Applying Genetic Programming to Bytecode and Assembly

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

ABSTRACT

Traditional genetic programming (GP) has yet been able to evolve entire programs at the source code level. Instead only small sections within in the programs are usually evolved. A proven method that solves this problem is evolving programs in either bytecode or assembly language. This paper provides an overview of applying genetic programming to Java bytecode and x86 assembly to solve problems. This paper explores two examples of how this method can be implemented. This paper also discusses the experimental results that show that using this method is feasible.

Keywords

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

1. INTRODUCTION

GP is a set of techniques used to automate computer problem solving. This is done by evolving programs with an evolutionary algorithm (EA) that imitates natural selection, to find a solution. Traditional GP has been used mostly to evolve only specific parts of programs and not full-fledged programs themselves. This is because traditionally we have to understand the structure of the program being evolved. An EA is usually designed for a specific program with knowledge about how the program works. Being able to evolve an entire program without knowing its structure would allow more flexibility in GP. A method that allows for this flexibility is evolving bytecode and assembly instead of source code. This is possible because bytecode and assembly languages are less restrictive syntactically than source code. This will be further explained in Section 3.

Orlov et al., [5] propose a method of applying GP to full-fledged programs which only requires a program to be compiled to Java bytecode. Once in Java bytecode an EA is applied to the program to solve the desired problem. Eric et al., [7] also apply a similar method with both Java bytecode and the x86 assembly family.

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

These methods are important because they show that evolving entire programs is possible, which has yet to be done with traditional GP in source code. This is also useful since once the EA has been created there is little modification that has to be done to it to evolve different programs and the structure of the program being evolved does not have to be known or modified.

This paper will examine how both Both Orlov et al., [6] and Eric et al., [7] apply GP to solve programs in Java bytecode and x86 assembly. We will also show that this meintentionallythodology is feasible by solving an array of problems in Section 6. Orlov et al., [6] focused on evolving simple programs as a whole while Eric et al., [7] focused on automated bug repair in programs.

This paper is organized as follows. Section 2 covers the background needed to understand this paper. It contains information on Evolutionary Computation (EC), x86 assembly, and Java bytecode. Section 3 describes the benefits of evolving assembly and bytecode. Section 4 discusses how Orlov, et al., [6] evolve Java bytecode. Similarly, in Section 5 we discuss how Eric, et al., evolve both x86 assembly and Java bytecode. In section 6 we report some of the problems solved, using this method, by Orlov et al., [6] and Eric et al., [7]. In section 7 we will address possible future work and ideas.

2. BACKGROUND

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

2.1 Evolutionary Computation

EC is a field of computer science and artificial intelligence that is loosely based off of evolutionary biology. EC imitates evolution through continuous optimization to solve problems. Optimization in EC is the selection of the best individual within a population. For example, Figure 1 shows the process of evolution in EC. An initial population of individuals is taken and a selection process is done to choose the most fit individuals. The most fit individual are then taken and modified with genetic operators that imitate procreation between two individuals. Then a check is done to see if any individuals from the population solve the desired problem. If not then the process of evolution is repeated. If the problem is solved then the individual with the best solution is returned.

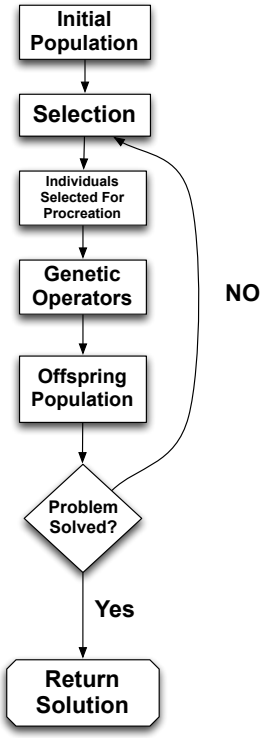

Figure 1: The process of of Evolutionary Computation

GP is tool that uses the EC process to evolve programs. 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. The fitness of each individual is the deciding factor on how likely it is chosen for evolution. The fitness is a value, usually numerical, 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. The selection of individuals for procreation is done through tournament selection. This is where a certain number of individuals are chosen to compete and the individual with the highest fitness wins and then is selected for procreation. One way that procreating is simulated 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; a new program. 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 the term instruction-level code will be used 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 [9, 8].

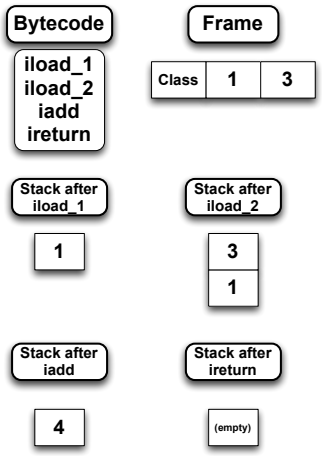

Figure 2: Index zero of the frame is a reference to the methods current class. Istore_n pushes an element from index n from frame onto stack. Iadd pops two elements from operand stack add them together and then pushes to the result to the stack. Ireturn pops an element from the stack and returns it

2.2.1 X86

x86 is a family of backward compatible low-level programming languages [10] 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 bytecode.

2.2.2 Bytecode and the JVM

Java bytecode "is [an] intermediate, platform-independent representation" of Java source code [5]. Even though it was originally designed for the Java compiler other high-level languages now also have compilers for the JVM. Some of those languages are Scala, Groovy, Jython, Kawa, JavaFx Script, Clojure [6], Erjang, and JRuby.

The JVM executes bytecode through a stack-based architecture. Each JVM thread has a private stack, which can only be popped and pushed to, stores frames. Each frame contains an array of local variables, its own operand stack, and a reference to the class of the current method[1]. A new frame is created when a method is invoked. When a method is done executing the frame is destroyed [1]. Only the frame of the current executing method can be active at a time.

Figure 2 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 local variables 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 two 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.

[1]It is a reference to the run time constant pool of the class of the current method. More details can be found in chapter 2.5.5 of [2]

```
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 3: Both (a) and (b) are valid code syntactically however only (a) is valid semantically.

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 know and understand the structure of the program in order to design an EA for that specific problem. At times we even have to modify the program so that it can be evolved. Evolving instruction-level code can 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 [1, 7]. 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 [6]. For example, in Figure 3 both 2(a) and 2(b) comply to the syntactical rules of Java. However, when taking into account semantics 2(b) is illegal code. In 2(b), variable `y` is uninitialized before the test in the if statement, and assigning `y` to `x` violates a type constraint. Also, 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 consists of a smaller set of instructions [7]. It is 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 Flexibility

Conventionally in EC, when creating an EA to evolve a program, we have to understand the structure of the program in order to evolve it. Usually an EA is designed for a specific program. When an EA is designed it is common to just evolve expressions or formulae [6] within the program. Most of the program would already be written and remain the same after evolution. The EA has to know what it can evolve and where it is located. This limitation is due to the difficulty of dealing with semantic constraints in source code, as previously discussed. Also, usually the EA isn't useful in evolving other programs that contain different problems. Instead, an entirely new EA would have to be designed.

When evolving instruction-level code the program does not have to be "intentionally written for the purpose of serving as an EA representation" [5]. Furthermore, an understanding of the structure of the program is not required. The only requirement is that it can be compiled to instruction-level code [5, 7]. The EA doesn't have to focus on a specific part of the program in order to evolve it. Instead the program that is being evolved stays as is and the EA goes through minor modifications to fit the problem trying to be solved. The EA 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 program being evolved one might also have to put a growth limit on the code. For example, Orlov et al., [6] enforced a growth limit when performing crossover on code so that the programs would not become too large.

A huge benefit that comes from evolving instruction-level code is that the initial program being evolved does not have to contain a whole lot of code. This is partially due to the fact that a small program can contain a large amount of instructions-level code [7]. The EA can evolve a minimal amount of code into a full-scale working program. However, when evolving instruction-level code it is also possible to evolve a very focused area within the program; like traditional GP does in source code. For example, Eric et al., [7] designed their EA to only evolve smaller areas of code and only made minor changes to the code as a whole. The resulting code would usually only have one line of code changed. However, since they were evolving instruction-level code, they were able to fix bugs such as incorrect type declarations, which was not possible in their earlier work [3] done in source code.

4. FINCH

FINCH is a program developed by Orlov et al., [5, 6] that evolves programs that have already been compiled into Java bytecode. FINCH also makes sure that all offspring produced are executable Java bytecode through a set of strict rules.

4.1 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.,[5] 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 bytecode. 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.

Let α and β be sections of code of two separate programs on which crossover is being applied. The following constraints in crossover are applied to assure that good offspring are produced:

The stack and stackframe must be type compatible. The

Line #	(A)	(B)	(C)	(D)
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

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 2 for a description of iload,iadd,isub,and ireturn.

the stack must have enough elements on it so that stack underflow does not occur. Stack underflow is where there is an attempt to pop from an empty stack. This is done by assuring that stack pops of β have identical or narrower types as α, and stack pushes of α must have identical or narrower types as stack pushes of β.

When Inserting α into β variables written before and after those sections must be compatible with the change. Variables written by α must have identical or narrower types that are read after β. Also, 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.

All jumps within the bytecode can not cause the program to break. There must be no jumps into β and no jumps out of α since there is a high probability that it would break the code. Also, the code before β must transition into β and α must transition into post α.

These checks, simply put, make sure that all the variables will be written before read, will be type-compatible, and will not cause stack underflow[5]. In Figure 4, for example, FINCH would not accept 3(D) as a good offspring since it would fail the checks for stack and frame compatibility. These checks are important because they ensure good offspring and variability in the population. Good variability indicates many individuals that vary in code and fitness.Need to explain why variability is important. Look at FINCH 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. 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.

4.2 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. 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. This is considered good crossover because the value at index 2 on the frame can be called twice. It is also an int type which keep isub from breaking. If we 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.

4.3 Non-Halting Offspring

Another issue that arises from evolving unrestricted bytecode 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., [6] 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 then an exception is thrown. The lowest possible fitness is assigned to an individual who fails this test.

Orlov et al., chose to count the calls to each function before running the code to prevent from having to either run it on a separate thread or set a run time limit. They decided against running each program on a separate thread because killing a thread can be unreliable and be unsafe for the GP as a whole. They also decided against setting a time limit due to the difficulty of defining how long a program should run. How long a program runs could vary greatly depending on the program being run and the CPU load. Also, this would limit the search space since an individual that is a desired solution could run longer than the time limit.

5. EVOLVING ASSEMBLY CODE

Eric, et al., [7] focused evolving x86 assembly 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 Selecting Offspring

Schulte et al., [7] chose to not make sure their offspring were valid instruction-level code. Instead they decided to let all produced offspring into the next generation. This produced a considerable amount of individuals with a fitness of zero due to being noncompilable.

Test cases were used to calculate the fitness of each individual. The test cases consisted of a set of passing tests and one failing test. The failing test was used to check if an offspring fixed the bug. The passing tests were used to make sure the program retained functionality. Each offspring was compiled into either an executable binary(x86) or class file (Java). If the program failed to compile it obtained a fitness of zero. If the program did compile it was run against the tests. The fitness score was calculated as the weighted sum of tests passed, the test that initially failed being worth more. A greater weight was placed on the non passing testing since that was the main goal.

5.2 Genetic Operators

Eric et al., [7] 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. Mutation was used over crossover 90% of the time because it produced better results for the type of problems being

solved. Since each bug was only required a minor change, such as changing a zero to a one, using a large amount of crossover or more complex operators generally lengthened their search time.

Many of the programs being evolved where very large; consisting of thousands of lines of instruction-level code. Because of this [7] used a "weighted path" to select what lines of instruction to apply mutation and crossover too. Each line of instruction-level code was given a weight. The weight was calculated by checking what tests executed that instruction. This weight was used to indicate how relevant the that line of code was too the bug. A path weight of 1.0 was assigned if the instruction was only executed by the negative test case. A weight of .1 was given if the instruction is executed by both the negative test case and atleast one positive test case. For all other cases a path weight of 0 is given.

Three mutation operators were used in the experiments: mutate-insert, mutate-delete, and mutate-swap. Mutate-insert selected an instruction based of off its positive weight. Mutate-delete selected and instruction based off it's negative weight and deleted it. Mutate-swap operator selected two instructions based off of their negative weight and swapped them. The probability of mutation for each path was calculated by multiplying the mutation rate and the weighted path. The higher the product the more likely that path was chosen for mutation. Since paths that were not executed by the negative test case received a weight of zero there was zero chance of them being selected for mutation.

A simple version of crossover was also implement that swapped sections from two programs and then would create two new offspring.

5.3 Non-Halting Offspring

Schulte, et al., [7] also chose a different approach than that of Orlov, et al., [6] when dealing with non-halting offspring. They decided to not check for non-halting cases and instead run each individual on a virtual machine (VM) with an eight second timeout on the process. A problem with programs not responding to termination requests was noted. There were also issues with stack buffer overflow which occurs when there is an attempt to write to a memory address outside of data structures size. This can cause the program and system as a whole to crash. However, this was expected since one of the bugs that Eric et al. [7] were trying to fix was buffer overflow. Since each individual was run on a VM, buffer overflow was not a significant problem.

6. RESULTS

Both Orlov et al., and Schulte et al., [6, 7] where able to evolve programs successfully at the instruction-level. The different designs of their EAs was due to the type of problems they were trying to solve. Orlov et. al., [6] focused on evolving simple programs that performed a specific task while Schulte et al., [7] focused on evolving programs to fix bugs within them.

6.1 FINCH

The five problems that Orlov et al. [7] 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. The researchers program, FINCH, was able to evolve programs and solve each of these problems.

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

Figure 5: Example of a possible starting program to evolve and solve symbolic regression.

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

In each test FINCH was given 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, if the problem consisted of adding all the elements in an array than a loop or a recursive call was provided along with one variable of each type needed.

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., [6] 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}[2][3] was used for most of the experiments. This was done in order to mimic previous experimentation done [4] and to compare the results to traditional GP.

Each experiment started off with an offspring of fitness zero and usually with a minimal amount of code, such as in Figure 5. Figure 5 only contains a simple function set of {+,*} and a number object. However, it was found possible to evolve such programs to full-fledged programs that solved symbolic regression of up to 9-degree polynomials. This is a good example of how little of the original program needs to be written to be able to evolve it and find a solution.

The authors evolved programs with a 90% crossover by using the simple fitness algorithm 99% of the time a maximum fitness individual was found. With a more complex fitness algorithm a maximum fitness individual was found 100% of the time.

6.1.2 *Array Sum*

[2] e is not being referred as the constant but as the function in the java.lang.Math library.

[3] % and ln are protected division and logarithm. This means that if there is something like division by zero the EA avoids the computation and outputs some pre-selected value. Orlov et al. [6] do not specify how they deal them.

```
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 7: FINCH's solution to the intial program presented in Figure 6

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 6. Due to the way FINCH deals with non-halting programs this wasn't a problem and evolution was able to ensue. The resulting code from evolving Figure 6 is shown in Figure 7.

6.2 Automated Bug Repair

Eric,et al., were able to show that it is possible to fix human programming errors though bytecode and assembly. The authors 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 bugs and at times the programs consisted of thousands of lines of code. Some of the bugs that they were able to fix through instruction-level code were not possible in their previous work [3] with source code. This shows that a wider array of bugs can be fixed by using instruction-level code.

Although a considerable amount of offspring with a fitness of zero were produced this did not seem to damage the result. 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 computational work is needed to to evolve repairs in assembly. Even programs that contained thousands of lines of code only required a few runs. Thus, evolving programs in assembly is feasible.

6.3 Future Work

Future work that could be done, through evolving instruction-level code is to evolve programs whose solutions are much longer and complicated. Orlov et al., [6] proved that by applying GP to bytecode it is possible to solve many simple problems. However, most of the problems only consisted of a small number of lines of code. It would be exciting to see something large and more complex be evolved.

An extension of future work, using the research by Eric et al., could include debugging on less focused areas. For example, attempting to fix bugs that require more fixes throughout the code. Also, there is the question of how applicable automated bug repair is in a real world situation. In most real world scenarios test coverage is very minimal and would rarely cover the entire code base.

7. CONCLUSION

There are things that can be at the instruction-level that are a lot simpler than at the source code level. Things including evolving programs as a whole and reduced assumptions, of the program being evolved, by the EA, 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 exciting for the field of EC since it opens up many possibilities that were once unavailable.

8. ACKNOWLEDGMENTS

Nic Mcphee, Elena Machkasova

9. REFERENCES

[1] *The Java Language Specification*. Oracles America, Inc. And/or affiliates, 500 Oracle Parkway, Redwood City, California 94065, U.S.A., 2013.

[2] *The Java Virtual Machine Spefication*. Oracles America, Inc. And/or affiliates, 500 Oracle Parkway, Redwood City, California 94065, U.S.A., 2013.

[3] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 947–954, New York, NY, USA, 2009. ACM.

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

[5] 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.

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

[7] 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.

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

[9] Wikipedia. Java bytecode — wikipedia, the free encyclopedia, 2014. [Online; accessed 5-April-2014].

[10] Wikipedia. X86 assembly language — wikipedia, the free encyclopedia, 2014. [Online; accessed 23-March-2014].