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 perform unrestricted evolution on entire programs at the source code level. Instead only small sections within the programs are usually evolved. Not being able to evolve whole programs is an issue since it limits the flexibility on what can be evolved. Evolving programs in either bytecode or assembly language is a method that has been used to perform unrestricted evolution. This paper provides an overview of applying genetic programming to Java bytecode and x86 assembly. Two examples of how this method can be implemented will be explored. We will also discuss experimental results that include evolving recursive functions and automated bug repair.

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 commonly been used to evolve specific parts of programs instead of full-fledged programs. 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 and Sipper [5] propose a method for applying GP to full-fledged programs that requires a program to be in Java bytecode. Schulte et al., [7] apply a similar method with both Java bytecode and x86 assembly. Orlov and Sipper [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 the application of GP to bytecode and assembly. It contains information on Evolutionary Computation (EC), Java bytecode, and the Java Virtual Machine. Section 3 describes the benefits of evolving assembly and bytecode. Section 4 discusses how Orlov and Sipper [6] evolved Java bytecode. Section 5 explores how Schulte, et al., [7] evolved both x86 assembly and Java bytecode. Section 6 summarizes some of the experimental results from [6] and [7]. Possible future work and ideas is also discussed.

2. BACKGROUND

This section explores the components of EC and GP. Java bytecode will also be discussed in addition to various details of the Java Virtual Machine (JVM). This section will only address Java bytecode since knowledge of x86 assembly is not needed to understand the research done by Schulte et al., [7]. Also, throughout the rest of this paper, the term *instruction-level code* will be used when referring to either Java bytecode or x86 assembly.

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, in the best case scenario, is the selection of a few the best individuals within a population. The representation of what an individual is depends on the problem being solved. For example, it can be a string of bits, a parse tree, or an object. In this paper, the individuals will be programs in Java bytecode and x86 assembly.

The EC process is summarized in Figure 1. An initial population of individuals is generated and a selection process is used to choose the most fit individuals. The selection process gives a fitness value, usually numerical, to each individual which indicates how well it solves the specific problem. For the research discussed in this paper a higher fitness indicates a more fit individual. The individuals with the highest fitness are then taken and modified using genetic operators that imitate mutation and sexual reproduction. A check is then done to see if any individuals from the new population solve the desired problem or a predetermined number of generations is reached. If not, the process of evolution is

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

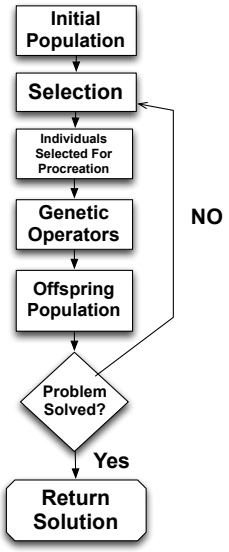

Figure 1: The process of Evolutionary Computation

repeated. If the problem is solved the individual with the best solution is returned.

GP is a tool that uses the EC process to evolve programs. In the research discussed in this paper, the selection of individuals for sexual reproduction 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 selected to be a parent.

One way sexual reproduction is simulated in GP is through the genetic operator called crossover. Crossover is the process of taking two parent individuals and extracting a section of code from one and then replacing it with a section from the other program, to form an offspring. Mutation is another genetic operator that is used to produce offspring. Mutation takes a program and performs random changes on a randomly chosen section.

2.2 Java Bytecode and the JVM

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. All these languages can be compiled to Java bytcode and be evolved.

The JVM executes bytecode through a stack-based architecture. Each method has a frame that contains an array of local variables and its own operand stack. 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.

Figure 2 is a simple example of what Java bytecode looks like and how the stack works. Bytecode consists of operation codes (opcodes) that perform operations and can manipulate both the operand stack and the frame. Each of the opcodes in Figure 2 contain the prefix `i` which stands for the primitive integer type. These opcodes can only manip-

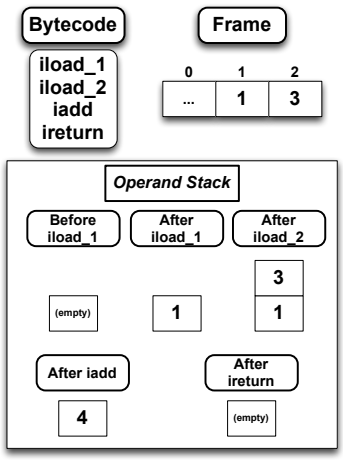

Figure 2: In this example we are assuming that the frame already contains the local variables 1 and 3 to retain simplicity. When `iload_1` is executed, it takes the element from the frame at index 1 and pushes it onto the stack. `iload_2` does the same thing but with index 2. `iadd` pops two elements from the stack, which both must be of integer type, adds them and then pushes the result to the stack. `ireturn` simply pops the stack and returns that element.

ulate integers. The prefix is followed by the operation to be executed.

The opcodes that we will be using in this paper are: `iconst_n`, `istore_n`, `iload_n`, `iadd`, `isub`, and `ireturn`. `iconst_n` pushes an integer value of `n` on the stack. `istore_n` pops the stack and stores that value at index n on the local variable array on the frame. `iload_n` takes the value of the element at index `n` on the array of local variables 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 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 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.

3. CONSTRAINTS AND BENEFITS

While it would be useful, it is difficult to evolve an entire program in source code due to semantic constraints. However, instruction-level code can deal with these constraints in a more graceful manner. This section will explore why it is difficult for source code to deal with semantic constraints and why it is easier at the instruction-level. We will discuss some of the benefits that arise from being able to deal with these constraints and evolving at the instruction-level.

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

```
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 (b) is not valid semantically. This is because `y` is uninitialized before the being used in the if statement. Also, assigning a float to an int violates type constraints and `z` in the print statement is undefined. Adapted from [6].

```
class Robot{

...

  double robotSpeed(){
      double evolvedVariable = valueFromEA;
      double speed;
      speed = (robot.location * evolvedVariable)/2;
      return speed;
  }

...

}
```

Figure 4: This is a simple example of how traditional GP typically only evolves small sections of code. In this example everything before and after `robotSpeed()` is already written and is not evolved. The only thing that the EA modifies is the variable `evolvedVariable` which is assigned some value determined by the EA.

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 [6].

3.2 Instruction-Level Code Benefits

In traditional GP it is common to evolve expressions or formulae within an existing program structure [6]. This is due to the difficulty of dealing with semantic constrains in source code. As a work around, it has been common to evolve only specific areas in a program in order to avoid dealing with too many semantic constraints. A side effect of this is that an understanding of the program's structure is needed. This usually results in most of the program being written before, and remaining the same after, evolution. Figure 4 is an example of how GP typically evolves source code.

Instruction-level code generally consists of a smaller set of instructions than source code [7]. For example, Java bytecode consists of two hundred and two instructions where as the Java source code grammar alphabet is much larger [2, 1].[1] Because of its smaller instruction size, instruction-level code is simpler syntactically and there are less semantic constraints to violate. There is a lower risk of producing a non-executable program during evolution and it is easier to design an EA that deals with the semantic constraints. Thus, it is easier to evolve whole programs at the instruction-level.

One of the benefits from being able to perform unrestricted evolution on a program is that an understanding of the structure of the program is not required. The only requirement is that it is in instruction-level code [5, 7]. The EA does not have to focus on a specific part of the program in order to perform evolution. Also, when evolving instruction-level code the initial population of programs do not have to contain a great deal of code in order to find a solution. An EA can evolve a minimal amount of code into a full-scale working program.

Typically, when an EA is developed for source code it only works for a single high-level language. However, if a program written in a high-level language can be compiled to instruction-level code then that program can be evolved at the instruction-level. All the evolution occurs at the instruction-level and once complete the result can be decompiled back to its original language. Figure 8 is an example of this being done with Java source code. There are many high-level languages that compile into instruction-level code, such as the ones listed in Section 2.2 that compile to Java bytecode.

4. FINCH

FINCH is a program developed by Orlov and Sipper [5, 6] that evolves Java bytecode programs. FINCH uses crossover to evolve Java bytecode. It takes two programs `A` and `B` and identifies randomly chosen sections α and β respectively. It then takes section α and inserts it into the previous location of β. Then, the resulting program is analyzed to see if it is executable.

4.1 Crossover and Validating Offspring

Evolving Java bytecode reduces but does not remove the possibility of producing non-executable bytecode. Even though Java bytecode has a simpler syntax than source code, it still continues to have syntactical constraints. Orlov and Sipper [5] address this issue by checking if an offspring is executable before letting it join the evolved population, thus ensuring offspring produced through crossover contain valid

[1] [1] Contains information on the Java grammars in Chapters 2 and 3

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 5: This is an example of two possible outcomes of performing unrestricted crossover on parents 1 and 2. The bold in the good and bad offspring represent the code from parent 1 replacing code in parent 2. The bad offspring breaks at `istore_2` since there is nothing on the stack to pop.

bytecode. If an illegal offspring is produced, the reproduction process is repeated, with the same parents, until a good offspring is produced or a predetermined number of attempts have been made. The specific semantic constraints that were checked were stack depth, variable type, and control flow.

In order to clarify, let α and β be sections of code of two parents `A` and `B`, respectively, on which crossover is being applied. The following constraints 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 types than α, and that stack pushes of α have identical types than stack pushes of β.[2]

When inserting α into β, variables written before and after must be compatible with the change. Variables written by α must have identical 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 such a jump would break the code. For example, even though a jump out of α is assured to go to an existing line of code in `A`, it is not assured do go to an existing line of code in `B` when replacing β.

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 and Sipper [6] would perform crossover and then check if the resulting offspring passed all the constraints previously mentioned.

Figure 5 is an example of unrestricted crossover 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 a good a crossover. This is because the value at index 2 on the frame is an integer which respects the type constraints. Now, let α be lines

```
istore_1
istore_2
```

[2]In [5] the constraints are more sophisticated. See [5] for more details.

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 pushed to the stack which is proceeded by too many istore calls. This results in stack underflow at line 3 where there is an attempt to pop from an empty stack. This offspring would fail the stack depth constraint.

4.2 Non-Halting Offspring

An problem that arises from evolving unrestricted bytecode is that the resulting program might enter a non-halting state. The previously mentioned checks for good offspring, in Section 4.1, do not check for this issue. This is because it is a run time error. This, especially becomes a concern when evolving programs that contain loops and recursion. [6] deal with this by counting how many calls are made to each function while running the program. An exception is thrown if too many calls are made to a function. The lowest possible fitness is assigned to an individual who fails this test.

[6] chose to count the calls to each function to avoid having to either run each program 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 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, since this could vary greatly depending on the program being run and the CPU load.

5. AUTOMATED BUG REPAIR

Schulte, et al., [7] focused on 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. These bugs were common human errors such as having a for-loop index off by one.

5.1 Validating Offspring

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

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 tests were already passing tests and the negative test was the initially 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 assembled and linked to either an executable binary(x86) or a class file (Java bytecode). If the program failed to assemble, it obtained a fitness of zero. If the program did assemble, 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 the 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 they found it produced better results for the type of problems being solved. Since each bug 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 the 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 sections of code to apply mutation and crossover to. Each line of instruction-level code was given a weight that was calculated by checking which tests had executed that instruction. This weight was used to indicate how relevant that line of code was to 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 at least 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 on its positive weight. Mutate-delete selected an instruction based on its negative weight and deleted it. Mutate-swap selected two instructions based on 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, they had a probability of zero of being selected for mutation.

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

5.3 Non-Halting Offspring

Schulte, et al., [7] chose a different approach than that of Orlov and Sipper [6] when dealing with non-halting offspring. They decided to not check for non-halting cases and instead ran 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, which supports Orlov and Sipper's claim that terminating threads is unreliable. There were also issues with buffer overflow which occurs when there is an attempt to write to a memory address outside of a data structure's legal memory. This can cause the program and system as a whole to crash. However, this was expected since one of the types of bugs that Schulte et al. [7] were trying to fix was buffer overflow. Since each individual was ran on a VM if buffer overflow did cause the system to crash only the VM would crash and not the EA.

6. RESULTS

Orlov, Sipper, 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 and Sipper [6] focused on evolving simple programs that performed a specific task while Schulte et al., [7] focused on debugging programs through evolution.

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

Figure 6: Example of a possible starting program,before compilation, for a symbolic regression problem.

6.1 FINCH

The five problems that Orlov and Sipper [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. FINCH was able to evolve programs that solved each of these problems.

In each test, FINCH was given a program that had a zero fitness. The elements included in the programs were the minimal components needed to successfully evolve and solve the problem. For example, if the problem consisted of adding all the elements in an array, then a loop or a recursive call was provided along with one variable of each type needed. Figure 6 is an example of this. Only one variable of type `double` was proved along with a the small function set of {+, *}.

6.1.1 Symbolic Regression

Symbolic regression is a method of finding a mathematical function that best fits a finite set of points. Orlov and Sipper [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 $\{+, -, *, \%, sin, cos, e, ln\}$[3] was used for most of the experiments. This was done in order to mimic previous experimentation [4] and to compare the results to traditional GP.

Each experiment started off with an individual of fitness zero and usually with a minimal amount of code, such as in Figure 6. Figure 6 only contains a simple function set of {+,*} and a number object. Orlov and Sipper [6] found it possible to evolve minimalist programs,such as in Figure 6, to full-fledged working programs that solved symbolic regression of up to 9-degree polynomials. This demonstrates how little content is needed to find a solution.

In [6] they evolved programs with a chance of 90% crossover using a simple fitness algorithm. With a maximum of fifty-one generation 99% of the time a maximum fitness individual was found. With a more complex fitness algorithm, that allowed for a small margin of error when calculating each point, an individual with a maximum fitness was always found.

6.1.2 Array Sum

[3] e is being referred to as the function in the java.lang.Math library. Also, % and ln are protected division and logarithm. This means that if division by zero or $ln(n)$, $n \leq 0$, occur the result is set to a pre-selected value such as 1. Orlov and Sipper [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 7: Initial population function, for the array sum problem, before being compiled to Java bytecode to be evolved.This function enters into infinite recursion in the if statement. Taken from [6].

```
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 8: FINCH's decompiled solution to the initial program presented in Figure 7. Taken from [6].

The array sum problem consists of adding up all the values in an array. This problem is important because it requires evolving a loop or recursion to find a solution. This would show that FINCH is capable of evolving more complex programs. FINCH was able to produce a solution to this problem using both recursion and loops. It was also shown that FINCH was able to apply evolution to different list abstractions such as List and ArrayList.

When evolving array sum with recursion the initial population consisted of an individual who entered infinite recursion as shown in Figure 7. Due to the way FINCH deals with non-halting programs, this was not a problem and evolution ensued. The resulting code from evolving Figure 7 is shown in Figure 8.

6.2 Automated Bug Repair

Schulte et al., [7] were able to show that it is possible to fix human programming errors by evolving instruction-level code. They were able to successfully debug various programs containing bugs such as infinite loops and buffer overflow. The interesting thing about these experiments was that they were performed on programs written by software developers and at times the programs consisted of thousands of lines of code. Some bugs that were fixed through the evolution of instruction-level code were not possible in their previous work [3] with source code. For example, their EA in their previous work was not capable of fixing incorrect type declarations. This suggests it is easier to repair a wider array of bugs by evolving instruction-level code.

Although a considerable number 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 74.4 for assembly compared to 63.6 for C from previous work [3]. This indicates that computational work needed to evolve repairs in assembly is comparable to that of source code. Even programs that contained thousands of lines of code only required a few generations. Thus, using instruction-level evolution to automate bug repair is feasible.

7. CONCLUSIONS AND FUTURE WORK

7.1 Future Work

Future work may include evolving programs whose solutions are much longer and more complicated. Orlov and Sipper [6] proved that by applying GP to bytecode it is possible to solve many simple problems. However, most of the solutions only consisted of a small number of lines of code. It would be exciting to see something more complex be evolved. Also, as shown in Figure 8 a solution given by FINCH might not be clearly 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 in [7], could include debugging on less focused areas, such as attempting to fix bugs that require multiple fixes throughout the code rather than simply altering 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. Some changes to a program could cause it to break. If test cases do not cover that case then a fix to the original bug could contain a different bug.

7.2 Conclusion

Evolving entire programs at the instruction-level is possible at the source code level. No assumptions about the program being evolved is needed, and it is easier to deal with the semantic constraints. Schulte et al., and [7, 6] showed that evolving instruction-level code is just as feasible as source code and at times better, such as for debugging incorrect type declarations. In conclusion, evolving instruction-level code is feasible and exciting for the field of EC. It opens up many possibilities that were once unavailable.

Acknowledgments

A special acknowledgment to Nic McPhee and Elena Machkasova for their time and knowledge. Also, thanks to Debra Dogotch, Chris Thomas, Peter Dolan, and Tim Snyder.

8. 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.