Genetic Improvement in Code Interpreters and Compilers

Oliver Krauss*
Johannes Keppler University Linz
Altenberger Street 69
Linz, Austria 4040
Institute for System Software

University of Applied Sciences Upper Austria
Softwarepark 13
Hagenberg, Austria 4232
oliver.krauss@fh-hagenberg.at

Abstract

Modern compilers provide different code optimizations before and during run-time, thus moving required domain knowledge about the compilation process away from the developer and speeding up resulting software. These optimizations are often based on formal proof, or alternatively have recovery paths as backup. Genetic improvement (GI), a field of science utilizing the genetic programming algorithm, a stochastic optimization technique, has been previously utilized to both fix bugs in software, as well as improving non-functional software requirements.

This work proposes to research the applicability of GI in an offline phase directly at the interpreter or compiler level, utilizing abstract syntax trees. The primary goal is to reformulate existing source code in such a way that existing optimizations can be applied in order to increase performance even further and requiring even less domain knowledge from the developer about a specific programming language and/or compiler. From these reformulations, language-specific patterns can be identified that allow code restructuring without the execution overhead GI poses.

CCS Concepts •**Software and its engineering → Software evolution; Search-based software engineering;**

Keywords Genetic Programming, Genetic Improvement, Non-Functional, Compilation, Code Optimization

1 Motivation

The process of creating a new programming language, or execution environment (interpreter or compiler) brings with it the challenge of optimizations to remain competitive with alternative environments. This work suggests the application of Genetic Improvement (GI) [3, 4] directly at the compiler or interpreter level to utilize it for programming languages. The research target is to adapt source code represented as an AST in such a way that the pre-existing optimizations in the interpreter or compiler can be utilized.

An example of a possible GI optimization is the reduction of the function body when containing purely functional code. While this is an optimization generally done by a compiler, this would have to be considered manually by someone developing or using an AST interpreter. During initial results this optimization has been consistently observed when applying GI to different ASTs. This makes it possible to find patterns that can be applied as optimizations.

2 Problem

Modern compilers utilize various optimizations such as dead-code elimination, control-flow optimization and branch prediction. These optimizations take place on different levels (representations) of the source code. Additionally, some optimizations are only applicable during the run-time of a program. Run-time optimizations require a warm-up phase in which the code is analyzed. Optimizations, such as branch predictions, can be stochastic, and don't guarantee an improved performance over every code instance. [5]

Truffle [12] is an interpreter and framework utilizing self - rewriting ASTs. It provides the implementer of a language with an API that can be used to write nodes to prototype and implement programming languages which can be interpreted in the Java Virtual Machine (JVM). In addition to interpreting the language, Truffle integrates with the Graal compiler to create optimized machine code from AST. As the framework is open source and extensible, it is used as research basis.

The research field of GI utilizes search based optimization, specifically the genetic programming (GP) algorithm, to improve software. The GP algorithm is an evolutionary search strategy that finds a solution to a given problem by breeding and mutating a population of different solutions over several generations. [2]

3 Approach

GI will be applied in programming languages, implemented in the Truffle framework and optimized by the Graal compiler in an offline phase. Due to the level of available reference publications and existing benchmarks an implementation of C in the Truffle framework has been chosen as initial starting point.

The application of GI in Truffle (RQ-1) requires analysis of the node implementations existing in the target language, since they represent the operators that can be utilized. Figure 1 shows a possible AST rewrite. The left side shows the AST parsed from developer-written source code. The function evaluates a mathematical expression using given input, which is loaded into the execution frame from the parameters, and returns the calculated value. On the right side the GI has removed the function body and the return statement, instead opting to directly access the parameters and returning the value. Nodes existing in both ASTs are marked by background color.

Concerning the execution context of an AST (RQ-2) the current approach in Genetic Improvement relies on the inherent mutational robustness of software [11]. The currently considered approach

*Advisors: Prof. Dr. Dr. h.c. Hanspeter Mössenböck (hanspeter.moessenboeck@jku.at), Prof. (FH) Priv.-Doz. Dipl.-Ing. Dr. Michael Affenzeller (Michael.Affenzeller@fh-hagenberg.at)

SPLASH, Vancouver, Canada

DOI: 10.1145/nnnnnnn.nnnnnnn

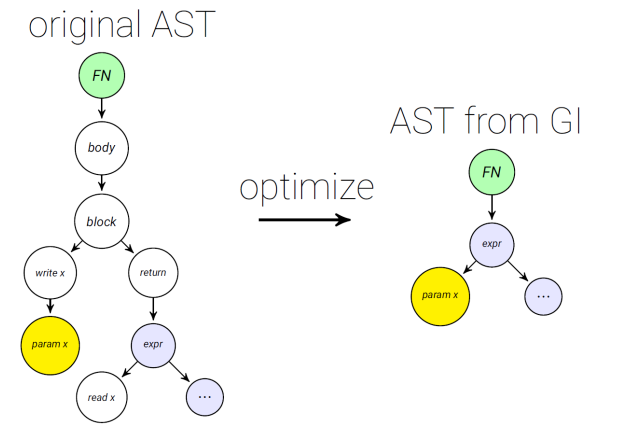

Figure 1. GI optimization of a function containing only a mathematical expression:
`int FN (int x) { return expr using x; }`

for this project is, in addition to manual test-cases, to automatically create a regression test suite for the AST being optimized. This is achieved by utilizing already existing test suite generation frameworks [8, 10].

Since some optimizations by GI have been observed to be similar in different tests, patterns can be identified that can be applied to an AST (RQ-3). Figure 2 shows a possible pattern created from the optimization shown in Figure 1. The concept is that the original AST structure is compared to the optimized AST and similarities are specifically marked (background color). From this comparison a transformation-pattern is generated which can be applied to an AST that shows the same structural hierarchy as defined in the pattern.

4 Results and Expected Contributions

This work is currently in the starting phase. A subset of the C11 version [1] of C has been implemented using the Truffle framework. It will gradually be extended to enable tests and benchmarks required for a thorough evaluation. The language has been enriched with an optimizer that uses GI to improve an AST based on a simple fitness function combining the amount of successfully executed tests and the average run-time of the AST.

A beneficial side effect of integrating GI directly with an interpreter was discovered during testing. While the GP algorithm can still produce ASTs that result in endless loops or other faults

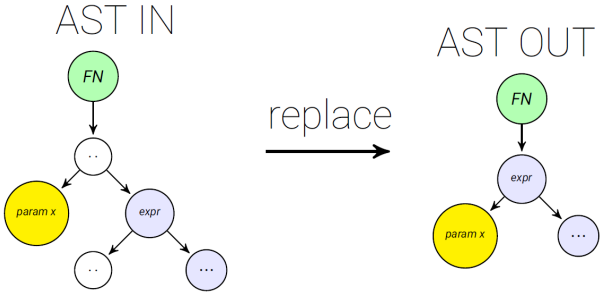

Figure 2. Pattern based on the example in Figure 1, automatically reducing the function body of a function containing a mathematical expression

observed during run-time, all generated ASTs are executable. In literature uncompilable individuals are often observed [6, 7, 9].

The expected contribution of this research work is:

- A case study presenting the effects of GI for optimizing non-functional requirements utilizing existing compiler techniques.
- An approach to optimize a prototyped language using the truffle framework.
- A pattern-identification framework to enable further research into compiler optimization techniques

References

[1] ISO. 2011. *ISO/IEC 9899:2011 Information technology — Programming languages — C.* International Organization for Standardization, Geneva, Switzerland. 683 (est.) pages. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853

[2] John R. Koza. 1990. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems.* Technical Report. Stanford, CA, USA.

[3] William B. Langdon. 2015. *Genetic Improvement of Software for Multiple Objectives.* Springer International Publishing, Cham, 12–28. DOI: https://doi.org/10.1007/978-3-319-22183-0_2

[4] W. B. Langdon and M. Harman. 2015. Optimizing Existing Software With Genetic Programming. *IEEE Transactions on Evolutionary Computation* 19, 1 (Feb 2015), 118–135. DOI: https://doi.org/10.1109/TEVC.2013.2281544

[5] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[6] Michael Orlov and Moshe Sipper. 2009. Genetic Programming in the Wild: Evolving Unrestricted Bytecode. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09).* ACM, New York, NY, USA, 1043–1050. DOI: https://doi.org/10.1145/1569901.1570042

[7] M. Orlov and M. Sipper. 2011. Flight of the FINCH Through the Java Wilderness. *IEEE Transactions on Evolutionary Computation* 15, 2 (April 2011), 166–182. DOI: https://doi.org/10.1109/TEVC.2010.2052622

[8] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07).* IEEE Computer Society, Washington, DC, USA, 75–84. DOI: https://doi.org/10.1109/ICSE.2007.37

[9] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2014. *Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class.* Springer Berlin Heidelberg, Berlin, Heidelberg, 137–149. DOI: https://doi.org/10.1007/978-3-662-44303-3_12

[10] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893. DOI: https://doi.org/10.1007/s10664-015-9424-2

[11] Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software Mutational Robustness. *Genetic Programming and Evolvable Machines* 15, 3 (Sept. 2014), 281–312. DOI: https://doi.org/10.1007/s10710-013-9195-8

[12] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013).* ACM, New York, NY, USA, 187–204. DOI: https://doi.org/10.1145/2509578.2509581