Evolutionary Computation for Program Synthesis in SuSLik

Yutaka Nagashima[1]

Independent Researcher, Cambridge, the United Kingdom
united.reasoning@gmail.com

Abstract

A deductive program synthesis tool takes a specification as input and derives a program that satisfies the specification. The drawback of this approach is that search spaces for such correct programs tend to be enormous, making it difficult to derive correct programs within a realistic timeout. To speed up such program derivation, we improve the search strategy of a deductive program synthesis tool, SuSLik, using evolutionary computation. Our cross-validation shows that the improvement brought by evolutionary computation generalises well to unforeseen problems.

1 Deductive Program Synthesis

A far-fetched goal of artificial intelligence (AI) research is to build a system that writes computer programs for humans. To achieve this goal, researchers take two distinct approaches for program synthesis: deductive program synthesis and inductive program synthesis.

Both approaches attempt to produce programs requested by human users. The difference lies how they produce programs and the guarantee of the resulting programs: deductive synthesis tries to *deduce* programs that satisfy specifications, while inductive program synthesis tries to *induce* programs from examples. A notable example of inductive program synthesis is the automated spreadsheet data manipulation implemented as an add-in for Microsoft Excel spreadsheet system [1].

While such inductive synthesis alleviates the burden of implementation by guessing programs from given input-output examples, in inductive synthesis the resulting programs are never trustworthy: there is always a risk that incorrect generalisation results in programs that are correct for the present examples but not for future cases.

Deductive synthesis overcomes this limitation with formal specifications: it allows users to formalise *what* they want as specifications, whereas inductive synthesis tools guess *how* programs should behave from examples provided by users. Thus, in deductive synthesis providing formal specifications remains as users' responsibility. The upside of deductive synthesis is, however, users can obtain *correct* programs automatically upon success. This correctness assurance is particularly useful when it comes to synthesising imperative programs with pointers, as manually developing heap-manipulating programs is known to be error-prone.

SuSLik [4], for example, is one of such deductive synthesis tools. It takes a specification provided by humans and attempts to produce heap-manipulating programs satisfying the specification in a language that resembles the C language. Internally, this derivation process is formulated as proof search: SuSLik composes a heap-manipulating program by conducting a best-first search for a proof goal presented as specification. The drawback is that the search algorithm often fails to find a proof within a realistic timeout. That is, even we pass a specification to SuSLik, SuSLik may not produce a program satisfying the specification. According to Itzhaky *et al.* [2],

experiment	gen-0	gen-20	gen-40
1st (32)	18	16	15
2nd (41)	21	21	15
3rd (31)	18	16	15
4th (31)	16	13	13

(a) Unsolved problems in the training set

experiment	gen-0	gen-20	gen-40
1st (33)	22	16	16
2nd (24)	17	16	15
3rd (34)	22	18	16
4th (34)	24	21	18

(b) Unsolved problems in the validation set

different synthesis tasks benefit from different search parameters, and that we might need a mechanism to tune SuSLik 's search strategy for a given synthesis task.

2 Evolutionary Computation for Better Search Strategies

To address this issue, we built an evolutionary framework that improves SuSLik's synthesis strategy. Basically, this framework tries to identify suitable search parameters for SuSLik's proof search strategy. These parameters include the weights associated with each step of search. Our artefact is publicly available at GitHub [3].

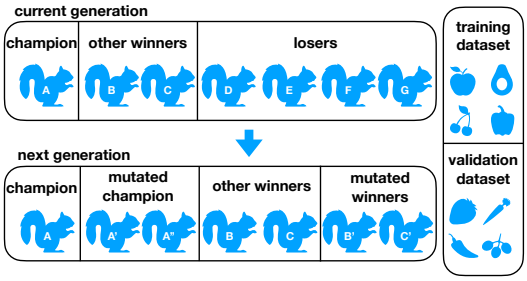

Figure 1: Evolution of SuSLik instances

In this framework, we firstly create a pair of specification sets: one for training and the other for validation. Secondly, we produce the initial population consisting of 40 instances of SuSLik by mutating the original search parameters. In each generation, we assign the specifications in the training set to each SuSLik instance. Then, we count how many specifications each SuSLik instance manages to solve within 2.5 seconds. We take 20 best performing instances and produce new mutants from them. Then, we pass these winners and their mutants to the next generation and repeat this process 40 times. To accelerate evolution, we allow the champion of each generation to produce two instances of mutants as shown in Figure 1.

We experimented our framework four times. Table 1a shows the results of training. For example, the second row in Table 1a reads as follows: in the first experiment 32 specifications fell into the set for training, and 18 specifications were left unsolved by the best SuSLik instance in the zeroth generation. This number decreased to 16 and 15 for the 20th and 40th generation, respectively.

In our experiments, we conducted cross-validation for each generation. Their results are shown in Table 1b. Note that we used a fixed pair of training set and validation set throughout the evolution of each experiment to maintain the distinction between the two sets. All these four experiments showed that improvements from training sets translates to improvements on validation sets despite the small size of dataset. That is, we found that

there are strategies that tend to perform better for unforeseen problems, and we can find such strategies using genetic algorithms.

References

[1] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330. ACM, 2011.

[2] Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. Deductive synthesis of programs with pointers: Techniques, challenges, opportunities - (invited paper). In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 110–134. Springer, 2021.

[3] Yutaka Nagashima. https://github.com/yutakang/suslik/tree/evolutionary, 2021.

[4] Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3(POPL):72:1–72:30, 2019.