Evolving Bytcode and Assembly

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

ABSTRACT

evolutionary computation, x86 assembly code, Java bytecode

Keywords

1. INTRODUCTION

The limitation of being able to only evolve small parts of programs is an issue in Evolutionary Computation (EC). Much of these constraints are due to 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 due to syntactical constraints.

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

In section 2 I will cover the background needed to understand this paprt. It will contain information on Evolutionary Computation (EC), x86 assembly, and Java bytecode. In section 3 I will cover the reasons for evolving assembly and bytcode and its advantages over source code. In section 4 I will cover how Olov, et al., [5] program the FINCH works and evolves Java bytcode. In section 5 I will cover Eric, et al., similar work on both x86 assembly and Java bytecode.

2. BACKGROUND

2.1 Evolutionary Computation

Evolutionary Computation (EC) is a field of computer science and Artificial Intelligence that uses many of the concepts of biological evolution to evolve programs. EC is useful because it allows us to not know the structure of a solution in order to find it.

Genetic Programming(GP) consists of evolving an initial population of programs either until the best solution to the problem is found, or the maximum number of generations is reached. Each individuals is rated with a fitness value. The fitness value determines how likely the individual is going to procreate or proceed to the next generation. 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

x86 Assembly.

X86 assembly is a low level programming language designed for specific computer architectures. This means that it is designed to only run on certain physical machines. Assembly instructions consist of operands that are a byte or more in size which are called operation codes (opcode).

Java Bytcode.

Java bytecode is compiled source code that runs on the Java Virtual Machine(JVM). The programming langauges whos source code compile into Java Bytecode are Scala, Groovy, Jython, Kawa, JavaFx Script, and Clojure [5]. Java bytecode "is the intermediate, platform-independent representation of Java programs [1]." Since Java Bytcode runs on a virtual machine this means that it is portable across multiple architectures. However, this also means that it needs to be interpreted or compiled for the desired architecture. For example, Java Bytcode can be compiled to x86 assembly. Many bytecode opcodes are one byte in length. However, some opcodes, that take parameters, are multiple bytes long [3].

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.

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

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

(a)

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

(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.

Backus-Naur Form(BNF) [2] is a notation technique used for context free grammars (CFG) and is used to represent the syntax of computer programming languages.

3. WHY EVOLVE MACHINE CODE?

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 evolvabledescribe what evolvable is. 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 BNF which is used to represent the syntax of the programming language. 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 [5]. 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 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 writing a full-scale compiler.

Java bytecode and x86 assembly contain a small alphabet of instructions [6]. Also they are less syntactical and there is less statistical 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 program has to be designed specifically to be evolved. Evoling, bytcode or x86 doesn't require the program to be designed specifically for evolution [6] [5]. Thus, any language that can compile into either bytecode or x86 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 program is designed it is more common to evolve 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 does not even need to be written to be able to evolve it and find a solution. Such as 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 a lot more flexibility in evolving programs. This also allows us to solve the problem as a whole instead of a small section of the problem.

4. HOW THE FINCH WORKS

4.1 Variable Access Sets

[4] much hard
In progress. Will probably move this into 4.2

4.2 Selecting Good 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.[4] addresses this issue by checking if a good offspring has been created before letting it join the evolved population. This is 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.

The checks that are made make sure that all the variables will be read, written, type-compatible, and will not cause stack underflow[1]. 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.

4.3 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.

4.4 Non-Halting Offspring

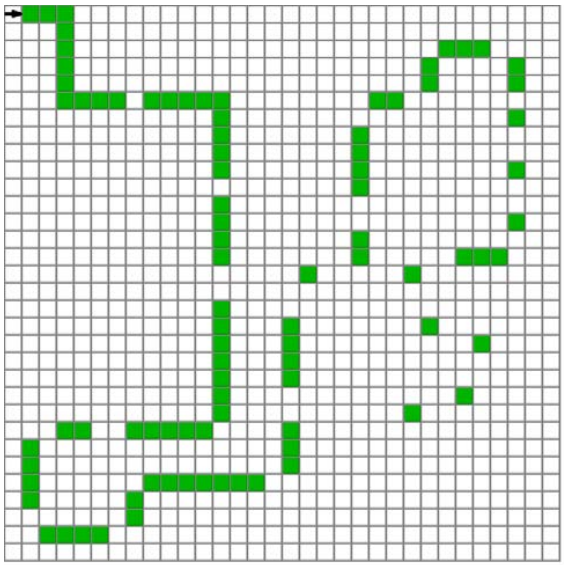

Figure 3: Artificial Ant Grid

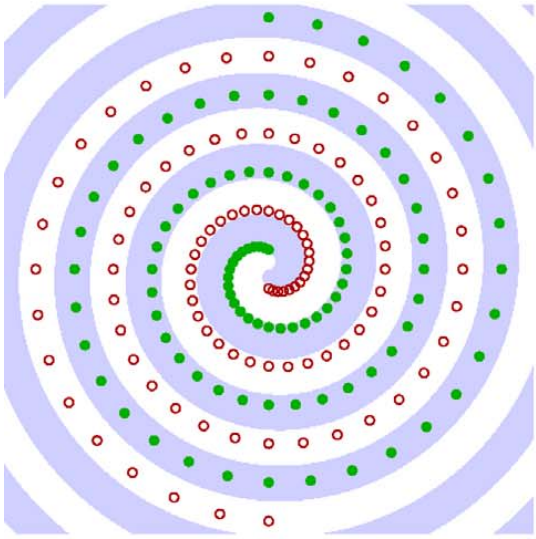

Figure 4: 194 points on a Spiral

Another issue that arises from evolving unrestricted bytcode is that the resulting program might enter a non-halting state. These problems don't 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.[5] 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 human prone bugs such as a for loop index off by one. Most of their experiments focused on evolving a small section of the program.

5.0.1 Fitness and Selecting Good Offspring

However, since they ran each individual in a VM this was not a problem. 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.

5.0.2 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.

5.0.3 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., voted 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 huge not a problem.

6. RESULTS

6.1 FINCH

Orlov et al. [5] 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. 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. Usually the only stuff included in the programs was 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 double objects.

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., 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 do I need to provide a reference here? It is a reference from their paper. However, they were able to evolve 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 Artificial Ant

The artificial ant problem is a problem that involves moving an ant on a square grid trying to find all the food on a grid. Figure 3 is an example of the grid. The ant starts off on the upper right hand corner of the grid facing to the left and the food is represented as green squares. The terminals used to control the ant are LEFT,RIGHT, and MOVE. LEFT and RIGHT turn the ant 90 degrees to its left and right respectively. The move function moves the ant forward. Additional functions that can be used is IF-FOOD-AHEAD,PROGN2,and PROGN3. IF-FOOD-AHEAD is a test which checks if there is food in front of the ant. PROGN2 and PROGN3 allow for two and tree sequence operations respectively.

The fitness was calculated as the number of food squares consumed. A restriction of 100 non-eating moves was placed so that FINCH would not evolve a program that traversed the entire grid.

For this experiment they also started off with an individual of zero fitness. The population was evolved with a 90% chance of crossover. For the population sizes of 500, 2,000, and 10,000 a maximum fitness individual was found 2%, 11%, and 35% of the time respectively.

6.1.3 Intertwined Spirals

The intertwined spirals problem involves creating a mathematical equation that best fits 194 point on two spirals as shown in figure 3. For this problem Orlov et al. [5] used mutation instead of crossover look into Gaussian-distributed random value for mutation probability. The function set used was {-,+,*,$arctan^2$,sine, and hypotenuse(x,y)}. Sine and $arctan^2$ were used to facilitate producing a spiral like equation. The fitness was calculated as the number of points correctly classified. a population of 2000 individuals was used with a total of 251 generations.

10% of the time an individual with a maximum fitness was found. When further testing the ten best-of run individuals against sample points twice and ten times as dense FINCH produced better results than previous experiments done by Kozashould i include this? if so get citation who used more tradition EC methods. For seven of the individuals all the points on both denser sets matched 100%. For the other 3 individuals it was 99%.

6.1.4 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 paper 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] *Genetic Programming Theory and Practice VIII.* Springer New York, 2011.

[2] Backus-naur form. January 2014.

[3] Java bytcode. Febuary 2014.

[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.

[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.

[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.