Program Synthesis: Synthesizing Operators for Integer Manipulation

Jayasurya Seenuvasan, Shalini Sai Prasad, N. S. Kumar

To cite this version:

Jayasurya Seenuvasan, Shalini Sai Prasad, N. S. Kumar. Program Synthesis: Synthesizing Operators for Integer Manipulation. 3rd International Conference on Computational Intelligence in Data Science (ICCIDS), Feb 2020, Chennai, India. pp.312-319, 10.1007/978-3-030-63467-4_26 . hal-03434779

Program Synthesis : Synthesizing Operators For Integer Manipulation[⋆]

Jayasurya Seenuvasan[1,2][0000−0002−8987−418X], Shalini Sai Prasad[1,3][0000−0002−2776−8437], and N S Kumar[1,4][0000−0001−8831−0625]

[1] PES University, Bangalore Karnataka 560085, India
[2] suryaseenu@gmail.com
[3] shalinisaiprasad@gmail.com
[4] nskumar@pes.edu

Abstract. We describe a language to synthesize a linear sequence of arithmetic operations for integer manipulation. Given an input-output example, our language synthesizes a set of operators to be applied to the input integers to obtain the given output. The sequence is generated by using Microsoft Prose, a program synthesis framework and the Genetic Algorithm. Our approach generates a set of ranked solutions that can be made unique on additional input-output examples that are consistent.

Keywords: Program Synthesis · Genetic Algorithm · PROSE · Arithmetic Operations.

1 Introduction

Program synthesis is the generation of programs in some underlying Domain Specific Language (DSL) from a high-level specification. The concept of program synthesis was theorized in the 1950s, where Alonzo Church defined the problem to synthesize a circuit from mathematical requirements [1]. Since then, various domains such as spreadsheet manipulation [2], data wrangling [3], and generation of competitive coding programs [4] have made use of the concept of synthesizing programs.

In this paper, we explore various techniques of synthesizing programs and develop a language to generate formulae consisting of arithmetic operations. We implemented inductive programming, which is the generation of a program from input-output specifications, through the use of Microsoft PROSE[5] and have used the Genetic Algorithm[6] in synthesizing a combination of expressions for a given list of integers.

The synthesizer was able to learn various operators to be linearly applied to different kinds of inputs.

[⋆] Supported by PES University.

2 Literature Survey

There are several ways to approach the task of synthesizing programs. The synthesis problem is reduced to a search problem that involves searching over a candidate space of possible programs to find an expression or combination of expressions that satisfies a set of constraints given by a user.

According to a study by Sumit Gulwani et al. [7], there are four methodologies of program synthesis: enumerative search, stochastic search, constraint solving, and deduction based program synthesis.

Oracle Guided Component-Based Synthesis[8] by Susmit Jha et al. delves into the synthesis of loop-free programs for bit manipulation. The approach involves the use of a combination of an oracle to guide learning from examples and synthesis of components from constraints using satisfiability modulo theory(SMT) solvers[9]. Given a set of bitwise operators and arithmetic operators, the tool they developed, Brahma synthesizes loop-free programs to obtain a one-bit vector from the other. The tool also helped in the deobfuscation of programs. The program is synthesized from a set of base components and a set of input-output examples. It uses an I/O oracle, which returns an output when given an input and a validation oracle, which verifies the correctness of a synthesized program. We have provided an extension of this concept with our synthesizer for integer manipulation.

Automatic String Processing in Spreadsheets Using Input-Output Examples, by Sumit Gulwani [2], is a benchmark of program synthesis. Here, a language to perform string manipulation tasks was developed, which synthesized a program from input-output examples. The DSL uses regular expressions to add constraints to the synthesized operators. The algorithm used also allows ranking in the case of multiple synthesized programs. This algorithm has been used as an add-in feature in Microsoft Excel, called Flashfill. The algorithm uses version space algebra and a Directed Acyclic Graph (DAG) representation of the code to be synthesized. The shortcomings of Flashfill include not being able to handle semantic synthesis, i.e., converting a date to a day as well as being limited to strings. Our DSL overcomes this data type dependency by implementing this feature for integers.

DeepCoder by Matej Balog et al. [4] adopted a machine learning technique to solving the task of program synthesis for competitive style coding problems by using a neural network to predict the program synthesized for various outputs given an input. They solve the problem of inductive program synthesis by defining a DSL with high-level operations. A recurrent neural network predicts how well an operation will solve a given problem and then searches over the program space using the predicted probabilities to find the solution. The limitations of DeepCoder include the inability to synthesize complex programs that require algorithms like dynamic programming as their DSL is limited. Another limitation is the number of examples to be given to the tool has to be five or more, which is not reflective of actual competitive coding examples that provide only two or three input-output examples. Our approach requires one example to synthesize multiple formulae, which are narrowed down if provided with additional examples.

3 Overview

Microsoft PROSE[5] is a program synthesis framework for creating custom Domain-Specific Languages. The framework requires that a user specifies four features of the DSL.

i) The grammar consisting of operations provided by the DSL
ii) The semantic meaning of these operations
iii) The constraints on the operations, i.e., witness functions
iv) The ranking of features in the case of multiple synthesized programs

PROSE takes an input-output example, generates the operator(s) to be learned, and applies the synthesized operators on a new input to produce an output. We have incrementally created the synthesizer using four approaches to tackle more complex inputs. We define our DSL as the list of arithmetic operations: Addition, Subtraction, Multiplication, and Division.

The types of syntheses provided by our DSL are:
i) Synthesis of a binary operator and an integer.
ii) Synthesis of a binary operator.
iii) Synthesis of a single operator to reduce a list of integers to an aggregate.
iv) Synthesis of a sequence of operators to reduce a list of integers to an aggregate.

3.1 Approach 1: Synthesizing an operation and an integer

For this type of program, the input to the PROSE framework is two integers: x and y, such that x is the input to the synthesized program, and y is the expected output. The synthesizer generates an operator *op* and an integer *k*, such that *op(x, k)* results in *y*. The grammar is as follows:

$$S \rightarrow Add(x,k)|Subtract(x,k)|Multiply(x,k)|Divide(x,k)$$

If the input-output example given is

$$x = 5, \qquad y = 20$$

The programs synthesized are

$$Multiply(x, 4)$$

$$Add(x, 15)$$

$$Subtract(x, -15)$$

If a second example is given where,

$$x = 2, \qquad y = 8$$

The synthesized programs narrow down to a program that satisfies both input-output examples, i.e.,

$$Multiply(x, 4)$$

The synthesizer generates multiple operators to obtain the final solution, and these solutions are ranked by assigning a score to each synthesized program. The programs are listed such that the absolute value of the smallest operand synthesized would get a higher rank. Thus, in the above example, the first input-output example would rank *Multiply*(x,4) first.

3.2 Approach 2: Synthesizing a binary operation

This type of program takes a pair of integers x and y as input and an integer j as the expected output. The synthesizer generates as operator *op* such that *op(x,y)* results in *j*. The grammar for this synthesis required the use of delegates. The grammar is as follows:

$$S \to Eval(x, y, Func)$$

where *Func* is synthesized based on the output given along with the witness functions provided. The synthesizer compares the expected output j and the outputs obtained by applying all the operators on the given inputs x and y. If the outputs match, the synthesizer learns that the operator must be used for subsequent inputs. Multiple operators may be learned in this program and we use arithmetic precedence to rank the learned operators.

If the input-output example given is

$$x = 2, \qquad y = 2, \qquad j = 4$$

The programs synthesized are

$$Eval(x, y, Multiply)$$

$$Eval(x, y, Add)$$

If a second example is given where,

$$x = 3, \qquad y = 3, \qquad j = 9$$

The synthesized programs narrow down to a program that satisfies both input-output examples, i.e.,

$$Eval(x, y, Multiply)$$

3.3 Approach 3: Synthesis of a single operator to reduce a list of integers to an aggregate

This type is an extension of the previous approach, but the synthesized operator is applied to a list of integers *l= [x1, x2, x3, ..., xn]* and an expected output integer *j*. The synthesizer applies the operator sequentially between each element in the list and checks if the aggregate is equal to the expected result. The grammar for this type also uses delegates.

$$S \rightarrow Eval(< list >, Func)$$

The input-output example given is

$$l = [1, 2, 3], \qquad j = 6$$

The programs synthesized are

$$Eval(l, Multiply)$$

$$Eval(l, Add)$$

If a second example is given where,

$$l = [1, 2, 3, 4], \qquad j = 10$$

The synthesized programs narrow down to a program that satisfies both input-output examples, i.e.,

$$Eval(l, Add)$$

3.4 Approach 4: Synthesis of a sequence of operators to reduce a list of integers to an aggregate using Genetic Algorithm

The last type is where a sequence of operators is synthesized. The input to the synthesizer is a list of integers *[x1, x2, x3, ..., xn]* and an expected output *j*. The synthesizer generates a sequence of operations to be applied sequentially on each list element to obtain *j*.

If the input-output example given is

$$l = [1, 2, 3, 4], \qquad j = 4$$

The synthesizer returns the following sequences of operators:

$$[+, /, *]$$

$$[+, -, +]$$

If a second example is given where,

$$l = [4, 4, 3, 5], \qquad j = 10$$

The synthesized programs narrow down to a program that satisfies both input-output examples, i.e.,

$$[+, -, +]$$

We have used the Genetic Algorithm [12] to help us generate the possible sequence of operators that can be applied to the input list of integers. The algorithm initially generates a random list of sequences to form the initial population. The size of the initial population is 25% of the candidate space of all series of operators possible. Each sequence of operators is applied to the input list to compute the output.

During subsequent iterations of the algorithm, the fitness score for each series is calculated as the absolute difference between the generated output and expected output. The lesser the absolute difference between the outputs, the fitter is the individual. 40% of the fittest individuals pass onto the next generation. The remaining individuals for the population for the next generation are computed by performing crossover and mutation on randomly picked pairs of individuals as parents in the current population. Each parent produces two children for the next population. We have used single-point crossover and coin-flip mutation for this approach.

The algorithm runs for a certain number of iterations that were varied during testing, and finally returns combinations of operators that give the expected output.

4 Results

We illustrate the working of the genetic algorithm with an increasing number of operations to learn. We test the algorithm by providing simple examples where one or more known solutions exist and tabulating the results, as in Table 1.

Table 1. Tests run on Genetic Algorithm

Iterations	Input	Output	No. of Sequences Learned	Time(ms)
100	1 2 3	6	2	957
150	1 2 3 4	4	2	932
200	8 4 6 2 1	13	5	982
250	9 2 3 3 7 2	60	1	1286
300	5 3 4 6 5 8 3	15	19	2122
350	4 6 6 4 9 5 3 6	53	29	6934
400	6 5 4 2 7 9 5 8 3	36	222	27771
450	1 2 3 4 5 6 7 8 9 1	46	176	128429

The algorithm runs until convergence and generates sequences of operators that result in the output. The algorithm takes exponential time to converge as the input size increases linearly, as shown in the graph in Fig. 1.

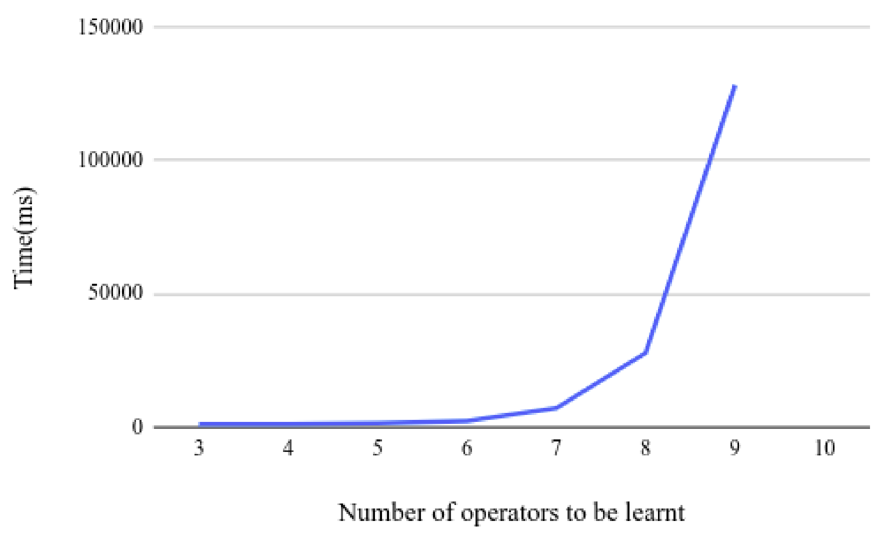

Fig. 1. Graph showing convergence time based on input size

It is observed that the genetic algorithm may result in the same sequence to be generated multiple times as a result of crossover and mutation. The algorithm may not generate some possible sequences because of the randomness of the initial population. These are the limitations of choosing such an approach as compared to an exhaustive search.

5 Future Work

We are attempting to strengthen the implementation of the genetic algorithm by eliminating duplicate sequences that could potentially reduce the time taken by it to converge. Compared to an exhaustive search the number of sequences tested are lesser but due to the crossover and mutation overhead, the genetic algorithm is much slower than the exhaustive search for this small subset of inputs.

Our program's DSL is limited to simple arithmetic operations, namely Addition, Subtraction, Multiplication, and Division. The DSL can be extended to include relational operators, and the grammar could be modified to include statements such as branches and loops to generate complex programs.

References

1. Friedman, J.: Review:Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesis. Summaries of talks presented at the Summer Institute

for Symbolic Logic Cornell University, 1957, 2nd edn., Communications Research Division, Institute for Defense Analyses, Princeton, N. J., 1960, pp. 3–50. 3a-45a. Journal of Symbolic Logic. 28, 289–290 (1963).

2. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL11): January 26-28, 2011, Austin, Texas, USA. Assoc. for Computing Machinery, New York, NY (2011).
3. Gulwani, S., Jain, P.: Programming by Examples: PL Meets ML. Programming Languages and Systems Lecture Notes in Computer Science. 3–20 (2017).
4. Balog, M., Gaunt, A.L., Brockschmidt, M.L., Nowozin, S.L., Tarlow, D.L.: Deepcoder: Learning to write programs. . In: International Conference on Learning Representations. Toulon, France, April 24 - 26 (2017).
5. Microsoft Program Synthesis using Examples SDK, https://microsoft.github.io/prose/.
6. Mallawaarachchi, V.: Introduction to Genetic Algorithms - Including Example Code, https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3.
7. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. now Publishers Inc., Boston (2017).
8. Jha, S.A., Gulwani, S.A., Seshia, S.A., Tiwari, A.A.: Oracle-guided component-based program synthesis. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE 10. 1, 215–224 (2010).
9. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability Modulo Theories. Handbook of Satisfiability. 185, 825–885 (2009).