Some Remarks on Code Evolution with Genetic Programming

Wolfgang Banzhaf

Abstract In this essay we shall take a fresh look at the current status of evolving computer code using Genetic Programming methods. The emphasis will not be so much on what has been achieved in detail in the past few years, but on the general research direction of code evolution and its ramifications for GP. We shall begin by a quick glance at the area of Search-based Software Engineering (SBSE), discuss the history of GP as applied to code evolution, consider various application scenarios and speculate on techniques, that might lead to a scaling-up of present-day approaches.

1 Search-based Software Engineering

Search-based Software Engineering is a technique within the area of Software Engineering that formulates software engineering problems as search problems which can be addressed by established (meta-)heuristic search methods [20]. More generally, all kinds of Machine Learning approaches can brought to bear on these search/optimization problems, and possibly help accelerate the pace of programming, debugging and testing or repairing and repurposing of software. The ultimate goal of these approaches would be to allow computers to program computers, in other words ”automatic programming”.

All Engineering disciplines are pragmatic in the approach to their respective object of study. Software Engineering (SE) is no exception. When Harman and Jones wrote their manifesto on search-based approaches to SE in 2001 [20], they wondered why SE had been so slow to take up the challenges of search/optimization that other

Wolfgang Banzhaf
BEACON Center for the Study of Evolution in Action and
Department of Computer Science and Engineering
Michigan State University, East Lansing, MI, 48824, USA
e-mail: banzhafw@msu.edu

Engineering disciplines had taken up much earlier. They cite the widespread use of genetic algorithms and other metaheuristic search techniques in order to find satisfactory solutions to problems in SE often characterized by competing constraints, inconsistencies and contradictory goals, multiple solutions of varying complexity and reachable perfect solutions. Perhaps it was the ”youth” of SE that has been one of the reasons why this engineering discipline was not yet adopting techniques that at that time had already made serious in-roads in civil or mechanical, chemical, biomedical or electric engineering. But a decade after John Koza’s seminal book [29] on Genetic Programming, the time seemed to have been right to make this bold suggestion about the possible use of metaheuristics in SE.

A review of the search-based software engineering (SBSE) literature shows the main current thrusts of this area to be:

- Software testing [43]
- Non-functional property optimization [1, 55]
- Software bug repair [14]
- Requirement analysis [17]
- Software design, development and maintenance planning [9, 2, 3]
- Software refactoring [34]

These techniques are, however, far from a complete list of search methods applied in software engineering. For example, there is model checking as a way to provide a quality measure for searching in the space of programs [28, 27], software duplication (clone) search using various methods [51], semantics-based code search [50], code completion [46], code refactoring [45], code smell detection [13] and feature summarization [42], to name a few more examples where search techniques are or can be applied.

2 History of Genetic Programming as applied to Code Evolution

The core idea of Genetic Programming has, of course, always been to *program* computers using inspiration from the theory of natural evolution [29]. However, for the first two decades of its application, GP mostly was used to evolve algorithms that could model systems (symbolic regression) [30] or classify data (a typical machine learning task) [12].

When we wrote our textbook for GP in 1998 [6], the ambition of writing larger pieces of software with GP - and thus to automatically program - was considered to be far from being realistic. Though the vision was upheld and we claimed that we live, as far as programming computers is concerned, in an age similar to the Middle Ages before the invention of printing with movable letters, we did not see an easy path to achieving this goal. But why would one have to put code together by hand? Wouldn’t an automatic method, just taking the specification of program behavior in a suitable form be enough to instruct a GP-driven process to fulfill these expectations? We also claimed that programmers are acting in a similar way as one would have to expect from a GP method [6], by writing code in pieces, copying and pasting, inserting changes, i.e. mutating, by recombining and adapting functions that were originally designed for different purposes or in different contexts.

While GP had successfully evolved small pieces of code from scratch, the method could not easily scale up to large program packages or anything beyond 100-200 lines of code. So scalability was a big problem, along with the need to specify complex behavior for a multi-faceted application with many different use cases.

Nevertheless, researchers examined the search spaces of programs, often making use of radical simplifications, like examining GP's behavior in the search space of Boolean functions. In a series of works, W.B. Langdon and R. Poli examined these search spaces and the behavior of GP in it [37]. And while this led to the recognition of important factors, like the complexity of a solution [32], or the different effects of operators when searching in this space [33], or the advantages of neutrality [25], the question of how to scale up GP to large search spaces remained unresolved [7].

In the domain of symbolic regression and classification/supervised learning, it was again John Koza who has demonstrated how to scale up to more complex problems. Using a method devised by Gruau [18] for neural networks akin to a developmental process, Koza and co-workers were able to demonstrate the ability of GP to solve more complex problems [31]. Development and the rewriting of code was also at the center of self-modifying Cartesian GP (SMCGP) [19]. In that case, rather than complexity, the issue was generalization ability.

As mentioned, these latter developments took place in the classical GP application spaces of regression/modeling and classification/learning. In the second half of the last decade, however, important steps were taken that ultimately led to the realization that we might not be so far away from using GP systematically for code evolution.

One of the most important was the demonstration of bug repair capabilities of a GP system later called GenProg [54]. A program is taken that performs well under many test cases, but fails under a few (the bug), and repaired by patching the bug. Three key innovations allowed this step, (i) operation in a high-level search space provided by the abstract syntax tree (AST) of programs; (ii) assuming that the correct program behavior is hidden somewhere in the program, and thus no new code needs to be generated, only already existing code of that program is used; (iii) variation operators focus on code executed on the failed test cases only. In a series of demonstrations, the authors showed convincingly that GP can be applied to this domain, even when the programs consist of hundreds of thousands of lines of code [40].

Another important step was what has become known as ”evolutionary” or ”genetic improvement” of programs. Again, existing programs were taken as input, and subjected to evolutionary/genetic optimization processes for non-functional features like energy consumption, execution time etc, functions that sometimes are under the influence of compiler flags [5]. In this case, test cases that could assure correct program behaviour were either used or co-evolved with the program. The goal was to keep the functionality of the program while optimizing for other features (non-functional). One of the main issues in these approaches is the seeding of the population. One wants to start with a semantically correct program, and move through the search space in a way that allows to keep the semantic validity of program variants, while at the same time optimizing one or more non-functional goals.

Arcuri et al. [5] solve this problem by applying variations to the program, then filtering - based on test cases that are taken from a test case population - only those program variants that behave correctly. In their study, which took 8 algorithms written as C functions of a maximum of 100 lines of code, they found that thousands of improved code versions were generated with various trade-offs. Many times, GP exploited features of the code that were on the semantic level and could not be addressed by a regular compiler.

Harman et al. [23] approach the improvement of code using a BNF grammar for decomposing the program to be improved. Again, non-functional features, like complexity or speed of execution are targeted, but also an improvement in the accuracy of the program behavior is selected for, so the process is multi-objective. The BNF grammar extracted is a representation of the program that can be subjected to variations via mutation and crossover, and the programs that can be generated by those modifications are subjected to test cases, with their behavior compared to the original program. While this works well for correct programs, the authors point out that often it is beneficial to have additional information at hand that allows to judge the functional properties of a program. The original program as well as other sources of information can be used as an oracle for testing whether the variant program is working correctly or not. In order to scale up to reasonable code examples, the authors focus the search on heavily used parts of the program and their corresponding BNF grammar parts. As for the non-functional criteria, again the pieces of code that are critical for their fulfillment are most targeted by variation operators [35].

In summary, we can see techniques being developed to improve or repair existing programs with a focus on instructions that play a role in the behavior to be repaired or improved. Methods are also developed to either co-evolve test cases, or distribute test cases such that more difficult tests are sufficiently represented in the set. More generally, all kinds of Machine Learning approaches can brought to bear on these search/optimization problems, and possibly help accelerate the pace of programming, debugging and testing or repairing and repurposing of software. I shall briefly indicate this research direction in the next section.

3 Machine Learning

The field of program synthesis is clearly much bigger than genetic programming. Learning programs from examples, a task frequently posed to Genetic Programming, is known in Machine Learning as ”oracle-based” program synthesis [26]. Integer Logic Programming (ILP) and other classical machine learning approaches have been applied on program synthesis [39].

For example, code completion is a task that can be approached from the language perspective. By analyzing usage patterns and building probabilistic models of patterns in code, it is possible to complete code in constrained applications with high precision [46]. So instead of the programmer being forced to type out all lines of code completely, a less tedious approach could be used. While this does not allow really automatic programming, the transition between manual and automatic programming becomes continuous.

Machine learning has also been applied to predict a correct program from a set of programs on the behavior of input/output pairs. The task here is to rank programs that are correct vs those that are not, based on their behavior. Programming-by-example tasks [44] are becoming more widespread and the ranking of correct programs is at the border between machine learning and formal methods in computing [53].

While Genetic Programming is a pioneer in the field of code synthesis, other Machine Learning paradigms have taken notice. We can expect, over the next decade, that a number of interesting developments in the area of automatic programming and code synthesis will emerge.

4 The Use of Genetic Programming in Code Evolution Tasks

4.1 An overview of the tasks approachable by GP

Before we discuss a few principles that shine through previous successes in applying GP to code evolution, let's first have a look at the different tasks that GP could be applied to. This list is not exhaustive as I am sure I have overlooked possibilities.

1. **Code Adaptation**
   Starting from working code for one processor type, how can we evolve code for another?
   The simple answer would be to say that if we have an abstract language construct, we can compile it to different target processors, so the compiler takes care of the adaptation. This is fine in general, but does not allow to take advantage of the functionality of the target architecture.
   A typical example would be code developed for a CPU (of any kind) needing adaptation to a GPU architecture. See Langdon et al. [36] for an example.

2. **Code Synthesis**
   Given a behavioral description of a program, how can we synthesize it from a source code base?
   We need to search in the semantic space of programs to find relevant functions or code to use and adapt, or at least snippets that can be put together to produce the overall code looked for.
   Semantic search is an active area of research in Software Engineering and its methods could be brought to bear on this problem [50].

3. **Code Generation / Code Creation**
   Given a behavioral description of a program, how can we generate the code to produce that behavior?
   This was the original question for GP. One uses test (fitness) cases to define the input/output relation of a program, to define its behavior. It normally starts from a population of random programs to produce the desired behavior. A description is sometimes called I/O oracles. It could also be done on a higher ”specification level”.

4. **Code Optimization**
   Given code, how can we optimize it with respect to certain (non-functional or functional) criteria?
   Those criteria might be efficiency, speed of execution, robustness, assurance and security criteria, resilience, or accuracy. This has been proposed under the title of Genetic Improvement (GI), with the GISMOE approach being a prime example [49].

5. **Code Repair**
   Given buggy code and a behavioral description, how can we repair the code to make sure it functions properly?
   This task sometimes overlaps with Task 4. Evolving bugs is a very successful application for GP, as exemplified by the GENPROG system [54, 14].

6. **Code Grafting**
   Given the need to add some functionality to a program that already exists in another piece of code for a different purpose, how to graft this piece?
   Examples of this task have been examined in [48, 8] under the name ”program transplantation”.

7. **Code Testing**
   Given a working program, how can we evolve tests that lead to its failure (in order to remove those conditions that lead to the failure)?
   The production/evolution of test cases is an active area of research in Software Engineering [11]. With GP, it could be considered as the evolution of test cases, or the co-evolution of program code and test cases, as already the work of White et al. have shown [55].

4.2 Main Aspects of Code Evolution

The above applications have a couple of overlapping aspects that are studied in connection with GP in code evolution. Those can be grouped into three categories:

(a) Population seeding
(b) Search bias

(c) Test case application

(a) refers to the question of how to initialize a population of programs at the beginning of the search. The most common procedure in GP so far, but probably not the method to be used in difficult programming problems, is to start the search with a random population. Similar to other engineering problems, it might be better to start from an existing solution and to evolve away from that solution in search of one that is better in regard to different criteria. Incorporating expert knowledge via the seeding strategy has been examined in the past [52], however, we believe there is much to be done to come up with efficient methods for programming.

(b) refers to the fact that not all search directions in a large search space are equally promising. For that reason, most researchers in SBSE adopt a strategy where they first examine which parts of the code are used (at all or frequently) under the conditions required (i.e. of input/output pairs, or for non-functional criteria). Once those code segments have been found, they are subjected to intense search modifications, while other parts which are not used at all, or used only infrequently, are left aside as probably not relevant. An inquiry into similar questions has occupied Biology since a long time under the heading of hyper-mutations [41].

(c) has been an ongoing discussion in Genetic Programming. To evaluate test cases is expensive, so the question is whether we have to run all test cases on all individuals in the population all the time. There have been various scenarios how to avoid this approach. Sampling techniques have been proposed, both based on difficulty of the test case, or on the prior frequency of its use [16, 38, 10]. The co-evolution of test cases has been a topic at least since the early work of Hillis [24] and has been applied very early in bug fixing [4, 5].

All three categories of questions aim at improving the scalability of GP search in programming spaces which for any reasonable program is of enormous size. In what follows we shall provide a few speculative thoughts of how these key questions will develop over the next few years, and where, to my mind, possible solutions could be found.

5 How to Scale Up?

The current section is not a comprehensive review of work in SBSE and what has been achieved (a good survey is [21, 22]), but an attempt at pointing to a number of key areas where progress can be expected in the future.

The reader might have noticed that we haven't discussed program representation for GP at all so far. How is the code represented such that GP can search efficiently and come up with functional code versions without having to wait until the end of the universe? We know that program spaces are notoriously large, so search efficiency is of tantamount importance. So we can safely add

(d) code representation

for GP into the list above.

The original work on evolutionary improvement of existing software used abstract syntax trees (ASTs) of the source code written in a computer language like C or JAVA. This representation is also commonly used in program analysis and program transformation, and a set of tools exist to extract ASTs from source code, as they are an intermediate step in compiling code. However, it turns out that working on the ASTs of program source code and manipulating it directly by genetic operations does not scale well when trying to improve programs of larger sizes.

As a result, most recent work is done with a differential representation where an input program remains the reference point for a collection of changes that comprise the GP individual. This approach scales much better than the non-differential representation. A sequence of insert/delete and replace operations on the AST - or in the case of a BNF grammar, changes to lines of code represented in the grammar - comprise the individual, again always requiring the input program as a reference point. This is in line with the idea of ”improvement” of an existing program and could be termed a linear GP system for code improvement.

Clearly, this approach is only useful for improvements of working programs, or is it? I think that the very same approach can be used to evolve programs more or less from scratch. The idea is to consider a sequence of program improvements just like the evolution from a primitive cell (primitive working program) to a sophisticated and specialized cell (functional program for an arbitrary task). What would need to happen is that the reference input program is, at certain moments we can call ”epoch ends”, replaced by the best program so far. In other words, the reference program is replaced by a (usually) more complex one that has made progress toward the task enshrined in fitness test cases. At the beginning of a new epoch, the GP individuals of earlier epochs are set aside and replaced by newly initialized very short programs (perhaps simple mutations to the now new reference individual). As changes accrue during an evolutionary epoch, individuals grow in length until again an epoch comes to an end and the reference individual is replaced.

This way, a very simple (stem cell type?) program at the beginning of a run, could be used to evolve in different directions, guided by test cases, and end up in different behavioral spaces that represent different functionalities. This type of programming would therefore require a very general and simple input program, that will gradually be augmented and specialized into different directions, depending on the test case specification. Is code evolvable in this way? I suspect so.

At first sight, this strikes as a complicated way to produce complex code, however, there are shortcuts. In this connection it is helpful to recall that, according to a statistical examination of code bases, most small segments of code (up to a length of approximately 6 or 7 lines of code) have already been written somewhere [15]. If there is a way to harness from the environment this code already existing, this would greatly facilitate the task of assembling a complex program. This will require to traverse the feature space of existing code to find relevant segments [42].

A well-known mechanism useful for building up complexity could be put to use as well: duplication and divergence of modules. Natural evolution seems to have proceeded through a number of repeated steps of duplication and divergence [47] and there is no reason to doubt that the same method couldn't be used with benefit when building up program complexity.

Another question: Are there other programmers (or programming agents) that have faced a similar task, and how did they proceed in the source language (or in another language) to write the code necessary to achieve the goal?

Finally: Is the task decomposable into smaller parts that have been addressed elsewhere in the programmer's own work or in that of others? If that is the case, the evolution could be split into independent parts run on the simpler tasks.

In summary: Besides the tricks of complexity handling indicated in [7], we would use a differential code representation that would undergo epochs of evolution under the control of test cases. Even the test cases could be switched at the changes of epochs, for instance, to provide more detailed guidance on the desired program behavior. Much of what is described here might be categorized as a developmental approach, though there are key ingredients of real computational development still missing from the picture. However, it is reasonable to expect that evo-devo approaches could become very successful.

6 Conclusion

In this chapter we have looked at code evolution and discussed a few recent developments in this area. We saw that the scalability of code evolution approaches is an important obstacle. However, I believe that in the last decade breakthroughs have been made that have fundmentally altered the playing field for the use of GP in code evolution. This application of GP has moved from a dream in the 1990s to a realistic opportunity in the last 10 years. It is time to invest huge research efforts in this area and to harness the continuing growth in speed of our computer hardware.

To give one example where the interaction of GP code evolution and new developments in Computer Science could be very useful is in the area of ”big code”. Following von Neumann's lead for using the same physical representation for data and code, the term ”big code” has been coined in equivalence to the earlier term ”big data”. Big code refers to collections of code, like on GitHub, which can be scoured or exploited for other tasks, if the appropariate tools are available to find the right entries [56]. This brings to attention the need for tools for semantic search in program spaces [50].

In this contribution we have discussed a number of research questions that might be fruitfully addressed in the application of GP to code evolution. Besides the different tasks mentioned in Section 4, key areas of investigation should be the topics (a) to (d) raised in the same Section later. As well, research about a good way to construct complexity, in particular in connection with an approach that makes use of the ”natural” complexity of the environment - how to find the right function, how to integrate it with the existing program - are pertinent to code evolution.

Code evolution is of enormous economic importance. Genetic programming has posed new kinds of questions and opened up new ways to think about this problem.

Exciting new avenues are before us. I believe that GP has a good opportunity to make serious contributions to this area.

Acknowledgements This essay was written on the occasion of the Festschrift for Julian Miller's 60th birthday. It is dedicated to Julian, a wonderful friend and inspiring colleague.

References

1. Afzal, W., Torkar, R. and Feldt, R.: A systematic review of search-based testing for non-functional system properties. *Information and Software Technology* **51** 957–976 (2009)
2. Alba, E., Chicano, J.F.: Software project management with GAs. *Information Sciences* **177** 23802401 (2007)
3. Antoniol, G., Di Penta, M., Harman, M.: Search-based techniques applied to optimization of project planning for a massive maintenance project. In: *Proc. of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05*, pp. 240249. IEEE Press, New Jersey (2005).
4. Arcuri, A. and Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In: *Proc. of the 2008 IEEE Congress on Evolutionary Computation*, IEEE Press, New York, pp. 162–168 (2008)
5. Arcuri, A., White, D.R., Clark, J. and Yao, X.: Multi-objective improvement of software using co-evolution and smart seeding. In: *Proc. Int. Conf. SEAL 2008*, pp. 61–70 (2008)
6. Banzhaf, W., Nordin, P., Keller, R. and Francone, F.: *Genetic Programming - An Introduction*. Morgan-Kaufmann, San Francisco, CA (1998)
7. Banzhaf, W. and Miller, J.: The challenge of complexity. In: Menon, A. (Ed.) *Frontiers of Evolutionary Computation*. Springer, Berlin. pp. 243-260 (2004)
8. Barr, E.T., Harman, M., Jia, Y., Marginean, A., and Petke, J.: Automated software transplantation. In: *Proc. of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015*, pp. 257–269 (2015)
9. Clark, J.A., Jacob, J.L.: Protocols are programs too: the meta-heuristic search for security protocols. *Information and Software Technology* **43** 891904 (2001)
10. Curry, R., Lichodzijewski, P., and Heywood, M.: Scaling genetic programming to large datasets using hierarchical dynamic subset selection. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* **37** 1065–1073 (2007)
11. Dustin, E., Rashka, J., and Paul, J.: *Automated Software Testing*. Addison Wesley (1999)
12. Espejo, P.G., Ventura, S. and Herrera, F.: A Survey on the Application of Genetic Programming to Classification. *IEEE Transactions on Systems, Man and Cybernetics - C* **40** 121–144 (2010)
13. Fontana, F., Zanoni, M. and Marin, A.: Code Smell Detection: Towards a Machine Learning-based Approach. In: *Proc IEEE International Conference on Software Maintenance*, pp.396-399. IEEE Press, New Jersey (2013)
14. Forrest, S., Nguyen, T., Weimer, W. and Le Goues, C.: A genetic programming approach to automated software repair. In: *Proc. of the 11th Annual conference on Genetic and evolutionary computation*, pp. 947–954. ACM Press, New York (2009)
15. Gabel, M., and Su, Z.: A Study of the Uniqueness of Source Code. In: *Proc. of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ACM Press, New York, pp. 147–156 (2010)
16. Gathercole, C. and Ross, P.: Dynamic training subset selection for supervised learning in genetic programming. In: *Proc. International Conference on Parallel Problem Solving from Nature*, Springer, Berlin, pp. 312–321 (1994)
17. Greer, D., Ruhe, G.: Software release planning: an evolutionary and iterative approach. *Information and Software Technology* **46** 243253 (2004)

18. Gruau, F.: Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm. PhD Thesis. Laboratoire de l'Informatique du Parallelisme, Ecole Normale Supirieure de Lyon (1994)
19. Harding, S., Miller, J.F., and Banzhaf, W.: Developments in cartesian genetic programming: self-modifying CGP. *Genetic Programming and Evolvable Machines* **11** 397–439 (2010)
20. Harman, M., Jones, B.F.: Search-based software engineering. *Information and Software Technology***43**, 833–839 (2001)
21. Harman, M., Mansouri, S. A., and Zhang, Y.: Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications. Department of Computer Science, King's College London Technical Report TR-09-03 (2009)
22. Harman, M., Mansouri, S. A., and Zhang, Y.: Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys* **45** 11:1–11:61 (2012)
23. Harman, M., Langdon, W.B., Jia, Y., White, D.R., Arcuri, A., and Clark, J.: The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs. In: *Proc. The 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 12)*, ACM Press, New York, pp. 1–14 (2012)
24. Hillis, W.D.: Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D: Nonlinear Phenomena* **42** 228–234 (1990)
25. Hu, T., Payne, J.L., Banzhaf, W. and Moore, J.H.: Evolutionary dynamics on multiple scales: a quantitative analysis of the interplay between genotype, phenotype, and fitness in linear genetic programming. *Genetic Programming and Evolvable Machines* **13** 305–337 (2012)
26. Jha, S., Gulwani, S., Seshia, S.A., and Tiwari, A.: Oracle-guided component-based program synthesis. In: *Proc. 32nd ACM/IEEE Intl. Conf. on Software Engineering (ICSE-2010)*, ACM, New York, pp.215–224 (2010)
27. Johnson, C.G.: Genetic Programming with Fitness based on Model Checking. In: *European Conference on Genetic Programming*, pp. 114–124. Springer, Berlin (2007)
28. Katz, G., Peled, D.: Model checking-based genetic programming with an application to mutual exclusion. In: C.R. Ramakrishnan and J. Rehof (eds.) *TACAS 2008*, LNCS 4963, pp. 141156 (2008)
29. Koza, J.: *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press, Cambridge (1992)
30. Koza, J.R.: *Genetic programming II: Automatic discovery of reusable subprograms*. MIT Press, Cambridge, MA (1994)
31. Koza, J.R.: *Genetic programming III: Darwinian invention and problem solving*. Morgan Kaufmann, San Francisco, CA (1999)
32. Langdon, W.B.: Scaling of program tree fitness spaces. *Evolutionary Computation* **7** 399–428 (1999)
33. Langdon, W.B.: Size-fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines* **1** 95–119 (2000)
34. Langdon, W.B. and Harman, M.: Genetically improved CUDA C++ software. In: *Proc. of the European Conference on Genetic Programming*, pp. 87–99. Springer, Berlin (2014)
35. Langdon, W.B. and Harman, M.: Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation* **19** 118–135 (2015)
36. Langdon, W.B. and Harman, M.: Genetically improved CUDA C++ software. In: *Proc. European Conference on Genetic Programming*, Springer, Berlin, pp. 87–99 (2014)
37. Langdon, W.B. and Poli, R.: *Foundations of Genetic Programming.* Springer, Berlin (2002)
38. Lasarczyk, C.W.G., Dittrich, P., and Banzhaf, W.: Dynamic subset selection based on a fitness case topology. *Evolutionary Computation* **12** 223–242 (2004)
39. Lau, T.A., Weld, D.S.: Programming by Demonstration: An Inductive Learning Formulation. In: *Proc. 4th international conference on Intelligent user interfaces IUI-1999*, ACM, New York, pp. 145–152 (1999)
40. Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W.: GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* **38** 54–72 (2012)
41. Martincorena, I. and Luscombe, N.M.: Non-random mutation: The evolution of targeted hypermutation and hypomutation. *Bioessays* **35** 123–130 (2012)

42. McBurney P.W., Liu, C. and McMillan, C.: Automated feature discovery via sentence selection and source code summarization. *J. of Software: Evolution and Process* **28** 120–145 (2016)
43. McMinn, P.: Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* **14** 105156 (2004)
44. Menon, A., Tamuz, O., Gulwani, S., Lampson, B., Kalai, A.: A machine learning framework for programming by example. In: *ICML (2013), JMLR W & CP Proc.*, Vol 28 (2013)
45. Mens, T., Tourwe, T.: A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* **30** 126–139 (2004)
46. Nguyen, A.T., Nguyen, T.T., Nguyen, H.A., Tamrawi, A., Nguyen, H.V., Al-Kofahi, J., and Nguyen, T.N.: Graph-based pattern-oriented, context-sensitive source code completion. In: *Proc. of the 34th International Conference on Software Engineering*, pp. 69–79. IEEE Press, New Jersey (2012)
47. Ohno, S.: *Evolution by Gene Duplication.* Springer, New York (1970)
48. Petke, J., Harman, M., Langdon, W.B., and Weimer, W.: Using genetic improve- ment and code transplants to specialise a C++ program to a problem class. In: *Genetic Programming - Proc of the 17th European Conference, EuroGP 2014*, Springer, Berlin, pp. 137–149 (2014)
49. Petke, J., Langdon, W.B., and Harman, M.: Applying genetic improvement to MiniSAT. In: *Proc. International Symposium on Search Based Software Engineering*, Springer, Berlin, pp. 257–262 (2013)
50. Reiss, S.P.: Semantics-Based Code Search. In: *ICSE09, May 16-24, 2009*, pp. 243–253, IEEE Press, New Jersey (2009)
51. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Queens School of Computing TR 541-2007 (2007)
52. Schmidt, M.D. and Lipson, H.: Incorporating expert knowledge in evolutionary search: a study of seeding methods. In: *Proc. of the 11th Annual conference on Genetic and evolutionary computation*, ACM Press, New York, pp. 1091–1098 (2009)
53. Singh, R. and Gulwani, S.: Predicting a Correct Program in Programming by Example. In: D. Kroening and C.S. Pasareanu (Eds.), *Proc. CAV 2015*, Springer, Switzerland, pp. 398–414 (2015)
54. Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S.: Automatically Finding Patches Using Genetic Programming. In: *Proc. of ICSE09, May 16-24, 2009, Vancouver, Canada*, pp. 364–374. IEEE Press, New Jersey (2009)
55. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary Improvement of Programs. In: *IEEE Transactions on Evolutionary Computation* **15** 515–538 (2011)
56. Yahav, E.: Analysis and Synthesis of "Big Code". In: J. Esparza et al. (eds.), *Dependable Software Systems Engineering*, IOS Press (2016)