Applying Genetic Programming to Bytecode and Assembly

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

ABSTRACT

Traditional genetic programming (GP) has not yet been able to evolve entire programs at the source code level. Instead only small sections within the programs are usually evolved. Evolving programs in either bytecode or assembly language is a method that solves this problem. This paper provides an overview of applying genetic programming to Java bytecode and x86 assembly to solve problems. Two examples of how this method can be implemented will be explored. We will explore how they evolve bytecode and assembly and discuss their experimental results.

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 in order 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 for more flexibility in GP. Evolving bytecode and assembly, instead of source code, is a method that allows for this flexibility. This is possible because bytecode and assembly languages are less restrictive syntactically than source code. We discuss this issue further in Section 3.

Orlov et al., [5] propose a method of applying GP to full-fledged programs that 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. Schulte et al., [7] also apply a similar method with both Java bytecode and the x86 assembly family.

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

This paper will examine how both Orlov et al., [6] and Schulte et al., [7] apply GP to solve programs in Java bytecode and x86 assembly. We will show that this methodology is feasible in Section 6. Orlov et al., [6] focused on evolving simple programs as a whole while Schulte et al., [7] focused on automated bug repair in programs.

This paper is organized as follows: Section 2 covers the background needed for understanding that application of GP to bytecode and assembly. 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] evolved Java bytecode. Similarly, in Section 5 we discuss how Schulte, et al., [7] evolved both x86 assembly and Java bytecode. In section 6, we report some of the problems solved by [6] and [7] in applying GP to bytecode and assembly. Section 7 will address possible future work and ideas.

2. BACKGROUND

This section explores the components of evolutionary computation (EC) and GP. x86 and Java bytecode are will also be discussed. Additionally, we will analyze 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 on evolutionary biology. EC imitates evolution through continuous optimization in order to solve problems. Optimization in EC is the selection of the best individual within a population. What an individual is depends on the problem being solved. For example, It can be a string of bits, a lisp tree, or an object. For clarity, in this paper the individuals will be programs.

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 who are then taken and modified with genetic operators that imitate procreation between two individuals. A check is then done to see if any individuals from the population solve the desired problem. If not, the process of evolution is repeated. If the problem is solved the individual with the best solution is returned.

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

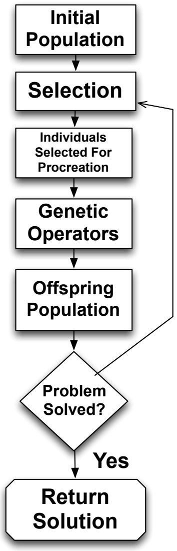

Figure 1: The process of of Evolutionary Computation

GP is a 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 to be chosen for evolution. The fitness is a value, usually numerical, that indicates how well an individual solves 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. A certain number of individuals are chosen for a simulated competition and the individual with the highest fitness wins. That individual is then used for procreation.

One way procreation 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 then replacing it with a section from the other program to form an offspring which is a new program.

Mutation is another genetic operator that is used to produce offspring. Mutation takes a program and randomly changes a section. Mutation can be used along with crossover to produce offspring.

2.2 x86 Assembly & Java Bytecode

Throughout the rest of this paper the term *instruction-level code* will be used when referring to Java bytecode and x86 assembly as one unit. Instruction-level code consist of operation codes (opcodes) which are one byte in length. However, some opcodes that take parameters are multiple bytes in length [9, 8].

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. They are designed to only run on certain physical machines.

2.2.2 Bytecode and the JVM

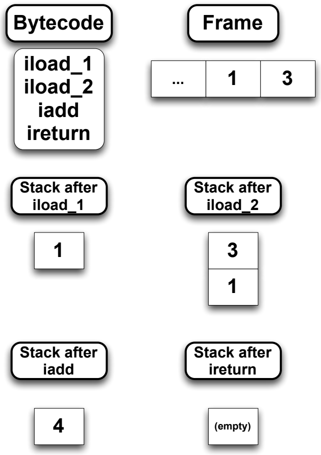

Figure 2: In this example we are assuming that the frame already contains the local variables 1 and 3 to retain simplicity. This Java bytecode sequence pushes two numbers to the stack and adds them and returns the value.

Java bytecode "is [an] intermediate, platform-independent representation" [5] of Java source code. However, "implementors of other languages can turn to the Java Virtual Machine as a delivery vehicle for their languages" [2]. A few examples of languages with which this has been done are Scala, Groovy, Jython, Kawa, JavaFx Script, Clojure, 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. Each private 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[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 within a thread can be active at any time. Popping from a stack is simply removing the top element while pushing is putting an element on top of the stack.

Figure 2 is a simple example of what Java bytecode looks like and how the stack works. Each of the opcodes in Figure 2 contain the prefix `i` which stands for the primitive integer type. These opcodes can only manipulate integers. The prefix is followed by the operation to be executed.

The opcodes that we will be examining in this paper are: `iconst_n`, `istore_n`, `iload_n`, `iadd`, `isub`, and `ireturn`. `iconst_n` produces an integer type of value `n` and pushes it on the stack. `istore_n` pops the stack and saves that element value at index n on the frame. `iload_n` takes the value of the element at index `n` on the frame and pushes it to the stack. `iadd` pops two elements from the stack, adds them and pushes the result to the stack. Similarly, `isub` pops two elements, subtracts the second element from the first element and pushes the result to the stack. `ireturn` simply pops the stack and returns that value.

In Figure, 2 when `iload_1` is executed, it takes the ele-

[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.

ment 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 operand stack, which both must be of integer type, adds them and then pushes the result to the stack. `ireturn` simply pops the stack and returns that element. I feel I don't need this anymore. It is a bit repetitive. Should I just explain in more general terms what figure 2 is doing without going into detail how each opcode works? Or should I just forget about that and just have the explanation in the caption?

3. INSTRUCTION-LEVEL CODE BENEFITS

We encounter many problems when trying to evolve source code. It is extremely difficult to evolve an entire program due to source code syntactical constraints. Traditional GP has yet to take a whole program and evolve it. The structure of the program must be known and understood in order to design an EA for that problem. When an EA is designed, it is common to simply evolve expressions or formulae within the program [6]. Most of the program would already be written and remain the same after evolution. The EA has must know what it can evolve and its location. This limitation is due to the difficulty of dealing with semantic constraints in source code.

3.1 Source Code Constraints

There is a high risk of producing a non-compilable program when evolving programs at the source code level. This is due to the fact that high-level programming languages are designed to simplify reading and writing programs. Most high-level programming languages are defined by using grammars which are used to represent the syntax of the programming language [1, 7]. The grammar does not represent the semantic constraints of a program. It does not capture the languages type system, variable visibility and accessibility, or other constraints [6]. For example, in Figure 3 both 3(a) and 3(b) comply with the syntactical rules of Java but 3(b) breaks the semantic rules and thus is illegal code. In 3(b), variable `y` is uninitialized before the test in the if statement, assigning `y` to `x` violates a type constraint, and variable `z` is not defined. 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. Thus, there is a lower risk of producing a noncompilable program during evolution and it is easier to design an EA that deals with the semantic constraints.

3.2 Flexibility for Individuals

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 does not have to focus on a specific part of the program in order to perform evolution. Understanding of the desired output, rather than input, is more important when designing the EA. A program that is being evolved stays as is and the EA goes through minor modifications to fit the problem being solved. The EA has to be given a fitness function in order to attempt to find a desirable individual. It also needs to know what genetic operators are needed as well as the desired number of generations. Depending on the program being evolved, a growth limit, on the individuals, might have to be enforced. 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 great deal 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.

When evolving instruction-level code it is also possible to evolve a very focused area within the program; similar to how traditional GP is applied to source code. For example, Schulte 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. Since they were evolving instruction-level code, they were able to fix bugs such as incorrect type declarations. This had not been possible in their earlier work done in source code [3].

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 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`. This is done by selecting code snippets that follow a set of strict rules.

4.1 Selecting Offspring

Evolving Java Byte reduces but does not remove the possibility of producing non-executable bytecode. Even though Java byte code has a simpler syntax than source code, it still continues to have syntactical constraints. Orlov et al., [5] address this issue by assessing for good offspring before letting it join the evolved population, thus ensuring offspring produced through crossover contain valid bytecode. If an illegal offspring is produced, this process is repeated, with the same parents, until a good offspring is produced or a prede-

Line	Parent 1	Parent 2	Good Offspring	Bad Offspring
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 two possible outcomes of performing unrestricted crossover on parent 1 and 2.

termined number of attempts have been made. These checks makes sure that stack depth, variable type, and control flow are respected.

In order to clarify, 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 its frame must be type compatible. The the stack must have enough elements on it so that stack underflow does not occur. Stack underflow is an attempt to pop from an empty stack. Stack and frame compatibility is accomplished by assuring that stack pops of β have identical or narrower types than α, and that stack pushes of α have identical or narrower types than stack pushes of β.

When inserting α into β, variables written before and after must be compatible with the change. Variables written by α must have identical or narrower types that are read after β. 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 should 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 the fourth bytecode sequence "Bad Offspring" since it would fail the checks for stack and frame depth.

4.2 Crossover

As previously stated crossover is a genetic operator that takes a section of code from one parent and replaces it with a section of code from another parent. Orlov et al., would perform cross over and then check if the resulting offspring passed all the constraints previously mentioned.

Figure 4 is an example of unrestricted crossover being performed resulting in both a good and a bad offspring. Let α be the opcode `iload_2` from parent 1 and β be `iload_1` from parent 2. Replacing β with α in parent 2 results in good cross over. This is because the value at index 2 on the frame can be called twice and it is an integer which respects the type constraints. Now, let α be lines

```
istore_1
istore_2
```

in parent 1 and β be lines

```
iconst_2
istore_1
```

in parent 2. In this case, replacing β with α in parent 2 results in a bad offspring. Only one integer is saved on the frame and at line 3 `istore_2` tries to pop from an empty stack causing stack underflow.

4.3 Non-Halting Offspring

An 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.

[6] 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. AUTOMATED BUG REPAIR

Schulte, 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 *positive* tests and one *negative* test. Positive meaning already passing tests and negative meaning a failing test. The negative test was used to check if an offspring fixed the bug. The positive tests were used to make sure the program retained functionality. Each offspring was compiled into either an executable binary(x86) or a class file (Java bytecode). 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 negative test being worth more. A greater weight was placed on the negative test since that was the main goal.

5.2 Genetic Operators

Schulte 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 that was calculated by checking what tests executed that instruction. This weight was used to indicate how relevant 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 was executed by both the negative test case and atleast one positive test case. For all other cases a path weight of 0 was given.

Three mutation operators were used in the experiments: mutate-insert, mutate-delete, and mutate-swap. Mutate-insert selected an instruction based off of its positive weight. Mutate-delete selected and instruction based off its 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 Schulte 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. [6] focused on evolving simple programs that performed a specific task while [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.

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. Orlov, 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.

[6] 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*

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 produce a solution to this problem for both recursion and loops. This experiment showed that FINCH is able to evolve using different list abstractions such

[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 some type of computation that results in DNE the EA avoids the computation and outputs some pre-selected value such as 1. Orlov et al. [6] do not specify how they deal this.

```
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

```
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

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 was not 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

Schulte et al., [7] were able to show that it is possible to fix human programming errors though bytecode and assembly. [7] 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,in previous work [3], 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 instruction-level code is feasible.

6.3 Future Work

Future work may include evolving 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. Also, as shown in Figure 7 a solution given by FINCH might not be very legible once decompiled back to source code. Refactoring would have to be done to make the code more readable and maintainable.

An extension of future work, using the research by [7], could include debugging on less focused areas. For example, attempting to fix bugs that require more fixes throughout the code; instead of one line. 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 at the instruction-level that are a lot simpler to do than at the source code level. Things including evolving programs as a whole and reduced assumptions of the program being evolved by the EA.Also it is easier to deal with the semantic constraints if so desired. Schulte et al., and Orlov et al., [7, 6] 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 they have only been able to do at 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].