Synthesizing Programs from Program Pieces using Genetic Programming and Refinement Type Checking

Sabrina Tseng Erik Hemberg Una-May O'Reilly

Massachusetts Institute of Technology, Cambridge, MA 02139, USA

Abstract

Program synthesis, the automatic construction of a computer program from a user specification, is a central problem in the field of AI and has broad applications in software engineering, such as automating tedious parts of the software development process, and allowing non-programmers to easily work with data [1].

However, existing program synthesis methods remain limited in scalability - many are focused on synthesizing small method bodies, or are constrianed in scope.

Our contributions are:

- A **program piece model** for program synthesis that enables the use of built-in functions and external libraries to scale to more complex programs
- A new **fitness function for genetic programming (GP) search**, which evaluates programs composed of pieces using **refinement type verification**
- An **evaluation** of the new fitness function in this model

Method

In our program synthesis model, programs are represented by a sequence of **program pieces**, which are functions provided by the user. To find a correct program for a given problem, we use a standard **genetic programming (GP)** search algorithm [2] as illustrated below.

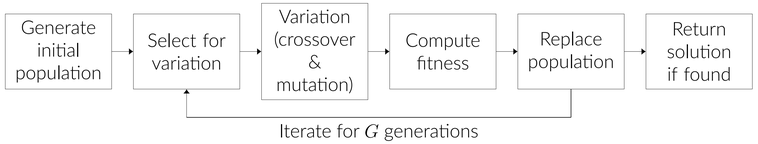


Figure 1. Base GP algorithm

The **fitness function** quantifies how "close to correct" a candidate program is, and is used as a heuristic to guide selection and replacement. For our fitness function, we use **refinement type checking** [3], a formal verification method that checks predicates on input and output types. In particular, we define a new fitness function shown below that is derived from refinement type checking using LiquidHaskell [4]:

$$f_{RT}(c) = \begin{cases} 0.5 - \frac{s}{2|c|} & \text{if } s > 0 \text{ (syntax checking fails)} \\ 1 - \frac{t}{2|c|} & \text{if } s = 0 \text{ (syntax checking succeeds)} \end{cases} \quad (1)$$

- s = number of syntax errors
- t = number of refinement type errors (can only be > 0 if $s = 0$)
- $|c|$ = size of the chromosome in GP; also equal to the max possible values for s and t

This fitness function has the following key properties:

1. Fewer errors results in higher fitness
2. Syntax errors result in lower fitness than refinement type errors, as they usually indicate structural issues such as missing function definitions

Experiments & Results

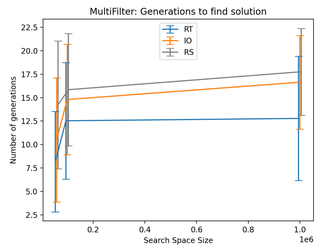
Experimental Setup

We compare our GP algorithm to two baselines for evaluation. The three variants are:

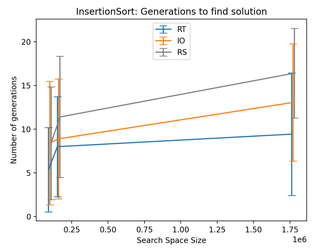
RT	`RefinementTypes`	GP search using our new fitness function defined in Equation 1.
IO	`IOExamples`	GP search using a baseline fitness function: accuracy on a small, but well-covered set of example input-output cases.
RS	`RandomSearch`	Random generation of programs from the provided program pieces. This baseline helps verify that GP is well suited to our program synthesis model.

We report performance as the number of generations each variant takes to find a correct program, for a set of 3 benchmark problems. To evaluate scalability, we also introduce 3 different search space sizes for each problem by using different sets of program pieces.

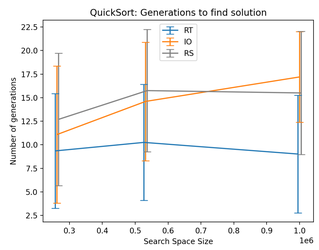
Results



(a) MultiFilter



(b) InsertionSort



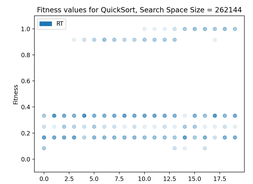
(c) QuickSort

Figure 2. For each of the 3 benchmark problems, we run 60 trials per search space size and variant. We plot the average number of generations taken to find a correct solution on the y-axis, against the search space size on the x-axis. Error bars represent one standard deviation. Note that each variant runs on the same search space sizes, but in the graph, the x-values are offset slightly for visibility.

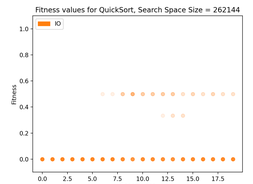
- In general, **finds solutions in fewer generations** than the two baselines
- Achieves an **average performance improvement of 20% over `IOExamples`**
  - Improvement is statistically significant ($p < 0.05$) in 6 out of 9 cases
- Achieves an average performance improvement of 32% over `RandomSearch`
  - Improvement is statistically significant ($p < 0.05$) in all cases
- **Improvement remains statistically significant for higher search space sizes**, demonstrating the potential to scale to larger problems

Discussion

We hypothesize that a key reason for the performance improvement we observe is the **increased diversity of fitness values**. We can see in Figure 1 that `RefinementTypes` yields more distinct fitness values than `IOExamples`, both for programs with and without syntax errors.



(a) `RefinementTypes`



(b) `IOExamples`

Figure 3. Scatter plots showing fitness values per generation for one trial run of QuickSort. Each point (g, f) represents an individual in generation g with a fitness value of f, where opacity increases with the number of individuals with fitness value f.

In addition, for problems where behavior can be expressed through refinement types (including our 3 benchmarks), `RefinementTypes` can provide a better guarantee of correctness than `IOExamples`, which is highly dependent on the coverage of the example cases.

Implications & Future Work

- It is possible to **express complex programs using program pieces and refinement types**, such as sorting. However, **refinement types are not applicable to every problem**. For example, string manipulations may be hard to express using refinement types.
- In this model, **using refinement type checking to evaluate fitness within GP search can provide an improvement over using an example-based fitness evaluation**

These results merit further investigation into different ways of incorporating symbolic solving within GP search to improve scalability. In particular, future work might include:

- Evaluating a wider set of benchmarks, including more complex problems with larger search spaces, and different types of problems
- Exploring ways to formalize and automate the construction of program pieces, for example by searching the standard library, or using GP to "fill in" missing pieces

References

[1] Cristina David and Daniel Kroening.
Program synthesis: challenges and opportunities.
Philosophical transactions. Series A, Mathematical, physical, and engineering sciences, 375, 2017.

[2] John R. Koza.
Survey of genetic algorithms and genetic programming.
Proceedings of WESCON'95, pages 589–, 1995.

[3] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala.
Abstract refinement types.
In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 209–228, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones.
Refinement types for haskell.
SIGPLAN Not., 49(9):269–282, aug 2014.