Evolving Bytecode and Assembly

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

ABSTRACT

Keywords

evolutionary computation, x86 assembly code, Java bytecode, FINCH, automated bug repair

1. INTRODUCTION

Genetic programming (GP) is a methodology, based off of evolutionary biology, that is used to automate solving problems. Traditional GP has been mostly used to solve only specific parts of problems in programs and not full-fledged programs themselves. Orlov et al., [6] purposes a method of applying GP to full-fledged programs. This method only requires the program to be able to be compiled to Java bytecode. Once in Java bytecode GP is applied to the program to solve the desired problem. Eric et al., [8] also apply a similar method with bot Java bytecode and x86 assembly.

Both Orlov et al., [7] and Eric et al., [8] show that this methodology is feasible by solving an array of problems. Orlov et al., [?] focused on automating solving simple programs while Eric et al., [8] focused on automated bug repair in programs.

Section 2 covers the background needed to understand this paper. It contains information on Evolutionary Computation (EC), x86 assembly, and Java bytecode. Section 3 covers the benefits of evolving assembly and bytcode. Section 4 discusses how Orlov, et al., [7] evolve Java bytecode. In section 5 we will cover Eric, et al., similar work on both x86 assembly and Java bytecode. In section 6 we will discuss the results from both Orlov et al., [7] and Eric et al., [8] experiments. In section 7 we will discuss future work that could be achieved with evolving

2. BACKGROUND

2.1 Evolutionary Computation

EC is a field of computer science and artificial intelligence that involves continuous optimization. Optimization in EC is the selection of the best element within a set. Traditionally each element in a set is called an individual, and the set is called the population. A methodology used to apply continuous optimization, to solve problems, is GP. A GP evolves an initial population of programs either until the best solution to the problem is found, or a specified number of generations is reached. To evolve an individual is to make changes to it. The fitness of each individual is the deciding factor on how likely it chosen for evolution. The fitness is a value which indicates how well we think they solve the specific problem. For the research discussed in this paper a higher fitness indicates a more fit individual. One way that individuals are chosen to procreate is through tournament selection. In tournament selection a certain amount of individuals are chosen to compete. The individual with the highest fitness wins and then is selected for procreation. A way that procreating is represented in GP is through the genetic operator called crossover. Crossover is the process of taking two individuals and extracting a section of code from one and replacing it with a section from the other program to form an offspring. Another genetic operator that is used to produce offspring is mutation. Mutation, takes a program and then randomly changes a section of it. Mutation can be used along with crossover to produce offspring. Also, in some cases, the most fit individuals are passed on to the next generation unchanged which is called elitism.

2.2 x86 Assembly & Java Bytecode

need to explain stacks and frames

Through out the rest of the paper when referring to both Java bytecode and x86 assembly I will use the term instruction-level code. Instruction level code consist of operation codes (opcodes) which are each one byte in length. Hover, some opcodes take parameters are multiple bytes long [4, 9].

x86 is a family of backward compatible assembly languages [?] created by Intel and designed for specific computer architectures. This means that it is designed to only run on certain physical machines. This is one of the main differences between assembly and bytcode.

Java Bytcode And The JVM.

Java bytecode "is the intermediate, platform-independent representation of" programs that can compile into it [2] and run on the Java Virtual Machine(JVM). Some programming languages whose source code compiles into Java bytecode are Scala, Groovy, Jython, Kawa, JavaFx Script, Clojure [7], and JRuby. Since Java bytcode runs on a virtual machine this means that it is portable across multiple architectures.

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

```
float x;                         int x=7;
int y=7;                         float y;
if(y>=0){                        if(y>=0){
    x=y;                             y=x;
}else{                               x=y;
    x=-y;                        }
}System.out.println(x);           System.out.println(z);

     (a)                           (b)
```

Figure 1: Going to use this figure as an example for section 3 make this look pretty

```
class SymbolicRegression{
    Number symbolicRegression(Number num){
        double x = num.doubleValue();
        return Double.valueOf((x+x)*x);
    }
}
```

Figure 2: Example of a possible starting program to evolve and solve symbolic regression.

However, this also means that it needs to be interpreted or compiled for the desired architecture. For example, Java bytecode can be compiled to x86 assembly.

The JVM executes bytecode through a stack-based architecture. There is one stack per thread and one frame is created on the stack per method invocation. A stack frame keeps track of local variables, the operand stack, and the current class.

Should I include examples of bytecode and x86?

2.3 Grammar Stuff

might get rid of this section if I don't find other useful information for the subsection

Backus-Naur Form.

Backus-Naur Form(BNF) [3] is a notation technique used for context free grammars (CFG) and is used to represent the syntax of computer programming languages.

3. THE BENEFITS

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 a GP for that specific problem. describe what evolvable is Evolving instruction-level code bypasses both of these problems.

3.1 Source Code 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 BNF which is used to represent the syntax of the programming language [?, 8]. This means that the grammar does not represent the semantic constraints of a program. The BNF form does not capture the languages type system, variable visibility and accessibility, and other constraints [7]. For example, in figure 1 both 1.(a) and 1.(b) comply to the CFG rules of Java. However, when taking into account semantics 1.(b) is illegal code. In 1.(b) variable y is uninitialized during read access in the if statement and setting x to y is incompatible. Plus, variable z is not defined anywhere. 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 creating a full-scale compiler to check for these semantic constraints.

Java bytecode and x86 assembly contain a small set of instructions [8]. Also they are less syntactical and there are less semantic constraints to violate. This means that there is a lot less risk of producing a noncompilable program during evolution.

3.2 Evolve Entire Program As Is

For traditional EC, a GP has to be designed specifically for a problem. This means that that GP will is optimized for that specific problem and might not even work on other problems. Evolving, instruction-level code doesn't require the program to be designed specifically for evolution [8, 7]. Thus, any language that can compile into instruction-level code can be evolved. The only thing required is an algorithm to compute the fitness. Also, the entire program can be evolved. In traditional EC when a GP is designed it is more common to evolve expressions or formulae [7] 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 does not even need to be written to be able to evolve it and find a solution. For example, the code in Figure 2 could be used to solve the symbolic regression problem for a polynomial. Also, the final product can be drastically different from the original program since the entire program was evolved. This allows for more flexibility in evolving programs and allows us to solve the problem as a whole instead of a small section of the problem.

4. FINCH

FINCH is a program developed by Orlov et al., [6, 7] that evolves programs that have already been compiled into Java bytecode.

4.1 Crossover

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

4.2 Selecting Offspring

Evolving Java Byte code only reduces the chance of evolving non-compilable bytecode to a certain extent. Even though byte code is less syntactical, than source code, it still is syntactical. Sipper et al.[6] addresses this issue by checking if a good offspring has been created before letting it join the evolved population. This is done by making sure offspring produced through crossover are valid bytcode. If it is not then another offspring with the same parents is made. This process is repeated until a good offspring is produced or a predetermined number of attempts have been made.

These checks makes sure that stack depth, variable type, and control flow is respected.

Stack and Variable Types.

The following checks are done to assure that the stack and stackframe are both type compatible and stack underflow does not happen. Stack underflow is where an attempt to pop from an empty stack occurs. Stack pops of β must have identical or narrower types and depth as α. Stack pushes of α must have identical or narrower types as stack pushes of β.

Variable Types.

The following checks are done within the bytcode to make sure that by inserting α into β that variables written before and after those sections are compatible with the change. First, variables written by α must have identical or narrower types that are read after β. It is also made sure that all variables read after β and not written by α must be written before β.Finally, all variables read by α must be written before β.

Control flow.

These checks are done to make sure that when jumps within the bytcode is done that it doesn't cause the program to break. First, a check is done to make sure that there are no jumps in β and jumps out of α. Secondly, a check is done to make sure that the code before β transitions into β and that α transitions into code post α.

The checks make sure that all the variables will be read, written, type-compatible, and will not cause stack underflow[2]. For example, in figure 1, if 1.(b) was the result of two parent programs then it would be rejected as a good offspring. This is due to its failure to comply to type compatibility, variable initialization before read access, and variable visibility.

It is important that only good offspring is selected because this provides good variability in the population. Noncompilable 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.

Orlov et al. [7] focused on evolving small programs to solve problems. The five problems they focused on were symbolic regression, artificial ant, intertwined spirals, array sum, and tic-tac-toe. We will discuss their results from the symbolic regression and array sum problems. Their program, FINCH, was able to evolve programs and solve each of these problems.

In each of tests FINCH was given each time a program that had a zero fitness. The elements included in the programs were the minimal components to successfully evolve and solve the problem. For example, in the case of symbolic regression the initial population contained the mathematical function set {+, -, *, %, *sin*, *cos*, *e*, *ln*} and number types.

4.3 Non-Halting Offspring

Another issue that arises from evolving unrestricted bytcode is that the resulting program might enter a non-halting state. These problems do not arise when the check to see if an offspring is good bytecode. Instead, this is a runtime issue. This, is especially true when evolving programs that contain loops and recursion. The way that Sipper et al.[7] deal with this, before running the program, is count how many calls are made to each function. If too many calls are made to a function than an exception is thrown.

5. EVOLVING ASSEMBLY CODE

Eric, et al, focus evolving x86 and Java Bytecode for the purpose of program repair and debugging. In their tests they took medium to large sized programs in Java, C, and Haskell that contained a bug. The types of bugs they used were common human errors such as having a for loop index off by one. Most of their experiments focused on evolving a small section of the program.

5.1 Fitness and Selecting Good Offspring

In selecting offspring a different path was chosen than Olov, et al. They decided to not make sure produced offspring were compilable. Instead they decided to let all produced offspring into the next generation. This produced a considerable amount of individuals with fitness zero due to being noncompilable.

Each program to be evolved was provided with a set of tests that passed and one test that failed. The failed test was used to check if an offspring fixed the bug. The tests that passed were used to make sure the program retained functionality. For each offspring they compiled them into either an executable binary(x86) or class file (Java). If the program failed to compile then it would obtain a fitness of zero. If the program did compile then it was ran with the tests. The fitness score is calculated as the weighted sum of tests passed, the negative test being worth more. This a great weight was placed on the non passing testing since that was the main goal.

5.2 Genetic Operators

Eric, et al., used mutation on 90% of each population and crossover on the rest to produce the offspring population. Multiple tournaments consisting of three individuals were performed to select fit individuals for reproduction. They used mutation over crossover 90% of the time because it produced better results for the type of problems they were solving. Since, each bug was a minor change, such as change a zero to a one, using a large amount of crossover or more complex operators generally slowed their search time.

5.3 Non-Halting Offspring

Eric, et al., also chose a different approach than that of Olav, et al., when dealing with non-halting offspring. They decided to not check for non-halting and instead run the individual on a virtual machine(VM) with an eight second timeout on the process. Olov, et al., decided against this due to two main issues. First it is hard to define how long a program should run which can vary greatly depending on the cpu load. Thus if good time limit is not selected then it can be waste to cpu resources. The second issue is that limiting the runtime requires running the program on a separate thread. Killing the thread can be unreliable and can be unsafe for the GP as a whole. Eric, et al., ran into the problem where sometimes programs would not respond to termination requests. They also ran into ”Stack Smashing” which is where the stack of the application or operator overflow which can cause the program and system as a whole to crash. However, since they ran each individual on a VM stack smashing was not a significant problem.

6. RESULTS

6.1 FINCH

Orlov et al. [7] focused on evolving small programs to solve problems. The five problems they focused on were symbolic regression, artificial ant, intertwined spirals, array sum, and tic-tac-toe. We will discuss their results from the symbolic regression and array sum problems. Their program, FINCH, was able to evolve programs and solve each of these problems.

In each of tests FINCH was given each time a program that had a zero fitness. The elements included in the programs were the minimal components to successfully evolve and solve the problem. For example, in the case of symbolic regression the initial population contained the mathematical function set {+, -, *, %, *sin*, *cos*, *e*, *ln*} and number types.

6.1.1 Symbolic Regression

Symbolic regression is a method of finding a mathematical function that best fits a a set of points between two intervals. Olov, et al., [7] chose to use 20 random points, between -1 and 1, from various polynomials. Fitness was calculated as the number of points hit by the function. The function set {+, -, *, %, sin, cos, *e*, *ln*}(note: referring to *e* as a function in the Java.lang.Math library) was used for most of their experiments. This was done in order to mimic previous experimentation done [5] in order to compare the results to traditional GP. They were able to evolve up to a 9-degree polynomial with just the function set {+, *}.

A more complex fitness was also tested. This symbolic regression consisted of a fitness that checked if all twenty points were within 10^{-8} degree of they polynomials output.

For each experiment they started off with an offspring of fitness zero and usually with a minimal amount of code, such as in figure 2. They evolved the programs with a 90% crossover. Using the simple fitness algorithm 99% of the time a maximum fitness individual was found. With the more complex fitness algorithm a maximum fitness individual was found 100% of the time.

6.1.2 Array Sum

The array sum problem consists of adding up all the values in an array. This problem is important because it would require evolving a loop or recursion to solve it. This would show that FINCH is capable of evolveing more complex programs

FINCH was able to solve this problem quite easily though recursion and a for loop. This experiment also showed that FINCH is able to evolve different list abstractions such as List and ArrayList.this section in the papscannerer is short. Should I include some their code to show their results and as an example?

6.2 x86

looking into other research/papers by this group to get more to write about.

Eric,et al., were able to show that it is possible to fix human programming errors though bytecode and assembly. They were able to successfully debug various programs containing bugs such as infinite loops and remote buffer overflows.

Even though they produced a considerable amount of offspring with fitness zero this did not seem to hurt their result very much. The average number of fitness evaluations required to produce an offspring that passed all the tests was 63.6 for C and 74.4 for assembly. This indicates that not much more work is needed to to evolve repairs in assembly. Even programs that contained thousands of lines of code only required a few runs. This shows that evolving programs in assembly is feasible. include weighted path?

It was also discovered that most repairs only required a small number of operations. This was also due to optimization of mutation.not much info is provided on this

6.3 Future Work

7. CONCLUSION

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] *A Field Guide to Genetic Programming.* 2008. *////This will prived general definitions/descriptions for EC terminology. //.*

[2] *Genetic Programming Theory and Practice VIII.* Springer New York, 2011.

[3] Backus-naur form. January 2014.

[4] Java bytcode. Febuary 2014.

[5] J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* A Bradford book. Bradford, 1992.

[6] 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.

[7] M. Orlov and M. Sipper. Flight of the finch through the java wilderness. *Evolutionary Computation, IEEE Transactions on*, 15(2):166–182, April 2011.

[8] 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.

[9] Wikibooks. X86 assembly/machine language conversion — wikibooks, the free textbook project, 2013. [Online; accessed 23-March-2014].