Synthesis of Recursive Functions from Sequences of Natural Numbers*

Thibault Gauthier

Czech Technical University, Prague

Abstract Given a sequence of natural numbers, we lay the foundations for automatically synthesizing a program that generates this sequence.

Techniques for synthesizing programs have been developed in the course of the last thirty years [3] in the domain of inductive logic programming. The subjects studied cover recursion, higher-order programs, optimal programs and library building. In parallel, program synthesis tasks have been attempted by reinforcement learning. Most recent developments have implemented additional features on top of the learning loop such as a library building mechanism [4] or inputs from a deductive reasoning system [2]. We as well have investigated how to synthesize mathematical objects, that could be considered programs such as: combinators [5], Diophantine equations [5] and set theory formulas [1].

The aim of our project is to automatically construct a program that generates a given sequence of natural numbers. Mastering this task would have important implications. Indeed, finding a "small" program matching a partial sequence would produce a powerful predictive model. Such automation could be applied any time a scientist is confronted with some unknown data representable with natural numbers. To train the automation, we will rely on a set of natural number sequences collected by mathematicians available in the online encyclopedia of integer sequences (OEIS)[6]. Indeed, there are currently 343338 integer sequences in the OEIS, 325191 of which only contain natural numbers. Some of the sequences in this dataset are simple such as the sequence of squares $A000290$. Others are related to open problems. For example, if the sequence $A073101 : [1, 1, 2, 5, 5, 6, 4, 9, 7, 15, 4, 14, 33, \ldots]$ contains 0 then the Erdős-Strauss conjecture is false. For this reason, this problem set is ideal as it provides a gradual learning curve and a large pool of interesting objectives. If a significant portion of the problems is solved, this will indicate that the automation has acquired an understanding of the programming task.

Design of the Programming Language Our language is inspired by the language of recursive functions defined in computability theory. It contains the projections represented by variables in the examples and the basic operators $0, 1, +, -, \times$, *modulo*, *division*, a conditional operator $if(a, b, c)$ that tests if a is equal to 0 and returns b and otherwise returns c, and recursive calls. This language is designed to be expressive enough to construct functions generating interesting sequences in a natural way. By synthesizing a function f from $\mathbb{N}$ to $\mathbb{N}$, we can generate the sequence of elements $f(n)$ for $n \in \mathbb{N}$. For example, a program generating the Fibonacci sequence $A000045$ can be written as:

$$f(n) := \text{ if } n = 0 \text{ then } 0 \text{ else if } n - 1 = 0 \text{ then } 1 \text{ else } f(n-1) + f((n-1)-1)$$

Using a subprogram g, we can also get a program with linear time complexity:

$$g(a, b, c) := \text{ if } a = 0 \text{ then } b \text{ else } g(a-1, c, b+c) \qquad f(n) := g(n, 0, 1)$$

*Supported by the Czech Science Foundation project 20-06390Y

Synthesis Strategy There are many ways to explore the space of programs just defined. Since programs can be represented as trees, we propose to use a bottom-up approach relying on a stack of program trees as intermediate synthesis steps. The synthesis process starts with an empty stack and updates the program stack depending on the operator chosen as illustrated on the following example:

$$[\,] \to [n] \to [n-1] \to [f(n-1)] \to [n, f(n-1)] \to [n + f(n-1)] \to$$

$$[0, n + f(n-1)] \to [n, 0, n + f(n-1)] \to [\text{if } n = 0 \text{ then } 0 \text{ else } n + f(n-1)]$$

After each synthesis step that ends with a single program tree t in the stack, we check if the program $f(n) := t$ generates the targeted sequence. When executing $f(i)$ we limit the number of execution steps to $30 \times (i+1)^3$. This bound was chosen to allow the program to run longer for larger input but only in a cubic manner. Therefore, in general programs with exponential complexity will not be synthesized.

Programs in the stack are complete trees and therefore can be defined by their behavior on the inputs which may facilitate learning their embeddings. In contrast, in a top-down approach the partially synthesized program trees have open branches and thus more complex semantics.

Reinforcement Learning We rely on the deep reinforcement learning framework developed in [5] to train a machine learning model (tree neural networks) on how to select the right synthesis step given learned embeddings for the current program stack and the a prefix of the targeted sequence (first 16 elements). At each generation, a pool of 200 sequences is targeted. The TNN learns from each attempt and re-uses this knowledge for the next generation.

Test Run The framework was tested on a set of 2000 sequences generated from random programs of size less or equal to 30. It was run for 200 generations. After that time, the system had synthesized programs for most of the sequences (first 16 elements).

Conclusion For the described synthesis task, we have designed a small but expressive programming language, have chosen a suitable synthesis strategy and have started testing a reinforcement learning framework. In the future, we would like to scale this approach on the OEIS and improve it by incorporating data augmentation techniques, library building mechanisms and deductive reasoning abilities.

References

[1] Chad E. Brown and Thibault Gauthier. Self-learned formula synthesis in set theory. *CoRR*, abs/1912.01525, 2019.

[2] Yanju Chen, Chenglong Wang, Osbert Bastani, Isil Dillig, and Yu Feng. Program synthesis using deduction-guided reinforcement learning. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 587–610. Springer, 2020.

[3] Andrew Cropper, Sebastijan Dumancic, and Stephen H. Muggleton. Turning 30: New ideas in inductive logic programming. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4833–4839. ijcai.org, 2020.

[4] Kevin Ellis, Catherine Wong, Maxwell I. Nye, Mathias Sablé-Meyer, Luc Cary, Lucas Morales, Luke B. Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *CoRR*, abs/2006.08381, 2020.

[5] Thibault Gauthier. Deep reinforcement learning for synthesizing functions in higher-order logic. In Elvira Albert and Laura Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pages 230–248. EasyChair, 2020.

[6] Neil J. A. Sloane. The on-line encyclopedia of integer sequences. *Electron. J. Comb.*, 1, 1994.