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

Evolutionary Computation (EC) is a methodology, based off of evolutionary biology, that is used to solve problems. Similarly Genetic Programming (GP) is used to solve programs as the problem. However, traditional GP has been used mostly 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 a 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 bytcode. We will cover similar work in section 5 by Eric, et al., 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

In this section we will explain what EC and GP, their components, and how they are used. We will also discuss what x86 and Java bytecode is. Additionally we will also go over basic details of the JVM.

2.1 Evolutionary Computation

EC is a field of computer science and artificial intelligence that involves continuous optimization to solve problems. 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 programs as the problem, is GP. A GP evolves an initial population of programs either until the desired 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 is 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, where a certain number of individuals are chosen to compete and the individual with the highest fitness wins and then is selected for procreation. A way that procreating is implemented 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

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

2.2.1 X86

x86 is a family of backward compatible low-level programming 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 x86 assembly and Java bytcode.

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: Both (a) and (b) are valid code syntactically however only (a) is valid semanticly.

```
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.

2.2.2 *Bytcode And The JVM*

Java bytecode "is the intermediate, platform-independent representation" [1] originaly designed for the Java compiler. However, other high-level languages now also have compilers for the Java Virtual Machine (JVM). Some of those languages are Scala, Groovy, Jython, Kawa, JavaFx Script, Clojure [7], Erjang, and JRuby.

The JVM executes bytecode through a stack-based architecture. Each JVM thread has a private stack, and each stack stores frames. Each frame contains an array of local variables, its own operand stack, and a reference to the class of the current method. Only the frame of the current executing method can be active at a time. Stack can only be popped and pushed to.

Figure 3 is a simple example of what Java bytecode looks like and how the stack works. In this example we are assuming that the frame already contains the integers 1 and 3 to retain simplicity. After iload_1 is executed it takes the element from the frame at index 1 and pushes it onto the stack. Iload_2 does the same thing but with index 2. Iadd pops to elements from the stack, which both must be of type int, and then adds them and pushes the result on the stack. Ireturn simply pops the stack and returns that element.

```
Bytcode    Frame1

iload_1    class
iload_2    1
iadd       3
ireturn


Execute iload_1  Execute iload_2

    Stack            Stack
      1                3
                       1

Execute iadd     Execute ireturn

  Stack            Stack(empty)
    4
```

Figure 3: Istore_n pushes an element from index n from frame onto stack. Iadd pops two elements from operand stack add them together and then pushes to the result to the stack. Ireturn pops an element from the stack and returns it

3. INSTRUCTION-LEVEL CODE 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. 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 using grammars which are used to represent the syntax of the programming language [2, 8]. This means that the grammar does not represent the semantic constraints of a program. The grammar 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 syntactical rules of Java. However, when taking into account semantics 1.(b) is illegal code. In 1.(b) variable `y` is uninitialized before the test in the if statement, and assigning `y` to `x` violates a type constraint. 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. Thus, it is easier to evolve instruction-level code.

Instruction-level code consist of a small set of instructions [8]. They are simpler syntactically and there are less semantic constraints to violate. This means that there is a lower risk of producing a noncompilable program during evolution.

3.2 Evolving As Is

In tradition EC when creating a GP to evolve a program there is a lot of scaffolding that has to be done. Usually we have to design the GP for that specific program. When a GP is designed it is common to just 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. The GP has to know what it can evolve and where it is located. Also, usually the GP isn't useful in evolving other programs to contain different problems.

```
1.   iconst_1    iconst_4    iconst_4    iconst_4
2.   iconst_3    iconst_2    iconst_2    istore_1
3    istore_1    istore_1    istore_1    istore_2
4    istore_2    istore_2    istore_2    istore_2
5.   iload_1     iload_1     iload_2     iload_2
6.   iload_2     iload_2     iload_2     iload_1
7.   iadd        isub        isub        isub
8.   ireturn     ireturn     ireturn     ireturn
9.
       (A)         (B)         (C)         (D)
```

Figure 4: This is an example of good and bad crossover. iconst_n pushes onto the stack an int of value n. istore_n pops the stack and saves the value on the frame at index n. Look at figure 3 for a description of iload,iadd,isub,and ireturn.

Instead an entirely new GP would have to be designed and the program along with it.

When evolving instruction-level code there is no requirement for the program being evolved to be designed for the GP. The GP doesn't have to focus on a specific part of the program or even be aware of what is in the program for the most part. Instead the program that is being evolved stays as is and the GP goes through minor modifications to fit the problem trying to be solved. The GP has to be given a fitness algorithm in order to attempt to find a desirable individual. It also needs to know what genetic operator and the desired number of generations is needed. Depending on the on the program being evolved one might also have to put a growth limit on the code or implement protected division or logarithm. For example, Orlov et al., [7] enforced a growth limit when performing crossover on code so that the programs would not become too large. The initial population can also almost contain no code in it in order for the GP to find a desirable solution.

A huge benefit that comes from most of the scaffolding happening on GP end is that the initial program being evolved does not have to contain a whole lot of code. The GP can evolve a minimal amount of code into a full-scale working program. However, evolving instruction-level code is also capable of being very focused like traditional GP is on source code. For example, Eric et al., [8] in their research focused on only evolving small areas of code and would only make minor changes to the code as a whole. The result code would usually only have one line of code changed.

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 Java bytecode. It takes two programs A and B and extracts sections α and β 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.

For example in Figure 4 let the Java bytecode snippets 4(A) and 4(B) be A and B. Let α be line 6 from A and β be line 5 from B. A(C) is the product of replacing β with α in B.

Now let α be lines 3-4 in A and β be lines 2-3 in B. A(D) is the result of replacing β with α in B. However in this case A(D) is incorrect crossover since only one integer is saved on the frame and at line 3 istore_2 tries to pop an empty stack causing stack underflow.

4.2 Selecting Offspring

Evolving Java Byte code only reduces the chance of evolving non-compilable bytecode to a certain extent. Even though Java byte code has a simpler syntax than source code, it still has syntactical constraints. Orlov et al.,[6] address this issue by checking if a good offspring has been created before letting it join the evolved population, thus ensuring offspring produced through crossover contain valid bytcode. If an illegal offspring is produced 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[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 due to its failure to comply to semantical constraints stated Section 3.1.

It is important that only good offspring is selected because this provides good variability in the population. Good variability is where there are many individuals that vary in code and fitness. 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 popu-

lation to have a zero fitness score. This would possibly result in having individuals with a zero fitness being selected for procreation and thus consistently resulting in bad crossover and bad offspring.

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 Orlov 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. The lowest possible fitness is assigned to an individual who fails this test.

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.

They use three mutation operators mutate-insert selects an instruction based of off its positive weight. Mutate-delete selects and instruction by proportionally by negative weight. Mutate-swap operator select to instruction proportionally by negative weight and swaps them. They also implement a simple version of crossover that swaps sections from two programs creating two new offspring.

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, math.*e*, *ln*} 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 {+, *}.

```
int sumlistrec(List<Integer> list) {
  int sum = 0;
  if (list.isEmpty())
    sum *= sumlistrec(list);
  else
    sum += list.get(0)/2 + sumlistrec(
           list.subList(1, list.size()));
  return sum;
}
```

Figure 5: FINCH's solution to the intial program presented in Figure 5.

```
int sumlistrec(List list) {
  int sum = 0;
  if (list.isEmpty())
    sum = sum;
  else
    sum += ((Integer)list.get(0)).intValue() +
        sumlistrec(list.subList(1,
            list.size()));
  return sum;
}
```

Figure 6: Initial population function, for array sum, that enters into infinite recursion in the if statement.

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. Figure 2 only contains a simple function set of +, and a number object. However they were able to evolve programs like this to full-fledged programs that solved symbolic regression of up to 9-degree polynomials. 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 evolving 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.

When evolving array sum with recursion the initial population consisted of an individual which entered infinite recursion as shown in figure 5. Due to the way FINCH deals with non-halting programs this wasn't a problem and evolution was able to ensure. The resulting code from evolving Figure 5 is shown in Figure 6.

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. The interesting thing about these experiments was that it was done on real programs with real buts and at times the programs consisted of thousands of lines of code. Another interesting thing that they discovered was that there were some bug repairs the they were able to solve through x86 assembly that were not possible in the source code.

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.

It was also discovered that most repairs only required a small number of operations. This was also due to optimization of mutation.

6.3 Future Work

Future work that could be done, thought evolving instruction-level code is trying to evolve programs who's solutions are much longer and complicated. Even though Orlov et al., proved that using bytecode evolution it is possible to solve many simple problems. Most of the problems only consisted of a few lines of code.

For the work done by Eric et al., it would be nice to see debugging done on less focused areas. Possibly a program as a whole. This would be a good development since it would greatly reduce how much scaffolding is required to set up the evolution.

7. CONCLUSION

There are things that evolving at the instruction-level can be done that can not be done at the source code level. Benefits such as evolving the program as a whole and reduced scaffolding to the GP, and no scaffolding for the program being evolved. Eric et al., and Orlov et al., showed that evolving instruction-level code is just as good as source code and at times even better due to the fact there some things that you can only do at the the instruction-level. In conclusion, evolving instruction-level is feasible and exiting for the field of EC since it opens up many possibilities that were not always available.

8. ACKNOWLEDGMENTS

Nic Mcphee, Elena Machkasova

9. REFERENCES

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

[2] *The Java Language Specification.* Oracles America, Inc. And/or affiliates, 500 Oracle Parkway, Redwood City, California 94065, U.S.A., 2013.
////This provides some needed background on the JVM, compiler, and Java bytecode. It is too long to read it all but will be a good //reference for any questions about the JVM //.

[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].