Extending Program Synthesis Grammars for Grammar-Guided Genetic Programming

Stefan Forstenlechner(✉), David Fagan, Miguel Nicolau, and Michael O'Neill

Natural Computing Research and Applications Group, School of Business, University College Dublin, Dublin, Ireland
stefan.forstenlechner@ucdconnect.ie,
{david.fagan,miguel.nicolau,m.oneill}@ucd.ie

Abstract. Program synthesis is a problem domain that due to its importance is tackled by many different fields, one being Genetic Programming. Two variants, Grammar-Guided Genetic Programming (G3P) and PushGP, have been applied to a vast general program synthesis benchmark suite and solved a variety of problems although with varying success rates. While G3P achieved higher success rates on some problems, PushGP was able to find solutions to more problem instances. Reason why G3P fails at some problems might be missing functionality in the grammars or knowledge that has to discovered during the runs. In this paper the current shortcomings of G3P are analysed and the papers contributions include an example of extending grammars for program synthesis, a fairer comparison between PushGP and G3P with a more similar function set as well as new results on problems that have not been solved with G3P and one that has not been solved with PushGP.

Keywords: Genetic Programming · Grammar · Program synthesis

1 Introduction

Genetic Programming has shown potential to solve a range of general program synthesis problems. In contrast to other problem domains like regression where an approximation of the solution might be acceptable, a partially correct solution is usually of no use in program synthesis. But for GP to be successful in program synthesis, the ability to find a correct solution should be high, as practitioners should not have to be required to run GP multiple times while researchers only do multiple runs for statistical tests. At the same time, it is essential that GP can solve a wide range of program synthesis problems rather than special cases.

To this end, a range of difficult or unsolved problems is identified in the general program synthesis benchmark suite [8], that has been used recently to test GP on program synthesis, especially with G3P [3] and PushGP [17]. While G3P was able to achieve a higher percentage of successful solutions found in

© Springer Nature Switzerland AG 2018
A. Auger et al. (Eds.): PPSN 2018, LNCS 11101, pp. 197–208, 2018.
https://doi.org/10.1007/978-3-319-99253-2_16

cases it found solutions, PushGP was able to solve more problems at least once in general.

The focus of this paper lies in identifying differences between the function set of G3P and PushGP, extending the grammars according to those differences as well as the identified difficult problems from the benchmark suite and extending the grammars accordingly. At the same time, the grammars shall stay as general as possible to be able to use them outside of the context of benchmark problems and should not be trimmed to "cheat" on any particular problem within the benchmark suite. As the benchmark suite that has been used so far, proposes to have an explicit *char* data type which is currently missing in G3P [3] the possibility of adding it is further investigated. Therefore, the functionality available in the grammars is not allowed to be extended further than the function set available to PushGP.

The rest of the paper is structured in the following way. Section 2 summaries related work on program synthesis. Section 3 describes the benchmark suite used in the GP community for program synthesis and what problems have been difficult for GP and particularly for G3P. Afterwards, Sect. 4 describes in what ways grammars can be extended to overcome the previous shortcomings. The experimental setup used to tackle the benchmark suite is described in Sect. 5 and the results are compared to previous approaches in Sect. 6. Finally, conclusion and future work are discussed in Sect. 7.

2 Related Work

Program synthesis problems have been tackled even before GP was used and many different approaches exist [11]. Nevertheless, GP systems have proven to be very flexible and successful at doing this. Therefore this paper will focus on GP systems.

2.1 Grammar-Guided Genetic Programming

Grammar-Guided Genetic Programming [12] is a GP variant that uses grammars to define the search space. This makes it easy to use and flexible as a grammar can be defined outside of the GP system instead of restricting GP to a certain function set. Additionally, it is quite powerful, because any program that can be generated with the grammar can be found by GP. Grammars also provide the possibility of adding bias, if necessary. The most famous variants are CFG-GP by Whigham [19] and grammatical evolution [14].

Forstenlechner et al. [3] proposed a grammar design for GP to tackle general program synthesis problems, as mainly bespoken grammars have been used before to solve program synthesis [13], which can not be reused to solve other problems. The idea of the grammar design is to have multiple smaller grammars and every grammar contains only the functionality for a single data type. Additionally, one general grammar exists which contains the structure of the program. The benefit of this design is that it is not limited to a single programming

language and depending on the problem at hand a subset of the data types required to solve the problem can be chosen. Therefore, the design is capable of solving general purpose program synthesis problems, while the search space can be kept small by not including unnecessary data types. Functions that require multiple data types of which some are not available, will be removed from the grammar automatically when combining the grammars for a chosen problem.

3 General Program Synthesis Benchmark Suite Remarks

A general program synthesis benchmark suite was introduced by Helmuth and Spector [8]. It provides a variety of problems from introductory computer science courses. It consists of a total of 29 problems with a description, training and test set, fitness function and general parameter settings, mainly for PushGP [17], for every problem. Additionally, every problem requires specific data types to be available to be solved. A more detailed description is available in form of a technical report [18], which also contains information about how to generate the training and test data as well as the instructions available for PushGP.

The two GP systems that have been tested on the benchmark suite are a G3P by Forstenlechner [3] and PushGP [17]. PushGP is a GP system that evolves programs in the language Push, which was solely designed for evolutionary algorithms. Push uses stacks to store data instead of using variables. It has a stack for every data type as well as for the code that is executed, which makes it possible to manipulate the code during runtime.

An additional comparison of systems outside of the GP community, namely Flash Fill [5] and MagicHaskeller [9], was done on the benchmark suite in [15]. The comparison showed that GP systems are more flexible and more successful on this benchmark suite, although it should be mentioned that these systems have been created with other use cases in mind like Flash Fill is used in Microsoft Excel for string manipulation tasks.

In the initial introduction of the grammar design for program synthesis problems [3], the functionality was kept to the basics of Python without including more than was available in PushGP. For example, adding the built-in *sum* function from Python would make solving the problem Vector Average fairly easy.

Table 1 shows the results achieved with G3P on the general program synthesis benchmark suite. The results have been taken from [3]. The datasets of Checksum and Vector Average have been changed since the benchmark suite has been introduced and a simpler version of Super Anagrams has been used in [3]. The table indicates that G3P with the current grammars has difficulty to solve problems that require *char* as a data type. At the moment it only uses *string*, most likely because the initial grammars are based on Python which treats char as string. While a programmer has no difficulty to understand how or when to use a single character string, it is definitely more complicated for GP to find out how or when to use it. Adding a *char* data type could yield better results. Additionally, PushGP was able to solve more problems from the benchmark suite, although in many cases with a low success rate. Nevertheless, adding further functionality could help improve the results of G3P.

Table 1. Results of G3P on the general program synthesis benchmark suite sorted by successfully found solutions. *String* and *Char* column indicate if these data types have to be used when solving the problem. A * indicates if the data set has been changed, since the results have been acquired.

	NumberIO	*Smallest*	*Vectors Summed*	*Median*	*String Lengths Backwards*	*Negative To Zero*	*Grade*	*Last Index of Zero*	*Super Anagrams**	*Count Odds*	*For Loop Index*	*Small Or Large*	*Vector Average**	*Sum of Squares*	*Compare String Lengths*	*Scrabble Score*	*Even Square*	*Checksum**	*Collatz Number*	*Digits*	*Double Letters*	*Mirror Image*	*Pig Latin*	*Replace Space with Newline*	*Syllables*	*Wallis Pi*	*Word Stat*	*X-Word Lines*
Successes	94	94	91	79	68	63	31	22	21	12	8	7	5	3	2	2	1	0	0	0	0	0	0	0	0	0	0	0
String					X		X		X			X			X	X		X		X	X		X	X	X		X	X
Char									X							X		X		X	X		X	X	X		X	X

4 Extending Program Synthesis Grammars

This section describes how the program synthesis grammars from [3] have been extended to include an additional *char* data type as well as additional functionality to have a fairer comparison to PushGP. Extending the grammar also means increasing the size of the search space as more programs can be generated from the grammar. Therefore, the extension of the grammars can also have a negative effect on the search performance.

4.1 Data Type Char

As shown in Sect. 3, G3P does poorly on problems that require a data type char. G3P only used string as it mainly relied on Python even though the concepts can be applied to other languages as well and because a char can be interpreted as a string of length one. As many problems in the general program synthesis benchmark suite require to check or manipulate single characters, G3P not using a *char* grammar could explain why it currently fails at solving such problems. While programmers have the intrinsic knowledge that a string consists of characters and a string of length one can be treated similar to a char, GP either has to discover this knowledge or has to be told a priori. The currently available grammar data types are *bool*, *integer*, *float* and *string*, as well as a list version grammar of each of these data types, plus the new *char* grammar. A list of char grammar is currently not included as the benchmark suite does not require it and strings can be viewed as a list of char. As G3P adds variables of the data types of every used grammar to the evolved program, including the char grammar makes it very likely that chars are used as opposed to before where G3P had to find that a string of length one is required.

4.2 Recursion

Recursion is a method of programming where a program calls itself to solve a smaller instance of the same problem first and uses that solution to solve the initial problem. Recursion is not an uncommon strategy to tackle problems in GP [1,20]. In many cases, a recursive solution can be significantly shorter in terms of code than an iterative program, which might make it easier for GP to find. PushGP is capable of evolving recursive programs and for a fair comparison should be part of the grammars for G3P as well.

To allow recursion, a program needs to be able to call itself and a way to stop the recursion, usually an *if* condition called guard. As the grammars in G3P are automatically merged together depending on the required data types, and the number of input/output variables, as well as there types, a rule for a recursive call can be generated and added to the grammar. The following is an example where `outputX` is replaced with the correct type variable non-terminal (e.g. `<bool_var>`) and `inputX` with the correct type (e.g. `<bool>`):

```
<output1>', '...', '<outputN>' = evolve('<input1>', '...', '<inputN>')'
```

In a similar way, a return statement can be generated:

```
'return result1, ..., resultN'
```

The grammar used to define the control flow (*structure.bnf*) already contains *if* statements, but it is very likely that it might not be used and the program gets stuck in an infinite recursion and at some point will throw an error due to a stack overflow. A problem that occurs with infinite loops as well and was handled by adding a guard to avoid any additional iterations if a certain limit is reached. A similar guard is used to avoid infinite recursion. The benefit of using this mechanism is that evolved programs will not throw an error and return a value. Therefore, the program will be given a fitness value based on what it returns instead of a default worst case fitness due to an error.

4.3 List Operations

When the grammars for program synthesis were introduced grammars for lists of all data types were included but kept to the essential functionality. Items could be added at the end, inserted or replaced at a specific index or removed. Lists could be iterated, compared, checked if they are empty and their length could be determined as well as slicing of lists was possible. Any additional functionality the algorithm had to find. PushGP offers more functionality out of the box that can be used, which has been added to the grammars for G3P, like reversing a list, counting the occurrences of an item, replacing or removing items if a condition is met etc. All of this functionality could be discovered as well, but as for example O'Neill et al. [13] showed that GP has difficulties finding a solution to the integer sorting problem, but by adding a swap function the problem was easily solvable. As stated before no further functionality has been added, that was not already available for PushGP as well. At the same time, it should be noted that adding

additional functionality also increases the search space, which can make it more difficult to find a correct solution. Even though the additional functionality can make it easier to solve one problem, it can make it more difficult to solve another. Therefore a decrease of successful solutions found on some problems is to be expected.

4.4 Additional Methods

Similar to the list operations in the previous section, additional methods were added to other data types that in general could have been discovered by G3P. One example that is also often not included for boolean problems is *XOR*, as it can be constructed with *AND*, *OR* and *NOT* and can make certain problems like multiplexer too easy [10]. To be able to have a better comparison between G3P and PushGP, such methods have been added as well. As there are too many to mention every single one of them, the reader is referred to the grammars themselves that are provided online [2] as well as [18]. Again, it should be noted that the extended grammars do not exceed the functionality that is provided by PushGP.

5 Experimental Setup

For the experiments, the extended grammars, which are described in the previous section, are used with the same G3P system as in [3], which is available online [2] including the extended grammars. The experiments are run on the problems from the general program synthesis benchmark suite [8]. The parameter settings are summarized in Table 2. The number of generations is set to 300[1]. As soon as a successful solution is found, the run is stopped as GP cannot improve it anymore. Lexicase selection [6] is used, as it has shown to be the most successful selection operator with GP on program synthesis problems. Instead of using a single fitness value for selection, lexicase operates on the fitness values of every single training cases. It randomly selects a fitness case and selects the best individual based on that case. In case of a tie, lexicase selection continues with a subset of individuals that were in this tie and continue to select other training cases until a single individual is left or until no fitness case is left, in which case an individual is selected randomly.

6 Results

First the overall performance of G3P with the extended grammars and also to PushGP. Afterwards, the effect of the extended grammars on the search is analysed in more detail.

[1] 200 for Normal IO, Median and Smallest as proposed in [8].

Table 2. Experimental parameter settings

Parameter	Setting
Runs	100
Generations	300 (see footnote 1)
Population size	1000
Selection	Lexicase
Crossover probability	0.9
Mutation probability	0.05
Elite size	1
Node limit	250
Variables per type	3
Max execution time	1 s
Max_Tries	10

6.1 Successful Solutions

Table 3 shows the solutions found for each problem with G3P with extended grammars for training and test with 100 runs. The results are compared to the previously achieved successful solutions of G3P from [3]. Of the eight problems that require a *char* data type and have not been solved with G3P before, three have been solved with the extended grammars, namely Pig Latin, Replace Space with Newline and Syllables. Pig Latin is one that has not been solved with PushGP either. Additionally, Mirror Image has been solved as well, probably due to the additional list operations, which was not solved with the G3P with previous grammars. Table 3 also includes the p-value for the Wilcoxon Rank sum test on best test fitness of the two grammar approaches and shows a significant difference for nearly all of the problems. This is not surprising as the grammar has a massive influence on the search, as a function set has on normal GP.

The results also show that due to the increased search space, which is caused by the additional functions added to the grammar, the number of successful solutions decreases for some problems. Three problems, Compare String Lengths, Even Squares and Vector Average, could not be solved anymore, but the success rate of the first two was rather small before as well. Especially, Compare String Lengths is highly overfit as 96 successful solutions were found on test, but none generalizes on test. This is a problem that occurs on multiple problem instances and has been noticed before [7].

Even though on the final experiments some problems, even those which require char as data type are still not solved, in preliminary experiments Checksum and Double Letters have been solved with G3P with extended grammars as well. Even then the success rate was rather small, but theoretically, it has been found that they can be solved with the extended grammars as well.

Table 3. Successful solutions found with G3P with extended grammars on training and test with 100 runs as well as increase and decrease to the previous grammars in brackets. The p-value shows if there is a significant difference in the best test performance between the two different grammars with 0.05 as level of significance. A significant difference is highlighted in bold. Finally, the results of PushGP on the benchmark suite from [8] and the difference to G3P with extended grammars in brackets are compared.

	G3P			PushGP
Problem Name	Test	Training	p-value	Test
Checksum	0 (+0)	0 (+0)	**8.74E-32**	0 (+0)
Collatz Numbers	0 (+0)	0 (+0)	0.0991	0 (+0)
Compare String Lengths	0 (–2)	96 (–1)	**8.06E-05**	7 (+7)
Count Odds	3 (–9)	4 (–8)	**1.81E-15**	8 (+5)
Digits	0 (+0)	0 (+0)	**0.0298**	7 (+7)
Double Letters	0 (+0)	0 (+0)	**0.0040**	6 (+6)
Even Squares	0 (–1)	0 (–1)	0.5683	2 (+2)
For Loop Index	6 (–2)	9 (–11)	**0.0006**	1 (–5)
Grade	31 (+0)	63 (–18)	0.1005	4 (–27)
Last Index of Zero	44 (+22)	97 (+43)	**5.71E-11**	21 (–23)
Median	59 (–20)	99 (–1)	**0.0039**	45 (–14)
Mirror Image	25 (+25)	89 (+38)	**3.25E-18**	78 (+53)
Negative To Zero	13 (–50)	24 (–42)	**9.12E-07**	45 (+32)
Number IO	83 (–11)	95 (–5)	**1.21E-15**	98 (+15)
Pig Latin	3 (+3)	4 (+4)	**4.02E-25**	0 (–3)
Replace Space with Newline	16 (+16)	29 (+29)	**5.08E-30**	51 (+35)
Scrabble Score	1 (–1)	1 (–4)	**0.0008**	2 (+1)
Small Or Large	9 (+2)	39 (–12)	0.5493	5 (–4)
Smallest	73 (–21)	100 (+0)	**9.58E-05**	81 (+8)
String Lengths Backwards	18 (–50)	20 (–48)	**6.70E-17**	66 (+48)
Sum of Squares	5 (+2)	5 (+2)	**8.02E-05**	6 (+1)
Super Anagrams	0 (+0)	43 (–1)	**2.33E-34**	0 (+0)
Syllables	39 (+39)	53 (+53)	**4.28E-29**	18 (–21)
Vector Average	0 (–16)	0 (–17)	**6.94E-32**	16 (+16)
Vectors Summed	21 (–70)	28 (–65)	**1.84E-23**	1 (–20)
Wallis Pi	0 (+0)	0 (+0)	**3.03E-24**	0 (+0)
Word Stats	0 (+0)	0 (+0)	0.7722	0 (+0)
X-Word Lines	0 (+0)	0 (+0)	**2.56E-34**	8 (+8)

Finally, Table 3 shows the results of PushGP taken from [8] compared to G3P with extended grammars. According to [7], PushGP is able to solve Checksum after the original dataset has been changed. The comparison shows that both approaches have problems where one method is more capable to find solutions than the other, but there does not seem to be a clear advantage over one or the other. Some problems have been solved with PushGP that have currently not been solved with G3P, but again the success rates of these problems are very small, below 10, in most cases, which makes a comparison difficult. The low success rate is an issue that needs to be addressed by both approaches.

6.2 *Char* Analysis

The grammar for the *char* data type is used by 10 problems. The grammar contains a rule `<char>` with productions for char variables, char constants and all functions that return a char value. Therefore, checking the percentage of nodes in individuals shows if GP is making use of the additional data type. Figure 1 depicts this usage.

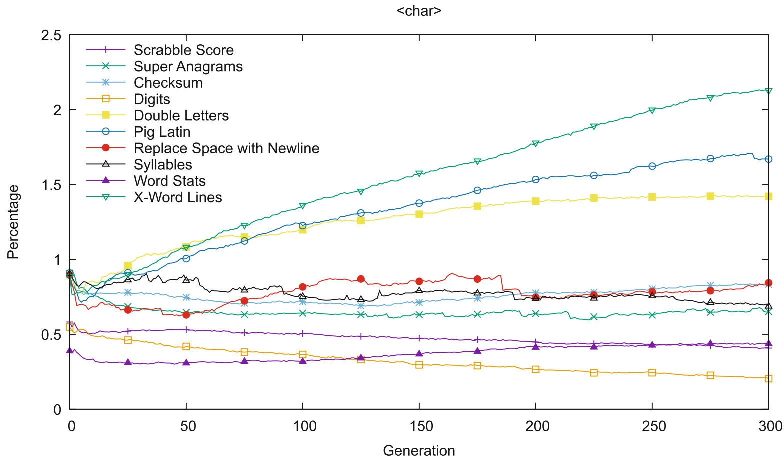

Fig. 1. Percentage of `<char>` nodes in individuals averaged over 100 runs over generations.

In the initial generation, the percentage of nodes being `<char>` is nearly identical for some problems, which is expected as these problems require the same data types, which means the grammars are nearly identical, except maybe input and output variables. Therefore the grammar has the same structure and the same number of possible nodes, which leads to this effect. The percentage of `<char>` nodes used may seem small being between 0.5% and 1.5%, but considering the number of productions available in the grammar, it is rather high. In case of almost all problems, the usage of `<char>` nodes is either constant or increases over time, after a few generations. The only problem that seems to slowly decrease the usage of `<char>` nodes is Digits. This can be explained by how G3P is tackling the problems. While PushGP prints every integer for Digits, G3P has to return a list of integers as it does not use print statements and therefore does not necessarily need a char data type.

For some problems, especially Replace Space with Newline, Syllables, Super Anagrams and Pig Latin, the lines are not as stable as for the other problems. The reason is that solutions that solve the problem at least for training have been found and runs are stopped as soon as this happens. Hence, the average percentage might drop or increase. In most cases, a sudden drop has been found, which shows that runs that use `<char>` nodes more often seem to be able to find a successful solution earlier. This indicates that the char grammar improves the search for successful solutions.

6.3 Recursion Analysis

The percentage of recursion used can be checked in a similar way as in the previous section for char. Figure 2 depicts the percentage of recursion nodes used over generations. The initial percentage is lower than with `<char>`, because there is only one recursion production rule in the grammar, whereas `<char>` is used by multiple functions. Afterwards, it drops even lower for all problems and is barely used overall. As explained in Sect. 4.2, to use recursion, a method needs to be able to call itself and a stopping criterion. At the moment the GP system can evolve a method to call itself, but at the same time has to evolve a stopping criterion, which seems to make it too complicated to be used. Without the stopping criterion, the evolved program runs into an infinite loop, which leads to a stack overflow or a timeout by the G3P system. A way to improve this might be to adapt the grammar that a stopping criterion is added to the same production rule as the recursion to always have both added at the same time. This could increase the chance to make G3P use recursion to solve problems.

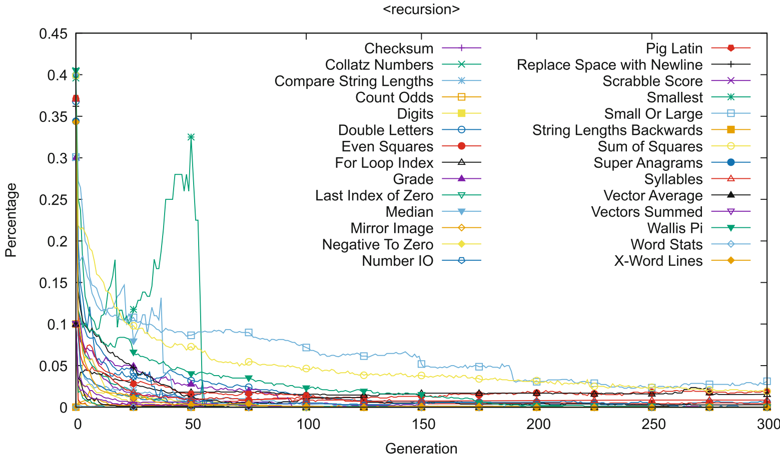

Fig. 2. Percentage of recursion nodes in individuals averaged over 100 runs over generations.

7 Conclusion and Future Work

The difficulties of solving multiple problems of the general program synthesis benchmark suite with a grammar design approach [3] have been discussed. As some of this problems have been solved with another approach before, the functionality of the grammars has been extended in various ways to be closer to previous approaches, without "cheating" by adding functionality not used before. An important enhancement of the grammars is that an explicit *char* grammar has been added as many problems operate on single characters instead of strings.

Programmers are able to identify such characteristics of a problem easily, while GP would have to discover such knowledge. As the benchmark suite proposes to use *char* as its own data type, this additional information does not give G3P an unfair advantage when comparing to other systems.

Afterwards, the extended grammars are used to tackle the program synthesis benchmark suite and the results are compared to the grammar design of [3]. The results show significant differences for nearly all problems and successful solutions have been found for previously unsolved problems with G3P. One problem, Pig Latin, has been successfully solved that was not solved by any other approach before. Additionally, a comparison with PushGP has been made, as the extended grammars are closer in functionality to PushGP as in [3].

Due to the increased search space created by the extended grammars, a decrease of successful solutions found on previously solved problems was expected. A way to dynamically adjust the functionality of grammars during runs could help avoid this problem [16]. Even the success rates of newly solved problems were rather low. This is a problem not only of G3P, but also of other approaches, and should be addressed in the future to make program synthesis with GP more usable outside of the research community as well. Current approaches include smarter operators [4] or post-run simplifications [7], but further research is required to increase success rates.

Acknowledgments. This research is based upon works supported by the Science Foundation Ireland, under Grant No. 13/IA/1850.

References

1. Agapitos, A., Lucas, S.M.: Learning recursive functions with object oriented genetic programming. In: Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A. (eds.) EuroGP 2006. LNCS, vol. 3905, pp. 166–177. Springer, Heidelberg (2006). https://doi.org/10.1007/11729976_15
2. Forstenlechner, S.: Github repository: HeuristicLab.CFGGP: Provides context free grammar problems for HeuristicLab (2016). https://github.com/t-h-e/HeuristicLab.CFGGP. Accessed 22 Mar 2018
3. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: A grammar design pattern for arbitrary program synthesis problems in genetic programming. In: McDermott, J., Castelli, M., Sekanina, L., Haasdijk, E., García-Sánchez, P. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 262–277. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55696-3_17
4. Forstenlechner, S., Fagan, D., Nicolau, M., O'Neill, M.: Semantics-based crossover for program synthesis in genetic programming. In: Lutton, E., Legrand, P., Parrend, P., Monmarché, N., Schoenauer, M. (eds.) EA 2017. LNCS, vol. 10764, pp. 58–71. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78133-4_5
5. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, pp. 317–330. ACM, New York (2011)
6. Helmuth, T., Spector, L., Matheson, J.: Solving uncompromising problems with lexicase selection. IEEE Trans. Evol. Comput. **19**(5), 630–643 (2015)

7. Helmuth, T., McPhee, N.F., Pantridge, E., Spector, L.: Improving generalization of evolved programs through automatic simplification. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, pp. 937–944. ACM, Berlin, 15–19 July 2017
8. Helmuth, T., Spector, L.: General program synthesis benchmark suite. In: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, GECCO 15, pp. 1039–1046. ACM, Madrid, 11–15 July 2015
9. Katayama, S.: Recent improvements of MagicHaskeller. In: Schmid, U., Kitzelmann, E., Plasmeijer, R. (eds.) AAIP 2009. LNCS, vol. 5812, pp. 174–193. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11931-6_9
10. Keijzer, M., Ryan, C., Murphy, G., Cattolico, M.: Undirected training of run transferable libraries. In: Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J., Tomassini, M. (eds.) EuroGP 2005. LNCS, vol. 3447, pp. 361–370. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31989-4_33
11. Kitzelmann, E.: Inductive programming: a survey of program synthesis techniques. In: Schmid, U., Kitzelmann, E., Plasmeijer, R. (eds.) AAIP 2009. LNCS, vol. 5812, pp. 50–73. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11931-6_3
12. McKay, R., Hoai, N., Whigham, P., Shan, Y., O'Neill, M.: Grammar-based genetic programming: a survey. Genet. Program. Evol. Mach. **11**(3–4), 365–396 (2010)
13. O'Neill, M., Nicolau, M., Agapitos, A.: Experiments in program synthesis with grammatical evolution: a focus on integer sorting. In: 2014 IEEE Congress on Evolutionary Computation (CEC), pp. 1504–1511, July 2014
14. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers, Norwell (2003)
15. Pantridge, E., Helmuth, T., McPhee, N.F., Spector, L.: On the difficulty of benchmarking inductive program synthesis methods. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2017, pp. 1589–1596. ACM, New York (2017)
16. Saber, T., Fagan, D., Lynch, D., Kucera, S., Claussen, H., O'Neill, M.: Multi-level grammar genetic programming for scheduling in heterogeneous networks. In: Castelli, M., Sekanina, L., Zhang, M., Cagnoni, S., García-Sánchez, P. (eds.) EuroGP 2018. LNCS, vol. 10781, pp. 118–134. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77553-1_8
17. Spector, L., Robinson, A.: Genetic programming and autoconstructive evolution with the push programming language. Genet. Program. Evol. Mach. **3**(1), 7–40 (2002)
18. Helmuth, T., Spector, L.: Detailed problem descriptions for general program synthesis benchmark suite. Technical report, School of Computer Science, University of Massachusetts Amherst (2015)
19. Whigham, P.A.: Grammatical bias for evolutionary learning. Ph.D. thesis, University of New South Wales, Australia (1996)
20. Yu, T.: A higher-order function approach to evolve recursive programs. In: Yu, T., Riolo, R., Worzel, B. (eds.) Genetic Programming Theory and Practice III. GPEM, pp. 93–108. Springer, Boston (2006). https://doi.org/10.1007/0-387-28111-8_7