

Byte Code Genetic Programming

Brad Harvey

Tandem Computers
14231 Tandem Bl.
Austin, Texas 78728
bharvey@austx.tandem.com

James A. Foster

Computer Science Dept.
University of Idaho
Moscow, Idaho 83844
foster@cs.uidaho.edu

Deborah Frincke

Computer Science Dept.
University of Idaho
Moscow, Idaho 83844
frincke@cs.uidaho.edu

ABSTRACT

This paper explores the idea of using Genetic Programming (GP) to evolve Java Virtual Machine (JVM) byte code to solve a sample symbolic regression problem. The evolutionary process is done completely in memory using a standard Java environment.

1. Introduction

An increasing number of software vendors are including Java, or Java-based components, as strategic parts of their systems. For example, the leading browser vendors (Microsoft, Netscape, and Sun) support Java applets. The leading database vendors (Oracle, IBM, Tandem, Informix, etc.) are collaborating on standards for embedding SQL in Java and Java within the database engine. A leading smart card vendor (Schlumberger) offers a subset of the Java operating environment on their smart card so that smart card application developers can develop applications using standard Java development tools. Numerous other vendors in more non-traditional computing environments (e.g., embedded devices or real-time operating systems) are in the process of supporting the Java environment.

One drawback of using Java in the case of time critical applications is performance. Java programs are slower than conventional compiled programs because a software virtual processor executes them. However, with just-in-time (JIT) compilers, adaptive compilers, and with the possible future option of having at least part of the JVM execution engine implemented in hardware, this performance gap will only shrink.

In the area of computer science research with respect to machine learning and automatic programming, GP has proved to be a very powerful paradigm for solving diverse problems from a variety of different problem domains. For example, it has been used to solve regression problems, control robots, design hardware, and classify protein segments.

Therefore, because of Java's rapid success as operating

environment and because of the power of GP, this paper explores using GP to directly evolve JVM byte code in the context of a standard Java environment. The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 covers preliminary information. Section 4 discusses the architecture of the system described in this work. Section 5 provides some sample results. And Section 6 concludes the paper.

2. Related Work

2.1 Evolution of Machine Code

Recently there has been work done using GP to evolve machine code "Nordin and Banzhaf (1995,1997)". This system is referred to as the Compiling Genetic Programming System (CGPS). It departs from using tree-structured individuals "Koza (1992)", and instead represents them as linear bit strings that represent actual machine code. It takes advantage of the von Neumann architecture of a computer in which a program and its data reside in the same memory space, enabling a program to manipulate itself as it would data. While the individuals of the evolutionary process are represented as machine code, the GP kernel including the genetic operators are written in C. The system has been implemented on Sun SPARC workstations which our register based machines with a 32-bit instruction size format. The benefits of the CGPS approach compared to the conventional GP approach are:

- Execution speed - no interpretation. For example, an individual's fitness is computed by directly executing machine code. Indications are that this speed-up is between 1500-2000 times compared to the tree-structured Lisp approach.
- Compact GP kernel (30KB) - A Lisp interpreter is not required.
- Compact representation - A single 32-bit instruction of the SPARC is able to represent four nodes in a conventional approach (operator, two operands, and result).
- Simple memory management - Due to the linear structure of the binary machine code.

3. Preliminaries

3.1 Initial Goals

The main effort of this work is to understand the issues of evolving JVM byte code in the context of a standard Java application and runtime environment. In other words, approaches such as building a custom JVM engine or using native methods to facilitate the evolutionary process were not considered because we felt the first approach would be of greater significance. Additionally, this paper does not try to show that GP is a viable technology as this has already been demonstrated. Therefore, the goals of this work are as follows:

- Create and execute a prototype Java application that uses GP to evolve JVM byte code using a standard Java development and execution environment.
- Directly execute the evolved individuals in the context of this program to compute their raw fitness values using the JVM runtime system.
- The evolutionary process is done completely in memory.

This Java program (or GP kernel) is called the byte code GP system (BCGP). A symbolic regression problem is used as the problem domain for this work because solutions for this problem domain are just functions. Symbolic regression is the process of finding a symbolic representation of a function that fits a given sampling of data points.

3.2 JVM Runtime

A detailed explanation of the JVM runtime system is beyond the scope of this paper (see "Lindholm and Yellin (1996)" and "Meyer and Downing (1997)"). A very brief overview follows.

The JVM runtime system executes a program that is contained in a set of class files. A class file is a binary stream of bytes similar in idea to an executable a.out file in UNIX. For example, it contains a magic number, versioning information, a constant pool which is similar to a symbol table, and for each method in the class the JVM instructions which constitute the method. There is a 1-1 relationship between a Java class in a Java source file and a binary class file. Multiple Java classes can reside in a single source file, but each Java class has its own binary class file after it has been compiled. Normally class files are created using a Java compiler (e.g., `javac`). However, the class file is not dependent on the Java language. In other words, other tools or language compilers not related to Java could generate class files that are executed by the JVM runtime system assuming they are valid with respect to the specification given in "Lindholm and Yellin (1996)".

The JVM execution engine is a stack based virtual processor containing an instruction set of about 150+ instructions, most of which are low-level instructions similar to those found in an actual hardware processor. Most instructions are a single byte so it does not have a fixed instruction format like a RISC processor. It does not contain

a set of explicit general purpose registers like a convention processor.

The JVM supports the notion of separate threads of execution. Each thread has its own program counter and stack for state related purposes. Each call in the context of a thread generates a new stack frame on the thread's stack. Each stack frame contains local variables, an operand stack, and other state related information. In many ways, the local variables serve the same purpose as registers (e.g., they contain the method's parameter values). The operand stack is used by the JVM instructions as a stack based scratch pad. For example, the floating point instruction (`fadd`) pops two values from the operand stack, adds their values, and pushes the result back on the operand stack. Each method's maximum operand stack size is determined at compile time.

The JVM loads and links classes using symbolic references dynamically at runtime. The JVM contains a system class loader that knows the details of loading classes from the local file system. However, applications that load classes from other sources provide their own version of the class loader. For example, a Java enabled browser contains a class loader that knows how to load applets over the network from a URL.

4. Architecture

4.1 Representation

During a run, each generation is represented by its own in memory, dynamically created Java class with a method in it for each individual in the population. Therefore, generation classes and methods are created using a naming scheme that logically identifies them. For example, if the population size is 100, then each class contains 100 functions: `f001 - f100`. The signature of a generation function is: `static float f(float x)` as each function takes a float as an a single parameter (independent variable) and returns a float as its result (dependent variable). Logically the body of a function is Java method, which is a return of a nested list of binary expressions:

```
static float f001(float x) {
    Return (((x <op> x) <op> x) <op> x)
};
```

Where `<op>` represents an element from the function set (+, -, *, /). And `x` is a variable representing the only element of the terminal set. The actual genotype is a linear, variable length, byte array of JVM instructions. The genotype consisting of byte code representing the function `f001()` listed above (assuming the functions from left to right are +, -, *, /) is: `0x22 0x22 0x62 0x22 0x66 0x22 0x6A 0x22 0x6E 0xAE`

The crossover operator in BCGP does not effect the first byte (header) or the last byte (footer) in an individual. Furthermore, it takes place at an odd byte offset for a length

which is a multiple of two to ensure that the resulting byte code is correct (i.e., it preserves the pushing of the local variable zero on the operand stack before the function operation).

4.2 Fitness

This work is based upon evolving individuals to solve the symbolic regression problem for the function: $f(x) = x^4 + x^3 + x^2 + x$ with twenty points being used for the independent variable (x) in the interval $[-1.0, 1.0]$. The raw fitness of an individual is the sum of the absolute difference between the expected value of $f(x)$ and the actual value returned from executing the individual (Java method) for a given value of the independent variable for each of the twenty fitness cases. Therefore, the smaller the fitness (error) value the fitter the individual. This is referred to as error-driven evolution.

4.3 Selection

BCGP uses k -tournament selection with $k=2$ in order to select an individual in the current generation to be used as a mating parent to create offspring for the next generation using one of the genetic operators. More specifically, two individuals are randomly selected from the current population with the fitter individual winning the selection. Tournament selection is used because its easy to implement, efficient ($O(n)$ time), and provides reasonable results.

4.4 Genetic Operators

BCGP uses the single-point crossover and the direct reproduction operators. Once two parents have been selected from the current population, crossover is used to create two offspring for the next generation each of which contains parts of the two parents. More specifically, the byte code between two parent methods is exchanged to create two new methods, each of which contains byte code from both parents. No repairing is required and as described above measures are taken to ensure the resulting children contain valid byte code. The direct reproduction operator simply copies a selected parent directly into the next generation.

4.5 Java Details

A few BCGP Java specific details are in order.

- BCGP dynamically creates the binary generation classes and associated methods in memory. The publicly available JAS class builder package facilitates this process "Meyer and Downing (1997)". These dynamically created binary classes files are verifiably sound with respect to the JVM specification "Lindholm and Yellin (1996)".
- BCGP provides its own class loader that knows how to load these in memory class files (i.e., generation class

files come from memory as compared to the file system).

- BCGP discovers a generation's methods and calls these methods using the Java reflection facilities.
- BCGP provides a switch such that these in memory class files can be written to disk so that they can be incorporated into other applications or so that they can be manipulated with standard Java tools (e.g., `javap` which is a class disassembler) for analysis purposes.
- The following instructions are used from the JVM instruction set: `fload_0 (0x22)`, `fadd (0x62)`, `fsub (0x66)`, `fmul (0x6A)`, `fdiv (0x6E)`, `freturn (0xAE)`, and `nop (0x00)`.
- The Sun JDK version 1.1.1 is used for this work.

4.6 Algorithm

BCGP basically uses the GP algorithm described in "Koza (1992)". This algorithm with some BCGP specific details is summarized in the following. Note, that each run of a multi-run session repeats this process.

4.6.1 Initialization

1. Initialize the GP parameter's (G , M , S , etc).
2. Create an in memory binary class for each generation with each class containing M no-op methods in it of size S (i.e., each method has a maximum fixed size with its code being initialized to all no-ops (`nop=0x00`)).
3. Initialize the initial random population (generation 0) with each method having a random size (maximum $< S/2$) and being composed of random operators from the function set.
4. Load the in memory class for the initial generation into the JVM using BCGP's dynamic class loader.

4.6.2 Evolve the Solution

Repeat the following steps for each generation g :

- Evaluate the fitness of the current generation as follows:
 1. Get the class for generation g from the dynamic class loader and then get the methods from the class.
 2. For each individual (method) in g and for each fitness case f :
 - a. Invoke the method for the given independent variable for case f .
 - b. Compute and accumulate individual's fitness for case f .
 - c. Collect relevant statistics (e.g., best-of-generation, hits).
- Create the next generation using one of the genetic operators as follows:
 1. Repeat until the size of the new generation is equal to M :
 - a. With probability PC use crossover:
 - Use tournament selection twice to select the two mating parents.
 - Use crossover on the two parents to produce two new individuals for the next generation.
 - Increment the size of the new generation by two.
 - b. Otherwise use direct reproduction:
 - Use tournament selection to select one individual to directly copy into the next generation.
 - Increment size of the new generation by one.
 2. Load the new generation class into the JVM using the dynamic class loader.

4.6.3 Report the Results

Reports relevant statistics such as best of run information as far as generation, individual, individual's fitness, etc.

5. Results

This section presents some sample results for experiments run with BCGP (see next page for tables and figures). During these experiments, the in memory class files were written to disk so that they could be examined to provide some of the information presented in these results. The parameters used for these experiments are summarized in Table-1. The first results presented are from a session that consisted of ten runs. A generation is considered successful if it contains an individual that correctly computes the answer to all twenty fitness cases. An answer is considered correct if its error is less than the error tolerance. The JVM instructions in symbolic form for the successful individual

(method number 23) for run 10 generation 24 are listed in Table-2. This individual effectively 38 bytes (nodes) in size. The first byte (fload_0) and last byte (freturn) are the header and footer, respectively. Between these two bytes, every other byte is a reference to the independent variable x (fload_0) or an element from the function set (e.g., fdiv, fadd). Figure-1 lists a possible Java interpretation for these JVM instructions. The data for both Table-2 and figure 1 are obtained by using a class file dissembler and a class file de-compiler, respectively, on the successful generation class file.

The last result provided is from a different ten run session, where the successful individual (generation 30, method 17) is more efficient as compared to the successful individual given above (14 versus 38 nodes with 6 versus 13 function evaluations). This individual is optimal in that it has the minimum number of operations required to solve the problem (i.e., 3 multiplications and 3 additions). Table-3 and Figure-2 provide the JVM instructions and Java interpretation, respectively

6. Conclusion

In this paper the idea of evolving JVM byte code in the context of a standard Java environment is explored. The technique developed in this work generated in memory class files to represent generations with each class file containing methods that represented the individuals in the population for the given generation. The fitness of an individual is computed by directly executing the byte code contained in a method using the JVM in the context of the hosting Java application referred to as BCBP.

Bibliography

- Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.
- Nordin, P. and Banzhaf, W. 1995. Evolving Turing-complete programs for a register machine with self-modifying code. *Proceedings of the International Conference on Genetic Algorithms, Pittsburgh*.
- Nordin, P. 1997. A Compiling Genetic Programming System that Directly Manipulates the Machine Code. *Handbook of Evolutionary Computation*. IOP Publishing Ltd and Oxford University Press.
- Lindholm, T. and Yellin, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley.
- Meyer, J. and Downing, T. 1997. *JAVA Virtual Machine*. O'REILLY.

Table 1 BCGP Parameters

Description	Value
Objective	Find a function fitting a given sampling of (x,y) data points where the function is $f(x) = x^4 + x^3 + x^2 + x$.
Terminal set	x, the independent variable.
Function set	+ (fadd=0x62), - (fsub=0x66), * (fmul=0x6A), / (fdiv=0x6E).
Fitness cases	The given sample of 20 data points (x,y) where x is in the interval [-1.0, +1.0].
Error tolerance	0.01
Number of generations (G)	32
Population size (M)	32
Maximum individual size (S)	64
Maximum method operand stack size	2
Selection method	Tournament (2)
Crossover method/probability (PC)	Single point/0.9
Direct reproduction/copy probability (PD)	0.1
Mutation probability (PM)	0.0

Table 2 Successful Individual

	1	2	3	4	5	6	7	8	9	10
0	fload_0	fload_0	fdiv	fload_0	fadd	fload_0	fmul	fload_0	fmul	fload_0
10	fdiv	fload_0	fmul	fload_0	fadd	fload_0	fmul	fload_0	fadd	fload_0
20	fmul	fload_0	fdiv	fload_0	fdiv	fload_0	fmul	freturn		

```
public static float f023(float x) {
    return ((x / x + x) * x * x / x * x + x) * x + x) * x / x / x * x;
}
```

Figure 1 Java Interpretation

Table 3 More Efficient Successful Individual

	1	2	3	4	5	6	7	8	9	10
0	fload_0	fload_0	fmul	fload_0	fadd	fload_0	fmul	fload_0	fadd	fload_0
10	fmul	fload_0	fadd	freturn						

```
public static float f0117(float x) {
    return ((x * x + x) * x + x) * x + x;
}
```

Figure 2 Java Interpretation of Efficient Individual