Design and Applications of a Program Synthesizer

Candidate Number: 1076461

University of Oxford

Project Report

Honour School of Computer Science, Part B

Trinity 2025

Abstract

Program Synthesis is the field of programs which write programs, usually with a formal logical specification. In this project, I focus on designing and building a PBE (Programming-by-Example) Synthesizer, with the primary goal of being able to deduce from a piece of data (such as, a set or sequence of integers), a program which might have generated it, or which can explain most of its content. Synthesizers of this sort are useful for pattern recognition in settings where datasets are small and approximate answers are not desired. I use a combination of enumerative and Montecarlo techniques, and discuss the practical and theoretical implications of different possible design choices. The resulting tool is very general, and can be used to evaluate and compare the expressivity of programming languages, or to try to determine the Kolmogorov Complexity (or Minimum Description Length), of different sequences in any (total) programming language. I discuss the results of some such experiments, and give outlines of several other similar potential future experiments.

Contents

1 Introduction

1.1 Background

Program synthesis is the problem of generating a program from a certain specification, often expressed as a logical constraint. The problem we consider is one of Programming-by-Example synthesis, where instead of a formal specification, we attempt to generate a program using input-output pairs. This is useful in situations where we are interested in discovering exact, possibly complex patterns in a piece of data. This method has strong limitations, as you might expect, but has the advantages that it can work even when datasets are far too small to use statistical methods, that we make very few assumptions about the data (beyond the fact that it has some computable pattern), and that once we generate a program, we can examine and completely understand its behaviour.

Some notable applications of this kind of PBE synthesis include:

- "Flash Fill" [1]: This is the technology behind Microsoft Excel's autocomplete feature. It is what allows the software to detect and extend a user's actions using very few examples (often even just one).
- "IntelliCode" [2]: This is a feature of Visual Studio Code which monitors a user's actions and attempts to discover the pattern of user's edits, and to complete their refactor.
- Query Synthesis [3]: PBE has been used to develop tools which construct SQL queries based on small numbers of tuple examples of rows which should be fetched.[1]

PBE problems are, by their nature, underspecified, as there will usually be many (even infinitely many) programs satisfying any set of input-output constraints, but it does allow us to generate reasonable conjectures about the underying structure of any given piece of data. It has the very strong advantage that we do not have to write any kind of formal specification for our program, which is often not much easier than writing the program itself.

[1] This is not the same as Query-by-Example, or QBE, which is a feature of many databases which provides a more user-friendly querying interface, and which has existed since the 1970s.

1.2 Goals & Motivation

Although the tools developed throughout this project are very general and could be applied to many different areas, we focus especially on synthesizing programs generating integer sequences. One motivating application of this could be in mathematical research, where we might wish to guess patterns in structured integer sequences. This is exactly the concern that the Online Encyclopedia of Integer Sequences (OEIS) was invented to address, and we will use its database to evaluate our program. However, we also discuss how these tools could be applied elsewhere.

We also focus on synthesizing small solution programs. Firstly, because this allows us to avoid overfitting (for example, programs which simply match their inputs to the given input-output pairs, without giving any insight into the larger pattern). Secondly because it will be helpful in eliminating redundant programs (i.e., programs which are syntactically distinct but semantically identical), vastly reducing our search space. And thirdly, because the minimum description length of a mathematical object or piece of data (i.e., the length of the shortest program generating) is known as its Kolmogorov Complexity, and though it is incomputable, it is of theoretical interest[2].

Of course, we cannot hope to produce a perfect synthesizer, so a great deal of effort has gone into the feasible search space of our synthesizer. This translates to trying to shrink the program space as much as possible, and expanding the portion of the space we can feasibly examine as much as possible. This means both pruning our search wherever possible and writing efficient code, both of which come with the cost of some complexity.

1.3 Report Structure

Each of the main chapters of this report covers a different aspect of this program synthesizer, in varying levels of detail. We discuss the reasons behind each major design decision, as well as their strengths and limits compared to other possible decisions.

The main chapters cover:

- The interpreter used to evaluate programs.

[2]Notably, there are theoretical results showing that if we could compute Kolmogorov complexity, we could approximate Algorithmic Probability, which in turn allows us to compute the source of an infinite sequence correctly and with relatively low error. This formalizes our intuition that, when looking at a piece of data from an unknown source, a "simple" program is more likely to have generated it than a "complex" one. See [4] for more information.

- The enumerator which allows us to iterate over certain program spaces.
- The incorporation of semantic analysis into our search.
- The use of the Metropolis-Hastings algorithm to expand the scope of our search space.
- The results and effectiveness of the synthesizer.

1.4 Choice of Technology

Because of the need for efficiency, I chose to use Rust to implement this synthesizer. This did complicate its implementation, it allowed me to make optimizations which would not have been possible in a higher level language (such as Haskell, which is probably the language which would have made a simple version of this implementation the simplest).

I try wheverever possible to show the relevant code, but for the sake of brevity & readability, I will often omit parts of the code which add complexity without offering any insight (such as casts between references & pointer types, unreachable code branches, etc...), meaning code snippets as they appear in this document may not be valid.

2 Interpreter

2.1 Interpreter

Bibliography

[1] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[2] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. On the fly synthesis of edit suggestions. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[3] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. Discovering queries based on example tuples. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[4] Ray J Solomonoff. Algorithmic probability: Theory and applications. *Information theory and statistical learning*, pages 1–23, 2009.