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

No Author Given

No Institute Given

Abstract. Program synthesis automates the process of writing code, which can be a very useful tool in allowing people to better leverage computational resources. However, a limiting factor in the scalability of current program synthesis techniques is the large size of the search space, especially for complex programs. We present a new model for synthesizing programs which reduces the search space by composing programs from program pieces, which are component functions provided by the user. Our method uses genetic programming search with a fitness function based on refinement type checking, which is a formal verification method that checks function behavior expressed through types. We evaluate our implementation of this method on a set of 3 benchmark problems, observing that our fitness function is able to find solutions in fewer generations than a fitness function that uses example test cases. These results indicate that using refinement types and other formal methods within genetic programming can improve the performance and practicality of program synthesis.

1 Introduction

Program synthesis, the automatic construction of a computer program from a user specification, is a challenging and central problem in the field of artificial intelligence (AI) [1]. Programming has been classified as "AI-hard" [2] since all problems in AI can reduce to programming, and thus it is considered one of the most difficult computational problems. Our progress in program synthesis serves as a good benchmark for how close we are to achieving artificial intelligence. In addition, program synthesis has broad applications in software engineering. For example, software development often entails refactoring old code to improve its structure or readability without changing its behavior, and program synthesis can help automate that process. In addition, program synthesis can allow non-programmers to efficiently perform computational tasks [4].

Two of the main approaches to program synthesis are stochastic search through genetic programming [3], and formal verification methods such as symbolic solving [1]. However, solver-based methods do not scale beyond small programs such as introductory programming problems [6]. In this paper, we propose a new program synthesis model which leverages pre-existing code, in the form of functions that we call “program pieces”, and synthesizes the high-level program structure. This model allows for an approach that incorporates refinement type checking [10], a formal verification method, into genetic programming search.

Genetic programming (GP) is a search technique that begins with an initial population of programs from the search space, and combines the most “fit” programs in ways similar to biological evolution to move towards an optimal solution [9]. In particular, GP proceeds in generations, where in each generation the search selects the most fit programs and varies them to get a new generation of more evolved and more fit programs. In GP systems, the performance of the search depends heavily on the fitness function, since incorrect programs need a good heuristic to optimize [1, 8]. A common fitness function is the program’s accuracy on a set of example inputs and outputs. However, having a large set of examples is computationally expensive [7], while a small set of examples leads to under-specification and often the wrong generalizations [1]. NetSyn [12] showed that using neural networks to learn a fitness function can improve GP performance. This suggests that there is still room for improvement in the design of the fitness function.

On the other hand, formal verification methods can be used to synthesize programs through symbolic proofs of satisfiability and correctness. One example of a formal verification method is refinement type checking [10], which is a stronger form of type checking that can enforce predicates on types. Specifically, a user can define stricter input and output types for a function using refinement types, so that the refinement type check enforces expected preconditions and postconditions. The liquid type system [11] allows for efficient static checking of refinement type safety, without requiring every expression to be manually annotated. However, as mentioned above, formal methods alone do not scale well beyond small programs.

Our key idea in this paper is to improve scalability by decomposing programs into *program pieces*, which are functions provided by the user or imported from a library. We form candidate programs by composing program pieces. By abstracting away logical units of code into these program pieces, we reduce the search space for the synthesis, thus enabling us to solve larger synthesis problems. Furthermore, this allows users of our system to make use of built-in functions or external libraries, which can provide complex logic for free.

An additional benefit of this decomposition is that we can use refinement types to specify the input and output types of program pieces, which specifies the overall intended behavior of the program we want to synthesize. In our proposed system, we use refinement type checking as a fitness function within our GP algorithm. In particular, we define a novel fitness function based on the number of errors that result from the refinement type check, so that programs with fewer errors have better fitness. Using this fitness function, we observe that the GP search converges towards a program that has no type errors, which we consider to be correct since the refinement types specify the intended behavior. In addition, unlike fitness functions based on input-output examples which are under-specified as mentioned above, refinement types provide a formal specification of the entire input and output spaces.

We present the following contributions in this paper:

- A general-purpose program synthesis model that synthesizes programs by composing preexisting program pieces
- A fitness function on programs composed of pieces that enables GP to find good programs, derived from the number and type of errors that result from refinement type checking
- An evaluation of the new fitness function in this model

We evaluate the performance of our proposed fitness function against a fitness function that uses accuracy on input-output examples. We find that on average, with our refinement type-based fitness function, the GP search finds solutions in about 20% fewer generations than when we use input-output examples.

The remainder of the paper is structured as follows: first, we outline our methods, including how we translate the refinement type check into a fitness function (Section 2). Next, we describe our experiments and results (Section 3). Finally, we discuss related work (Section 4) and conclusions (Section 5).

2 Method

We will present our method in 4 sections. First, we describe our program synthesis model, defining program pieces and introducing a running example (2.1). Next, we outline our base genetic programming (GP) algorithm and how it synthesizes programs (2.2). Next, we briefly introduce refinement types and LiquidHaskell (2.3). Then, we present our new fitness function, describing how we integrate information from LiquidHaskell into the base GP algorithm (2.4).

2.1 Program Synthesis Model

In our program synthesis model, programs are composed of *program pieces*, which are functions provided by the user or imported from built-in and external libraries. As a running example, we consider a list filtering problem that we call `FilterEvens`: given a list of integers, return a list containing only the even integers from the input. The example below, and subsequent examples, will use Haskell syntax [5]. A user might provide the following 3 program pieces:

Example 1. Program pieces for `FilterEvens`

1.
   ```
   condition :: Int -> Bool
   condition x = x `mod` 2 == 0
   ```

2.
   ```
   condition :: Int -> Bool
   condition x = x `mod` 2 /= 0
   ```

3.
```
filterEvens :: [Int] -> [Int]
filterEvens xs = [a | a <- xs, condition a]
```

A correct program would consist of pieces 1 and 3. Note that piece 2 is not ultimately needed; a user will not have complete knowledge of the implementation, so they may include pieces that the synthesis algorithm chooses not to use.

2.2 Genetic Programming Algorithm

In the context of program synthesis, genetic programming evolves a population of candidate programs over time to find an optimal program [3]. Candidate programs are defined by their *chromosome*, a sequence of integers representing the indexes of the program pieces that compose that program. For example, using our `FilterEvens` problem defined in Example 1, the chromosome $c = [1, 3]$ corresponds to this program consisting of piece numbers 1 and 3:

Example 2. Program defined by chromosome $[1, 3]$

```
condition :: Int -> Bool
condition x = x `mod` 2 == 0

filterEvens :: [Int] -> [Int]
filterEvens xs = [a | a <- xs, condition a]
```

A sketch of our base genetic programming algorithm is shown in Algorithm 1. We provide a set of parameters Θ which includes the population size, chromosome length, mutation and crossover rate for variation, tournament size, and elite size, and parameter G, the number of generations to run for. We also provide a fitness function f, which computes a heuristic representing how "good" each candidate solution is, along with a set of input/output examples X which we use to test candidate programs to compute fitness (described in more detail below).

The algorithm proceeds in the following steps, labeled with the corresponding line numbers in Algorithm 1:

– **Generate individuals** (1): Let $|c|$ be the chromosome length and $|P|$ the number of program pieces; both are provided in the parameters. We generate a random individual by generating $|c|$ random numbers, each in the range $[0, |P|)$. This list represents the chromosome for that individual. We repeat the process `pop_size` times to generate an initial population.
– **Compute fitness** (2, 6): We use the provided fitness function f to compute fitness for each individual in the population.
– **Selection** (4): To select individuals for variation, we use tournament selection [13]. The tournament size t is provided in the parameters Θ. We will run `pop_size` tournaments, where each tournament selects t individuals at random from the population and selects the individual with best fitness. Thus, individuals with higher fitness are more likely to be selected for variation.

- **Variation** (5): We use two variation operators to create new individuals.
  - **Mutation** [14]: With probability equal to the mutation rate, we mutate an individual as follows. Given a chromosome c, we choose an index uniformly at random from $[0, |c|)$, and change it to a new value, also chosen uniformly at random from the range of possible values $[0, |P|)$, to get new chromosome c'.
  - **Single-Point Crossover** [15]: With probability equal to the crossover rate we create two new individuals as follows. Given two chromosomes c_1 and c_2, we choose an index uniformly at random to be the crossover point p. We create new individuals c_1' and c_2' such that c_1' contains the left part of c_1, up to index p, and the right part of c_2, from index $p+1$ to the end, and vice versa for c_2'.
- **Replacement** (7): We use an elitism strategy [16] to update the population. Let e be the elite size provided in the parameters Θ. We choose our new population to consist of the e individuals from the current generation before variation with the best fitness, plus the (`pop_size` $- e$) individuals after variation with the best fitness.

Algorithm 1 Genetic Programming for Program Synthesis
evolve(Θ, G, f, X):

```
1:  $P \leftarrow$ generate_individuals($\Theta$)           // Generate random initial population
2:  $P \leftarrow$ computeFitness($P, f(X, \cdot)$)          // Compute fitness of initial pop
3:  for $G$ iterations do
4:      $P' \leftarrow$ selection($P, \Theta$)               // Select individuals for variation
5:      $P' \leftarrow$ variation($P', \Theta$)              // Mutation and crossover
6:      $P' \leftarrow$ computeFitness($P', f(X, \cdot)$)    // Compute fitness of new pop
7:      $P \leftarrow$ replacement($P, P', \Theta$)          // Update population depending on fitness
8:  end for
9:  $p^* \leftarrow$ max({p.fitness : $p \in P$})
10: return p*                                               // Return program with max fitness
```

Fitness Function In our base algorithm, we use a standard fitness function: the candidate program's accuracy on the example test cases X [3]. In particular, given some chromosome c, fitness is given by

$$f_{IO}(X, c) = \frac{\text{number of correct examples}}{\text{total number of examples}}$$

Under this fitness function, programs which perform better on the example cases will have higher fitness. However, there are potential problems with using input-output examples, as mentioned in Section 1. This fitness function only specifies a program's intended behavior for a small set of examples, and a solution that succeeds on these examples may not necessarily generalize to others [17]. This leads us to explore refinement types as an alternate way to compute fitness.

2.3 Refinement Types and LiquidHaskell

Refinement types are types that further restrict the space of possible values by specifying a predicate. For example, we can express the `filterEvens` function from our running example using refinement types as follows, indicating that it takes a list of integers as input and outputs a list of *even* integers:

Example 3. LiquidHaskell Refinement Type Specification for `filterEvens`

```
{-@ type Even = {v:Int | v mod 2 = 0} @-}
{-@ filterEvens :: [Int] -> [Even] @-}
```

LiquidHaskell [19] is a plugin for Haskell which supports refinement types, including static checking of refinement type safety using a symbolic solver such as Z3 [20]. We can express a function like `filterEvens` in Example 3, and LiquidHaskell will verify at compile time that `filterEvens` satisfies the refined type. In this case LiquidHaskell checks that the output of `filterEvens` is always a list of even integers. If the check fails, LiquidHaskell outputs errors showing which refinement type specifications were not satisfied. This static checking is able to not only restrict integer values, but also enforce properties of lists and other complex types, so it is applicable to a broad range of functions.

2.4 Refinement Types Fitness Function

For certain types of problems, such as the `FilterEvens` example we have defined, refinement types are able to express the intended behavior of the program. Because this is a symbolic check, it verifies that behavior over all valid inputs without relying on example test cases.

To make use of this property, we leverage LiquidHaskell's refinement type checking to define a new fitness function for the GP. To do so, we require that the user provide a refinement for each program piece. Since refinements are based only on the intended behavior of a function, and do not depend on the implementation, we assume that users will be able to provide refinements even for library functions that will be used in the synthesized code.

A naive fitness function that simply runs the LiquidHaskell type check would return a binary value (0 if it fails, 1 if it passes), which does not work well as a heuristic. Instead, we can look more closely at LiquidHaskell's output, which includes syntax errors and refinement type errors, to construct a more fine-grained function.

Syntax Errors We assume that individual program pieces, which are often built-in functions or library functions, are free of syntax errors. Under this assumption, the only syntax errors that can be produced by combining program pieces are multiple definition errors (for pieces that have the same name and function signature), and missing definition errors (for pieces that were declared in other pieces but don't appear in the solution). The maximum number of syntax errors that can result is equal to the length of the chromosome.

Refinement Type Errors Refinement type checking is only performed after regular syntax checking, so no refinement type errors are reported if a program has incorrect syntax. Otherwise, if the program has no syntax errors, LiquidHaskell will report one error per refinement (i.e. per function signature) that isn't satisfied. Thus, the maximum possible number of refinement type errors is also equal to the length of the chromosome.

Fitness Function We construct our fitness function using a linear scale based on the number and type of errors reported. In addition, we follow the principle that syntax errors are generally "worse" than refinement type errors; syntax errors indicate structural issues like duplicated or missing program pieces, while refinement type errors mean that the program has the right structure.

Therefore, for a given chromosome c (with length $|c|$) where LiquidHaskell produces s syntax errors and t refinement type errors, we calculate the following fitness function:

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

From Equation 1, programs that have syntax errors always have fitness ≤ 0.5 while programs that have no syntax errors will have fitness ≥ 0.5. A program that has no syntax or refinement type errors is considered to be correct and has a fitness value of 1. We will use this fitness function with our original GP algorithm as described in Algorithm 1.

3 Experiments and Results

In this section we present an evaluation of our new fitness function based on refinement type checking. Our goal is to assess whether it can provide a performance and scalability improvement over two baselines: a standard fitness function based on input-output examples, and random search. In Section 3.1 we specify our benchmark problems and what program pieces we use in the synthesis. In Section 3.2 we describe our experimental setup. Next, in Section 3.3 we outline the results of our evaluation. Lastly, in Section 3.4 we discuss limitations of our technique and possible threats to its validity.

3.1 Program Synthesis Problems

We use a set of 3 program synthesis problems for evaluation. Some are adapted from a general program synthesis benchmark suite [6] and expanded for our program synthesis model as described below. All of them have the property that their behavior can be expressed using refinement types. Below are the problem specifications and a high level description of what program pieces are included.

1. **List Filtering** (adapted from Count Odds in [6]): Given a list of integers, filter the list and return 3 new lists containing just the even integers, just the odd integers, and just the integers greater than 2. We provide several possible filtering conditions as program pieces, including the correct ones as well as others that are not needed for the correct solution.
2. **Insertion Sort**: Given a list of integers, sort them in ascending order using insertion sort. We provide several possible conditions for determining when to insert, as well as a skeleton for the sort. The skeleton provides the control flow, so our search needs to find the correct conditions and operations to fit into the skeleton.
3. **QuickSort**: Given a list of integers, sort them in ascending order using quicksort. We provide a skeleton for the sort function, as in Insertion Sort, as well as different possible ways of partitioning the list for quicksort.

3.2 Experimental Setup

For each selected program synthesis problem, we run 60 trials and report performance as the number of generations taken to find a solution. We compare the following 3 variants of GP search:

1. **RefinementTypes (RT)**: GP search using our new fitness function based on counting errors from refinement type checking (Equation 1).
2. **IOExamples (IO)**: GP search using a baseline fitness function using accuracy on a set of input-output example cases, as described in Section 2.2. For each problem, we choose a small (< 10) but diverse set of examples. Specifically, we ensure that the example sets cover all execution paths in a correct solution.
3. **RandomSearch (RS)**: Random generation of individuals. To make this comparable with GP search, we proceed in generations, where `pop_size` individuals are randomly generated and evaluated per generation. As with GP search we can report the number of generations taken to find a solution. Thus, the total number of fitness evaluations is the same (`pop_size * generations`), so the running time is approximately equal as well. We include this as a baseline to verify that GP is well suited to our program synthesis model and provides an improvement over naive random search.

We also run each problem on 3 different search space sizes to evaluate scalability; we vary the size of the search space by including or excluding different optional program pieces which are not needed in a correct solution.

The common parameters that we used for all experiments is shown in Table 1. Note that for ease of implementation, we terminate searches after 20 generations and report a run as having taken 20 generations if it does not find a solution.

We tuned the max generations and population size to find a setting in which most trials find a solution before reaching the max generation limit. We did not tune the other parameters.

Parameter	**Value**
Mutation rate	0.3
Crossover rate	0.8
Tournament size	3
Elite size	2
Population size	20
Max generations	20
Number of trials	60

Table 1. Experiment parameters

Our implementation, including problem specifications and program piece specifications for each problem, is available on GitHub [1].

3.3 Results

Table 2 shows the results of our experiments. For each problem and set of program pieces, the search space size is calculated as $|P|^{|c|}$, where $|P|$ is the number of program pieces and $|c|$ is the length of the chromosome. We present the sample mean $\bar{x}$ and standard deviation s of the number of generations taken to find a solution for each fitness function.

The p-values shown in the table come from comparing the two specified variants using the Mann-Whitney U nonparametric test [21], which tests the null hypothesis that two sets of samples have the same population distribution (in particular, the probability that a random member from population 1 is greater than a random member from population 2 is 1/2). The p-values have also been adjusted for multiple hypothesis testing using the Bonferroni correction to decrease the likelihood of Type I error [22]; specifically, we multiply p-values by 2, the number of simultaneous hypotheses we are testing.

We can see from the table that in general, `RefinementTypes` finds a solution in fewer generations than the two baselines. Across all the experiments, `RefinementTypes` achieves an average improvement of 20% over `IOExamples` and 32% over `RandomSearch`. The p-values show that the improvement is significant ($p < 0.05$) in most cases.

We hypothesize that a key reason for the performance improvement is the difference in fitness values for programs that have syntax errors. For `IOExamples`, all programs that have syntax errors have a fitness value of 0 since the fitness evaluation is not able to run at all (the program cannot be interpreted). We can see in Figure 1a that for `IOExamples`, many candidate programs (all those with syntax errors) have fitness values of 0, and there are not many distinct fitness values. On the other hand, the `RefinementTypes` fitness function provides a heuristic even if there are syntax errors, as seen in Figure 1b, where there are

[1] https://anonymous.4open.science/r/GAble-BD28/

Problem	Search Space Size	Generations to find solution: Refinement Types $\bar{x}$	Refinement Types s	IO Examples $\bar{x}$	IO Examples s	Random Search $\bar{x}$	Random Search s	$p_{RT=IO}$	$p_{RT=RS}$
List Filtering	5.9e4	8.2	5.4	10.5	6.6	14.2	6.8	0.065	**0.000**
	1.0e5	12.5	6.2	14.8	5.9	15.9	6.0	**0.046**	**0.002**
	1.0e6	12.8	6.6	16.7	5.0	17.8	4.6	**0.000**	**0.000**
Insertion Sort	1e5	5.4	4.8	8.4	7.1	8.4	6.5	**0.042**	**0.010**
	1.6e6	8.0	5.7	8.9	6.9	11.4	6.9	0.700	**0.008**
	1.8e7	9.4	7.0	13.1	6.7	16.4	5.1	**0.008**	**0.000**
QuickSort	2.6e5	9.3	6.1	11.1	7.3	12.7	7.0	0.181	**0.005**
	5.3e5	10.3	6.2	14.6	6.3	15.8	6.5	**0.000**	**0.000**
	1.0e6	9.0	6.2	17.2	4.8	15.5	6.5	**0.000**	**0.000**

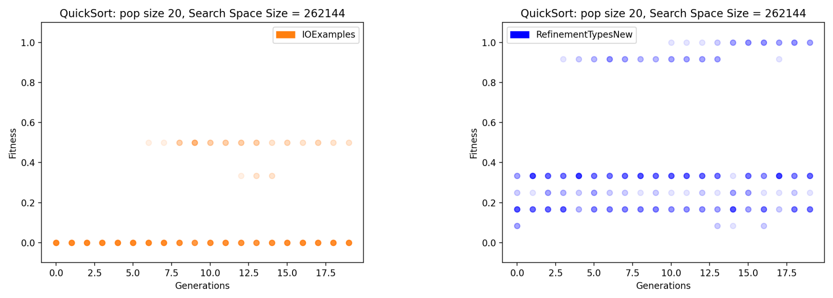
Table 2. Experiment Results. We run 60 trials per problem, search space size, and variant and record the number of generations taken to find a solution. We report the sample mean $\bar{x}$ and standard deviation s. The p-values come from the Mann-Whitney U nonparametric test and have been adjusted using the Bonferroni correction for multiple hypothesis testing. p-values less than 0.05 are in bold.

four distinct fitness values for programs with syntax errors (fitness < 0.5). This is helpful because among programs that have syntax errors, some are still closer to correct (e.g. less errors) and the `RefinementTypes` fitness function can capture that. Therefore, in areas of the search space corresponding to programs that have syntax errors, the new fitness function can still guide the GP search whereas those programs are all evaluated to be equally "unfit" by the `IOExamples` fitness function. In the trial shown in Fig. 1, the search using `IOExamples` is unable to find a solution after 20 generations, whereas the additional heuristic information provided by `RefinementTypes` allows the GP search to find a solution after 10 generations.

We also see from the table that the p-value generally remains below 0.05 as the search space size increases, which shows that the performance improvements that we observe can potentially scale to larger problems as well.

3.4 Threats to Validity

We note that refinement types are not applicable to every problem; for example, some string manipulations, such as the Double Letters problem from [6], would be difficult to express using refinement types since they involve complex dependencies between indices of the string. In addition, we observed that the GP search overall runs an order of magnitude slower in terms of wall-clock time when using the refinement type check rather than example cases as a fitness function. We did not optimize our implementations; in particular, there are many I/O op-



(a) `IOExamples` (b) `RefinementTypes`

Fig. 1. Scatter plots of population's fitness values over time (generations) for (a) `IOExamples` and (b) `RefinementTypes` fitness functions. Each plots was generated from one trial run on the QuickSort problem with the same search space size and population size. Each point (g, f) represents an individual in generation g with a fitness value of f, and the opacity increases with the number of individuals with fitness value f.

erations that may be unnecessary in a better implementation, so this difference may change after optimization.

4 Related Work

Since the fitness function is so integral in GP search, many researchers have studied different ways of defining the fitness function. NetSyn [12], mentioned in Section 1, uses a neural network to learn a better fitness function based on input-output examples. CrowdBoost [23] explores evolving the fitness function along with candidate programs during GP. Hemberg et al. [24] show that it is possible to improve search performance by using domain knowledge extracted from the problem description to construct the fitness function. Others have investigated using formal methods like model checking and Hoare logic for program verification as the basis for the fitness function in GP [25, 26]. Our approach similarly uses formal methods for the GP's fitness function, but we use refinement types; with LiquidHaskell, it is very easy to define and verify refinement types for program pieces [27].

Prior works have also explored refinement types and their applicability to program synthesis. Synquid [28] uses refinement types for program synthesis without GP by decomposing the type specifications and solving local type constraints. Fonseca et al. [29] suggest an approach for combining GP with refinement types, including a possible fitness function for refinements expressed in their programming language; however, they do not present any experimental data or results.

A similar approach for improving practicality of program synthesis is program *sketching*, where a user provides a partially-complete template of a program, and the synthesis algorithm fills in the missing low-level details [30]. This has been implemented successfully for certain problem domains in systems like Sketch [31] and Psketch [32], which achieve better efficiency because the search space is restricted. Our approach is analogous but inverted: the user provides building blocks and the synthesis algorithm finds a correct composition of those building blocks. This has the same benefit of restricting the search space and can be useful in situations where a user does not have enough knowledge of the program structure to build a sketch.

5 Conclusions

Our results show that it is possible to express complex programs such as sorting using our program piece-based model. Using this model for program synthesis, we can make use of refinement type checking to express correctness properties of the program. We show that 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 approaches to achieving scalability in program synthesis as well as different ways of incorporating symbolic solving within GP search. In future work, we hope to evaluate a wider set of benchmarks, including more complex problems with larger search spaces.

References

1. Gulwani, S., Polozov, O., and Singh, R.: Program synthesis. In Foundations and Trends in Programming Languages, 4(1-2):1–119, 2017.
2. Yampolskiy, R.V.: AI-Complete, AI-Hard, or AI-Easy – Classification of Problems in AI. In The 23rd Midwest Artificial Intelligence and Cognitive Science Conference. Cincinnati, OH, USA. 2012.
3. Koza, J.R.: Survey of Genetic Algorithms and Genetic Programming. In Wescon 95: E2. Neural-Fuzzy Technologies and Its Applications, pp. 589-594. IEEE, San Francisco. 1995.
4. David, C. and Kroening, D.: Program Synthesis: Challenges and Opportunities. In Philosophical Transactions of the Royal Society A: Mathematical Physical and Engineering Sciences, vol. 375, no. 2104. 2017.
5. Hudak P., Jones S. P., and Wadler P.: Report on the programming language Haskell: a non-strict, purely functional language, version 1.2. In ACM SIGPLAN Notices, vol. 27, no. 5. ACM, New York. 1992.
6. Helmuth, T. and Spector, L.: General program synthesis benchmark suite. In Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation Conference, GECCO 2015, pp. 1039–1046. ACM, Madrid. 2015.
7. Giacobini M., Tomassini M., Vanneschi L.: Limiting the Number of Fitness Cases in Genetic Programming Using Statistics. In Guervós J.J.M., Adamidis P., Beyer HG., Schwefel HP., Fernández-Villacañas JL. (eds) Parallel Problem Solving from Nature — PPSN VII. PPSN 2002. Lecture Notes in Computer Science, vol 2439. Springer, Berlin, Heidelberg. 2002.

8. O'Neill, M., Vanneschi, L., Gustafson, S., and Banzhaf, W.: Open issues in genetic programming. In Genetic Programming and Evolvable Machines, pp. 339–363. 2010.
9. Poli, R., Langdon, W.B., and McPhee, N.F.: A Field Guide to Genetic Programming. Lulu Enterprises, UK. 2008.
10. Vazou, N., Rondon, P.M., and Jhala, R.: Abstract refinement types. In Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 209–228. Springer, Heidelberg, 2013.
11. Rondon, P.M., Kawaguchi, M., and Jhala, R.: Liquid types. In Programming Language Design and Implementation (PLDI), pp. 159–169. 2008.
12. Mandal, S., Anderson, T.A., Gottschlich, J., Zhou, S., Muzahid, A.: Learning fitness functions for genetic algorithms. arXiv Preprint, pp. arXiv:1908.08783. 2019.
13. Fang, Y., Li, J.: A review of tournament selection in genetic programming. In Cai, Z., Hu, C., Kang, Z., Liu, Y. (eds.) ISICA 2010. LNCS, vol. 6382, pp. 181–192. Springer, Heidelberg. 2010.
14. Page J., Poli R., Langdon W.B.: Mutation in Genetic Programming: A Preliminary Study. In Poli R., Nordin P., Langdon W.B., Fogarty T.C. (eds) Genetic Programming. EuroGP 1999. Lecture Notes in Computer Science, vol 1598. Springer, Berlin, Heidelberg. 1999.
15. Poli R., Langdon W.B.: Genetic Programming with One-Point Crossover. In Chawdhry P.K., Roy R., Pant R.K. (eds.) Soft Computing in Engineering Design and Manufacturing. Springer, London. 1998.
16. Poli, R., McPhee, N.F., Vanneschi, L.: Elitism reduces bloat in genetic programming. In Keijzer, M. et al. (eds.) Proceedings of GECCO-2008, pp. 1343–1344. ACM Press, New York, NY. 2008.
17. Kitzelmann E.: Inductive Programming: A Survey of Program Synthesis Techniques. In Schmid U., Kitzelmann E., Plasmeijer R. (eds) Approaches and Applications of Inductive Programming. AAIP 2009. Lecture Notes in Computer Science, vol 5812. Springer, Berlin, Heidelberg. 2010.
18. Hillis, D: Co-evolving parasites improve simulated evolution as an optimizing procedure. In Physica D, vol. 42, pp. 228–234. 1990.
19. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for Haskell. In ACM SIGPLAN Notices. vol. 49, pp. 269–282. 2014.
20. de Moura, L., Bjãrner, N.: Z3: An efficient SMT solver. 2008.
21. Mann, H.B., Whitney, D. R.: On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. Annals of Mathematical Statistics. vol. 18, no. 1, pp. 50–60. 1947.
22. Bland, J.M., Altman, D.G.: Multiple significance tests: the Bonferroni method. BMJ, vol. 310, no. 6973, pp. 170. 1995.
23. Cochran, R., Livshits, B., Molnar, D., Veanes, M., and D'Antoni, L.: Program Boosting: Program Synthesis via Crowd-Sourcing. In ACM SIGPLAN Notices. 2015.
24. Hemberg, E., Kelly, J., and O'Reilly, U.M.: On domain knowledge and novelty to improve program synthesis performance with grammatical evolution. In Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, pp. 1039–1046. ACM, Prague, Czech Republic. 2019.
25. Johnson, C.: Genetic programming with fitness based on model checking. In Proceedings of the 10th European Conference on Genetic Programming, pp. 114–124. Lecture Notes in Computer Science, Vol. 4445. 2007.
26. He, P., Kang, L., Johnson, C. G., and Ying, S.: Hoare logic-based genetic programming. SCIENCE CHINA Information Sciences, 54(3):623–637. 2011.
27. Vazou, N., Seidel, E., and Jhala, R.: LiquidHaskell: Experience with Refinement Types in the Real World. ACM SIGPLAN Notices, vol. 49. 2014.

28. Polikarpova, N., Kuraj, I., and Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In Programming Language Design and Implementation (PLDI), pp. 522–538. 2016.
29. Fonseca A., Santos P., and Silva S.: The Usability Argument for Refinement Typed Genetic Programming. In Bäck T. et al. (eds.) Parallel Problem Solving from Nature – PPSN XVI. PPSN 2020. LNCS, vol. 12270. Springer, Cham. 2020.
30. Solar-Lezama, A.: Program synthesis by sketching. PhD thesis, Berkeley, CA, USA. 2008.
31. Solar-Lezama, A., Tancau, L., Bodik, R., Saraswat, V., and Seshia, S.: Combinatorial sketching for finite programs. In ASPLOS '06, San Jose, CA, USA. 2006.
32. Solar-Lezama, A., Jones, C. G., and Bodík, R.: Sketching concurrent data structures. In Programming Language Design and Implementation (PLDI), pp. 136-148. 2008.