Evolving Compiled Code

Eric C. Collom
University of Minnesota, Morris
coll0474@morris.umn.edu

ABSTRACT

Stuff

General Terms

Keywords

1. INTRODUCTION

The limitation of only being able to evolve small parts of programs is an issue in evolutionary computation (EC). Much of these constraints are due to there is no graceful way to evolve programs, as a whole, directly at the source code. This is because source code exists to make it easier for humans to read and write programs. However, it is very hard to create a genetic algorithm (GA) that would evolve source code do to syntactical constraints.

A solution to this problem is to to compile the program to bytecode or machine code and evolve it there. This has been done in Java bytecode and x86 assembly [5] [6] which I will discuss thoroughly in this paper add more.

2. BACKGROUND

2.1 Evolutionary Computation

2.2 Assembly

Machine code that runs on a physical machine.

2.3 Java Bytecode

Java bytecode is compiled Java source code that runs on the Java Virtual Machine(JVM). It "is the intermediate, platform-independent representation of Java programs[1]." Each bytecode operation code (opcode) is one byte in length. However, some opcodes, that take perimeters, are multiple bytes long[3].

2.4 Grammar Stuff

Talk about syntactic and semantic constraints and BNF.

UMM CSci Senior Seminar Conference, April 2014 Morris, MN.

3. WHY EVOLVE JAVA BYTECODE AND X86

There are many problems we run into when trying to evolve source code. One problem is that it is extremely difficult to evolve an entire program due to source code syntactical constraints. Another problem is that we cannot just take a program and evolve it, we have to design the program to be evolvable. Evolving bytecode and x86 bypasses both of these problems.

3.1 Source Code Grammar Constraints

One problem with trying to evolve entire programs at the source code level is that there is a very high risk of producing a non-compilable program. This is due to the fact that high-level programming languages are designed to make it easy for humans to read and write programs. Most high-level programming languages are defined in Backus-Naur Form (BNF) which is purely syntactical [2] [5]. This means that the grammar does not represent the semantic constraints of a program. So for example, the BNF form does not capture the languages type system, variable visibility and accessibility, and other constraints [5]. In order to write a program to evolve source code we would have to deal with all these constraints. While this task is possible it would require writing a full-scale compiler. This would be a very taxing task.

Java bytecode and x86 Assembly contain a small alphabet of primitives [6]. Also they are a lot less syntactical. This means that there is a lot less risk of producing a non-compilable program during evolution. Sipper et al. [5] simply deals with the cases where a non-compilable program is produced by simply re-evolving the programs [5]. Since, the check to see if a program is compilable is cheap enough and the rate of which a program produced is compilable is high enough this is feasible.

3.2 Evolve As Is

For traditional EC, a program has to be designed specifically to be evolved. However, evolving programs in bytecode or x86 only require that the program can compile to one of the two. It doesn't require the program to be designed specifically for evolution [6] [5]. Many programming languages compile into Java bytecode and x86 and thus can be evolved. For example, Scala, groovy, Jython, Kawa, JavaFx Script, and clojure all compile into Java bytcode [5]. The only thing required is an algorithm to compute the fitness.

Another benefit to evolving bytecode and x86 is that you can evolve the entire program. In traditional EC when a program is designed it is more common to evolve an expressions or formulae [5] within the program. Most of the program as a whole would be already written and remain the same after evolution. In evolving bytecode and x86 most of the program doesn't even need to be written to be able to evolve it and find a solution. Also, the final product can be drastically different from the original program since the entire program was evolved. This allows for a lot more flexibility in evolving a programs. This also allows us to solve the problem as a whole instead of a small section of the problem.

4. HOW THE FINCH EVOLVES CODE

4.1 Selecting Good Offspring

Evolving Java Byte code only reduces the chance of evolving non-compilable bytecode to a curtain extent. Even though byte code is less syntactical it still is syntactical. Sipper et al.[4] addresses this issue by checking if a good offspring has been created before letting it join the evolved population. This is done by checking if it compiles. It it doesn't it then makes another offspring with the same parents. It repeats this process until a good offspring is produced or a specified number of attempts have been made.

The checks that are made make sure that all the variables will be read, written, type-compatible, and will not cause stack underflow[1]. It is important that only good offspring is selected because this provides good variability in the population. If non-comparable code would result in a fitness score of zero. Since noncompilable code would occur frequently enough it would cause a large portion of the population to have a zero fitness score.

4.2 Crossover

The FINCH uses two-point crossover to evolve bytecode. It takes two programs A and B and extracts sections a and b of bytecode respectively. It then takes section a and inserts it into where section b used to be. It only selects an a that will compile after being inserted into B. Take into account that just because a can replace b in B does not imply that b can replace a in A. In other words it is not a biconditional relationship.

4.3 Variable Access Sets

[4] much hard

5. EVOLVING ASSEMBLY CODE

6. RESULTS

6.1 FINCH

6.2 x86

6.3 Future Work

7. CONCLUSION

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] *Genetic Programming Theory and Practice VIII.* Springer New York, 2011.
This book provides good information on how the JVM works, FINCH grammar logic and crossover. Brings an aditional perspective. .

[2] Backus-naur form. January 2014.
This provides background for Backus-Naur Form .

[3] Java bytcode. Febuary 2014.
Wikipedia on bytcode (not just Java). .

[4] M. Orlov and M. Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1043–1050, New York, NY, USA, 2009. ACM.
This is Sipper's and Orlov's first paper on FINCH. I'm using this because it descripes crossover and how it interacts with the JVM alot bettter than their journal paper. .

[5] M. Orlov and M. Sipper. Flight of the finch through the java wilderness. *Evolutionary Computation, IEEE Transactions on*, 15(2):166–182, April 2011.
This will be one of the backbones of my paper. Talks about a program called FINCH that the authors developed. Finch is used to evolve programs at the Java bytecode level. This paper Has lots of information and they have about six examples where they evolved programs using FINCH .

[6] E. Schulte, S. Forrest, and W. Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 313–316, New York, NY, USA, 2010. ACM.
This will be a main source for my paper. It talks about evolving programs at assembly language level and its benefits. These authors focus more on doing this to use EC to debug, refactor, and repair programs. It also breifly mentions FINCH .