# Rethinking Genetic Improvement Programming

David R. White
School of Computing Science
University of Glasgow
Scotland
david.r.white@glasgow.ac.uk

Jeremy Singer
School of Computing Science
University of Glasgow
Scotland
jeremy.singer@glasgow.ac.uk

## Keywords

Genetic Programming, Genetic Improvement Programming

## ABSTRACT

We re-examine the central motivation behind Genetic Improvement Programming (GIP), and argue that the most important insight is the concept of applying Genetic Programming *to existing software*. This viewpoint allows us to make several observations about potential directions for GIP research.

## 1. INTRODUCTION

In some ways, we may argue that Genetic Improvement Programming (GIP), or simply GI, would have been the logical *first step* for Genetic Programming (GP): to amend existing software rather than create it from scratch appears to be a logical line of attack for the development of search-based code generation. Perhaps because the roots of GP are in the artificial intelligence community, rather than the software engineering community, this was not the path that GP research followed. The recent emergence of GIP as a major strand of GP research, including several notable high-profile achievements and publications e.g. [12, 15], has pivoted the community back towards this more incremental approach.

Examined from this viewpoint, the major insight of GIP is that *existing software is a valuable resource for GP*. We examine the ways in which existing code can be exploited to aid the search process, and discuss the implications of those opportunities.

## 2. THE UTILITY OF EXISTING CODE

There are three reasons to value existing software:

1. As an Oracle.
2. To provide scalability.
3. To provide ready-made functionality.

We examine each in turn.



*GECCO '15, July 11 - 15, 2015, Madrid, Spain*



DOI: http://dx.doi.org/10.1145/2739482.2768426

### 2.1 As an Oracle

Existing code can function as an oracle for software testing or, in the case of bug-fixing applications, provide a partially correct solution. This enables the calculation of expected results from a set of test cases. Furthermore, if we relax our definition of existing software to include unit tests and other artifacts then we have a set of input-output pairs. Without tests, we require input data: this may be provided by an engineer or researcher when formulating the problem, but equally automated testing approaches such as fuzz testing [8] and Search-Based Software Testing [7] can be applied.

To date, GI research has used the oracle to test whether potentially semantics-breaking transformations have broken the functional specification of code. When bug-fixing, positive test cases are run through the oracle; when optimising a non-functional property such as execution time, the oracle can be used with any input and the output matched against the output of the modified program.

What other opportunities does the oracle offer us? Until now we have been focused on the process of program improvement; consider more general *transformation*. We may transform a program into a different form, for example by introducing concurrency. In fact, the latter was the subject of a book that may be the very first on GIP, though it was not recognised as such at the time [14]. Cloning software in a more general sense using GP may be useful for reverse engineering [4] or N-version programming [2].

Previous GIP work may be viewed as *specialisation* of software for anticipated usage, or for a given platform. For example, the work of Langdon [6] produced a specialised version of the Bowtie2 genomic analysis software that patched a program so as to make great gains in execution speed on average, at the cost of partial loss of correctness. There is no reason why we should not go a step further and optimise for a given architecture, such as ARM or MIPS.

**Example Research Direction** One way of transforming code using GIP would be to translate it to another language: we can use existing software as an oracle to ensure equivalence between the source and target versions.

### 2.2 For Scalability

GP has known scalability issues [11]: it suffers from bloat [13] and is mostly applied to the generation of small S-expressions in applications such as symbolic regression. The use of existing software promises to improve scalability, in that the overall artifact generated is large, even if our patch is relatively small; GP can be applied to "real-world" software. Furthermore, by copying parts of a program and in-

serting them elsewhere, the size of the individual units under manipulation are larger than in traditional GP. The principal contribution of the existing program to scalability in this sense is that *it provides a starting point in the search space that is close to the overall goal.* The reuse of program subcomponents also provides some guidance as to the possible mutation operations we may apply during the search.

Existing code provides a much richer source of information than this. The most obvious approach to aiding our search is to profile the original program, which is used in GI bug-fixing applications where the lines executed under both positive and negative test cases are contrasted. Profiling can also be used in optimisation of execution time, to identify areas of the code responsible for a great deal of time taken during execution. In the future, GIP should move beyond simple profiling: individual traces of a program may be used to extract the semantics of intermediate steps in the program, reducing the size of the problem to be solved by considering only a subset of operations. More generally, any static or dynamic analysis may be applied to extract useful knowledge from the input program, a small fraction of which is currently exploited.

**Example Research Direction** Little attention has been paid to memory usage in GIP. Static analysis techniques can be used to fix memory errors in existing code, and GIP could also be used to reduce memory usage through heap profiling.

## 2.3 As Ready-made Functionality

In recent bug-fixing work, existing code is used as raw material, i.e. genetic material to be inserted into the program. Weimer et al [15] themselves cite Engler et al. [1] when they observe:

> "In practice, a program that makes a mistake in one location often handles the situation correctly in another."

We may broaden our horizons much further than this. We may consider the combination of existing code from *multiple sources.* When solving problems "from scratch" in GP, we may exploit codebases found in open-source projects, libraries and forums. Consider the concept of StackSort, an idea posited by the XKCD author Randall Munroe [9] and later implemented by Koberger [5]. Although tongue-in-cheek, it illustrates an important point, and mirrors the way Noble and Biddle [10] describe (post-)modern programming as "Scrap-Heap System Construction".

Broadening our definition of software to include other development artifacts such as documentation based in code (e.g. Javadoc) and change control history, we may employ data-mining methods to identify important and potentially reusable code, as well as examining code for type-compatibility.

Rather than combine small components to construct large programs, an incremental approach seems more logical, i.e. adding new features to existing programs rather than starting from scratch. The method of "grow and graft" proposed by [3] attempts to create a subcomponent independently and insert it into a program; we propose that the subcomponent is generated by code scavenging from the vast repositories of code online.

**Example Research Direction** GIP can exploit the vast repositories of source code online to solve new problems using existing code. We term this "code scavenging".

# 3. CONCLUSION

In this position paper we have argued that the greatest insight of Genetic Improvement is the value of incorporating existing source code into the automated programming process. We have argued that this viewpoint identifies new ways forward for GI: program transformation, translation, cloning; code scavenging and recombination; and the full exploitation of code and related artifacts to guide our search for solutions.

# 4. REFERENCES


[1] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *SOSP '01*, 2001.

[2] R. Feldt. Generating Multiple Diverse Software Versions with Genetic Programming. In *Euromicro Conference, 1998. Proceedings. 24th*, 1998.

[3] M. Harman, Y. Jia, and W. B. Langdon. Babel Pidgin: SBSE Can Grow and Graft Entirely New Functionality into a Real World System. In *Search-Based Software Engineering 2014*. 2014.

[4] M. Harman, W. B. Langdon, and W. Weimer. Genetic Programming for Reverse Engineering. In *20th Working Conference on Reverse Engineering (WCRE 2013)*, 2013.

[5] G. Koberger. stacksort. http://gkoberger.github.io/stacksort/, 2015. [Online; accessed 8-April-2015].

[6] W. Langdon. Performance of Genetic Programming Optimised Bowtie2 on Genome Comparison and Analytic Testing (GCAT) Benchmarks. *BioData Mining*, 8(1):1, 2015.

[7] P. McMinn. Search-Based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[8] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, 1990.

[9] R. Munroe. XKCD: Ineffective Sorts. https://xkcd.com/1185/, 2015. [Online; accessed 8-April-2015].

[10] J. Noble and R. Biddle. Notes on Postmodern Programming. In *Proceedings of the Onward Track at OOPSLA 2002*, 2002.

[11] M. O'Neill, L. Vanneschi, S. Gustafson, and W. Banzhaf. Open Issues in Genetic Programming. *Genetic Programming and Evolvable Machines*, 11(3-4):339–363, Sept. 2010.

[12] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *EuroGP 2014 Proceedings*. 2014.

[13] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[14] C. Ryan. *Automatic Re-engineering of Software Using Genetic Programming*. Springer US, 2000.

[15] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE '09*, 2009.