs. Synthesis proponents favor this third option as it keeps users from sifting through many programs that may include challenging and unfamiliar constructs. However, *this assumes that the user is able to come up with input/output examples that can strengthen the specification, which is not always the case.* Once a solution space of programs has been generated after exhausting a set of given input/output examples, it can be challenging to provide further input/output examples that can effectively differentiate among those programs.

In the context of our scenario, consider the synthesized programs in Figures 1a (correct) and 1b (incorrect). Both programs have the same operations, just their filter parameters vary. To differentiate these simple programs the user must construct an input with a row that has `0.0` in the 5th column and not have `0.0, -122.433243, NA` on the 6th column, or an input with a row that has `-122.433243` in the 6th column and not have `0.0` or `NA` in the 5th column. As operations and programs grow more complex, it will become more difficult for users to come up with such differentiating inputs.

The approach we introduce in the next section addresses this challenge. Specifically, for the given scenario, it allows for the solution space for our running example to be pruned to two functionally equivalent programs after automatically generating three additional inputs coupled with user outputs.

III. Approach

We envision an approach aimed at narrowing the space of synthesized programs with less user effort, which we call SIPPS (Synthesizing Inputs to Prune Program Spaces). The approach is illustrated in Figure 2 and consists of:

1) Generating inputs from existing input/output examples.
2) Executing every input on every candidate program.
3) Clustering programs and selecting the "best" differentiating input based on the cluster structure.
4) Requesting the user to compute the output corresponding to that best input.
5) Pruning programs that produced a different output for that best input.
6) Repeating the process with candidate programs in the remaining cluster.

Implementation. We now describe an instantiation of SIPPS, as well as some alternatives we are exploring, for the data wrangling domain that operates on tabular forms such as the ones that we described earlier.

(1) For input generation, SIPPS takes the maximum number of rows in a table (it assumes the columns are fixed; S in Figure 2). When setting this parameter the user must consider that larger inputs are more likely to differentiate programs, but at the same time they will be more difficult for the user to understand and provide the corresponding output. SIPPS also takes the number of differentiating inputs to generate per iteration (n in Figure 2), and a terminating condition consisting of the number of inputs to try in an iteration before stopping the process (t in Figure 2).

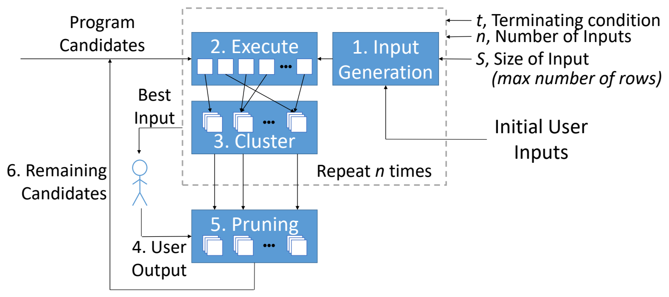

Fig. 2: SIPPS architecture

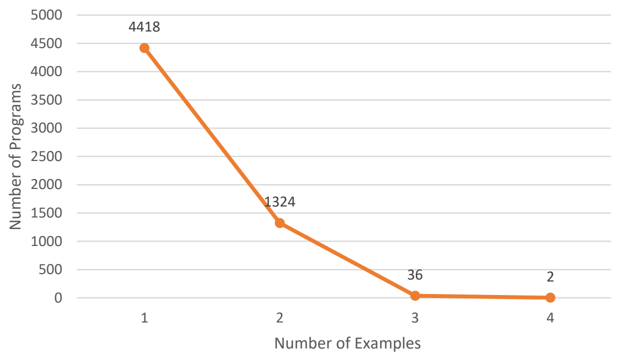

Fig. 3: Narrowing of the program space after each intput/ouput example is added to the synthesizer.

SIPPS then generates inputs by randomly permuting the example's input values within columns. For our scenario, to generate input values for "latitude", we sample from {`NA, 0.0, 37.712613, 37.790716, 37.754090`}. We assume these values cover a rich enough sample of initial data that has not yet been fully explored in combination with the values of other columns. We also believe that by using values familiar to the user it will be easier to derive the corresponding output later in the process.

We have also started to explore the use of symbolic execution to generate inputs that are more likely to differentiate programs by incorporating further constraints into the process (connecting "Program Candidates" with "Input Generation" in Figure 2), searching for inputs that satisfy all the constraints in one program but violate at least one constraint in another program. In spite of our initial enthusiasm for this symbolic implementation, the input space for this domain was small enough that generating random permutations and executing them on all programs was more cost-effective for identifying differentiating inputs than the symbolic approach.

(2) Once 1 to n differentiating inputs are generated, all candidate programs are executed and their outputs are hashed and used as the base for the clustering. This process can be computationally expensive but it is also trivially parallelizable.

(3) After obtaining all program outputs, the clustering process starts. Given an input, programs that generate the same output hash are clustered together. This simple and quick clustering scheme works well with the explored data set, but we have also started to incorporate notions of program distances to have more subtle forms of clustering. The clustering set resulting from an input is evaluated based on the number of clusters in the set. The input that produced the clustering set with the largest number of clusters is chosen. Ties are broken by choosing the input that produced the set with the smallest variance in the size of the clusters.

Scenario. In our motivating example, 4,418 programs are initially synthesized from the input/output example in Table I. We set the implementation maximum input size to `S=4` rows, after exploring with a range of sizes from 2 to 10. We set the number of differentiating inputs generated per iteration to `n=10`, with a bound of `t=100` non-differentiating inputs to be tried before terminating the process.

TABLE II: Random input generated by approach to differentiate programs

(a) Generated input table 1

bus_id	name	address	city	state	latitude	longitude	phone
1747	“CAFE BAKERY”	“ CANDLESTICK PARK”	“SFO”	“CA”	NA	-122.433243	NA
66793	“HAWAIIAN DR”	“2600 SAN BRUNO AVE”	“SFO”	“CA”	37.790716	0.0	NA
61073	“CAFE BAKERY”	“1365 NORIEGA ST ”	“SFO”	“CA”	0.0	0.0	NA
16441	“CAFE BAKERY”	“1365 NORIEGA ST ”	“SFO”	“CA”	37.712613	-122.404101	NA

(b) Generated input table 2

bus_id	Score	date
1747	100	20130424
1747	93	20130112
61073	100	20121204
66793	98	20130422

When SIPPS was run on the motivating example, the input that produced the best clustering during the first iteration is shown in Table II. This input partitioned the program space into 16 distinct clusters with a median cluster size of 145 programs. This input is shown to the user, who then provides the corresponding output (step (4) in Figure 2), which in this case is an empty table. Then, the cluster of programs that produced the empty table, which has 1,324 programs ((step (5) in Figure 2), is selected for the following narrowing iteration (step (6) in Figure 2). Figure 3 shows the number of candidate programs remaining after each iteration. The program space is reduced to two programs after three additional input/output examples, after which the process is terminated as the bound of 100 inputs is reached without being able to differentiate those programs. Manual inspection of the two remaining programs showed that they were functionally identical. The entire pruning process took 443 seconds.

IV. Related Work

The closest related work is that of Mayer et al. [13]. They target, within the domain of semi-structured data extraction, the same challenge by providing heuristics to rank synthesized programs, tools to navigate them, and reusing some of inputs (it assumes enough are already available) to narrow the candidates. We relax that assumption (so some inputs may indeed be missing), increasing the scope of the problem, and build on the insight that narrowing the space of generated programs can be stated as an input generation problem, which allows us to connect synthesis and test generation. Prophet [6] is similar in its goal of helping navigate the space of synthesized programs (i.e., patches for program repair), but it uses a language model to rank programs based on similarity to prior program patch structures. Oracle-guided component-based program synthesis [15] is similar in its use of differentiating inputs, but it seeks to find a single differentiating input, rather than a differentiating input that maximally divides the space of synthesized programs. Additional related work on program synthesis and its clients was briefly presented in Section I and II and is not discussed due to space constraints.

V. Conclusions and Future Work

As synthesis techniques become more powerful and their clients more prevalent and diverse, we contend that the bottleneck will shift to the users' ability to provide specifications that are effective at narrowing the set of synthesized solutions. We sketch an approach, SIPPS, aimed at aiding clients by generating this specification in the form of an input example that can differentiate among the synthesized programs, and asks the user just for the corresponding output, reducing the required effort. The approach is novel in how it connects synthesis and test generation to reduce that effort. We have yet to (1) assess the assumption about the cost of evaluating just an output being lower than that of generating a complete input/output example, (2) evaluate the cost-effectiveness of executing all synthesized programs during clustering, and (3) explore the spectrum of potential synergy between different test generation and synthesis approaches across different domains.

Acknowledgement

This work has been supported in part by National Science Foundation Awards #1526652, #1526253, and #1645136.

References

[1] E. Kitzelmann, *Inductive Programming: A Survey of Program Synthesis Techniques*. Springer Berlin Heidelberg, 2010, pp. 50–73.

[2] W. R. Harris and S. Gulwani, “Spreadsheet table transformations from examples,” in *Proceedings of the Conference on Programming Language Design and Implementation*. ACM, 2011, pp. 317–328.

[3] S. Gulwani, “Automating string processing in spreadsheets using input-output examples,” in *Proceedings of the Symposium on Principles of Programming Languages*. ACM, 2011, pp. 317–330.

[4] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in *Proceedings of the International Conference on Software Engineering*. ACM, 2016, pp. 691–701.

[5] ——, “Directfix: Looking for simple program repairs,” in *Proceedings of the International Conference on Software Engineering - Volume 1*. IEEE, 2015, pp. 448–458.

[6] F. Long and M. Rinard, “Automatic patch generation by learning correct code,” *SIGPLAN Not.*, vol. 51, no. 1, pp. 298–312, Jan. 2016.

[7] R. Singh and S. Gulwani, “Synthesizing number transformations from input-output examples,” in *Proceedings of the International Conference on Computer Aided Verification*. Springer-Verlag, 2012, pp. 634–651.

[8] A. Leung, J. Sarracino, and S. Lerner, “Interactive parser synthesis by example,” *SIGPLAN Not.*, vol. 50, no. 6, pp. 565–574, Jun. 2015.

[9] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, “Programming by sketching for bit-streaming programs,” *SIGPLAN Not.*, vol. 40, no. 6, pp. 281–294, Jun. 2005.

[10] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, “Codehint: Dynamic and interactive synthesis of code snippets,” in *Proceedings of the International Conference on Software Engineering*. ACM, 2014, pp. 653–663.

[11] K. T. Stolee, S. Elbaum, and D. Dobos, “Solving the search for source code,” *ACM Transactions on Software Engineering Methodology*, vol. 23, no. 3, pp. 26:1–26:45, May 2014.

[12] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer, “Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts,” in *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 2011, pp. 65–74.

[13] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani, “User interaction models for disambiguation in programming by example,” in *Proceedings of the Symposium on User Interface Software; Technology*. ACM, 2015, pp. 291–301.

[14] K. T. Stolee, S. G. Elbaum, and M. B. Dwyer, “Code search with input/output queries: Generalizing, ranking, and assessment,” *Journal of Systems and Software*, vol. 116, pp. 35–48, 2016.

[15] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, “Oracle-guided component-based program synthesis,” in *International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 215–224.