Chapter 1

FINCH: A SYSTEM FOR EVOLVING JAVA (BYTECODE)

Michael Orlov and Moshe Sipper
Department of Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel.

Abstract The established approach in genetic programming (GP) involves the definition of functions and terminals appropriate to the problem at hand, after which evolution of expressions using these definitions takes place. We have recently developed a system, dubbed FINCH (Fertile Darwinian Bytecode Harvester), to evolutionarily improve actual, *extant* software which was *not intentionally written* for the purpose of serving as a GP representation in particular, nor for evolution in general. This is in contrast to existing work that uses restricted subsets of the Java bytecode instruction set as a representation language for individuals in genetic programming. The ability to evolve Java programs will hopefully lead to a valuable new tool in the software engineer's toolkit.

Keywords: Java bytecode, automatic programming, software evolution, genetic programming.

1. Introduction

The established approach in genetic programming (GP) involves the definition of functions and terminals appropriate to the problem at hand, after which evolution of expressions using these definitions takes place (Koza, 1992; Poli et al., 2008). Poli et al. recently noted that:

> While it is common to describe GP as evolving *programs*, GP is not typically used to evolve programs in the familiar Turing-complete languages humans normally use for software development. It is instead more common to evolve programs (or expressions or formulae) in a more constrained and often domain-specific language. (Poli et al., 2008, ch. 3.1; emphasis in original)

The above statement is (arguably) true not only where "traditional" tree-based GP is concerned, but also for other forms of GP, such as linear GP and grammatical evolution (Poli et al., 2008).

R. Riolo et al. (eds.), *Genetic Programming Theory and Practice VIII,*
DOI 10.1007/978-1-4419-7747-2_1,

```
class F {
int fact(int n) {
    // offsets 0-1
    int ans = 1;

    // offsets 2-3
    if (n > 0)
        // offsets 6-15
        ans = n *
            fact(n-1);

    // offsets 16-17
    return ans;
}}
```

(a)

```
0 iconst_1
1 istore_2
2 iload_1
3 ifle 16
6 iload_1
7 aload_0
8 iload_1
9 iconst_1
10 isub
11 invokevirtual #2
14 imul
15 istore_2
16 iload_2
17 ireturn
```

(b)

Figure 1-1. A recursive factorial function in Java (a) and its corresponding bytecode (b). The argument to the virtual method invocation (`invokevirtual`) references the `int F.fact(int)` method via the constant pool.

We have recently developed a system, dubbed FINCH (Fertile Darwinian Bytecode Harvester), to evolutionarily improve actual, *extant* software, which was *not intentionally written* for the purpose of serving as a GP representation in particular, nor for evolution in general. The only requirement is that the software source code be either written in Java or can be compiled to Java bytecode. The following chapter provides an overview of our system, ending with a précis of results. Additional information can be found in (Orlov and Sipper, 2009; Orlov and Sipper, 2010).

Java compilers typically do not produce machine code directly, but instead compile source-code files to platform-independent *bytecode*, to be interpreted in software or, rarely, to be executed in hardware by a Java Virtual Machine (JVM) (Lindholm and Yellin, 1999). The JVM is free to apply its own optimization techniques, such as Just-in-Time (JIT) on-demand compilation to native machine code—a process that is transparent to the user. The JVM implements a stack-based architecture with high-level language features such as object management and garbage collection, virtual function calls, and strong typing. The bytecode language itself is a well-designed assembly-like language with a limited yet powerful instruction set (Engel, 1999; Lindholm and Yellin, 1999). Figure 1-1 shows a recursive Java program for computing the factorial of a number, and its corresponding bytecode.

The JVM architecture is successful enough that several programming languages compile directly to Java bytecode (e.g., Scala, Groovy, Jython, Kawa, JavaFX Script, and Clojure). Moreover, Java *decompilers* are available, which facilitate restoration of the Java source code from compiled bytecode. Since the design of the JVM is closely tied to the design of the Java programming

language, such decompilation often produces code that is very similar to the original source code (Miecznikowski and Hendren, 2002).

We chose to automatically improve extant Java programs by evolving the respective compiled bytecode versions. This allows us to leverage the power of a well-defined, cross-platform, intermediate machine language at just the right level of abstraction: We do not need to define a special evolutionary language, thus necessitating an elaborate two-way transformation between Java and our language; nor do we evolve at the Java level, with its encumbering syntactic constraints, which render the genetic operators of crossover and mutation arduous to implement.

Note that we do not wish to invent a language to improve upon some aspect or other of GP (efficiency, terseness, readability, etc.), as has been amply done. Nor do we wish to extend standard GP to become Turing complete, an issue which has also been addressed (Woodward, 2003). Rather, conversely, our point of departure is an *extant*, highly popular, general-purpose language, with our aim being to render it evolvable. The ability to evolve Java programs will hopefully lead to a valuable new tool in the software engineer's toolkit.

The motivation behind evolving Java bytecode is detailed in Section 2. The principles of bytecode evolution are described in Section 3. Section 4 describes compatible bytecode crossover—the main evolutionary operator driving the FINCH system. Alternative ways of evolving software are considered in Section 5. Program halting and compiler optimization issues are dealt with in Sections 6 and 7. Current experimental results are summarized in Section 8, and the concluding remarks are in Section 9.

2. Why Target Bytecode for Evolution?

Bytecode is the intermediate, platform-independent representation of Java programs, created by a Java compiler. Figure 1-2 depicts the process by which Java source code is *compiled* to bytecode and subsequently *loaded* by the JVM, which *verifies* it and (if the bytecode passes verification) decides whether to *interpret* the bytecode directly, or to *compile* and *optimize* it—thereupon executing the resultant native code. The decision regarding interpretation or further compilation (and optimization) depends upon the frequency at which a particular method is executed, its size, and other parameters.

Our decision to evolve bytecode instead of the more high-level Java source code is guided in part by the desire to avoid altogether the possibility of producing non-compilable source code. The purpose of source code is to be easy for human programmers to create and to modify, a purpose which conflicts with the ability to automatically modify such code. We note in passing that we do not seek an evolvable programming language—a problem tackled, e.g., by

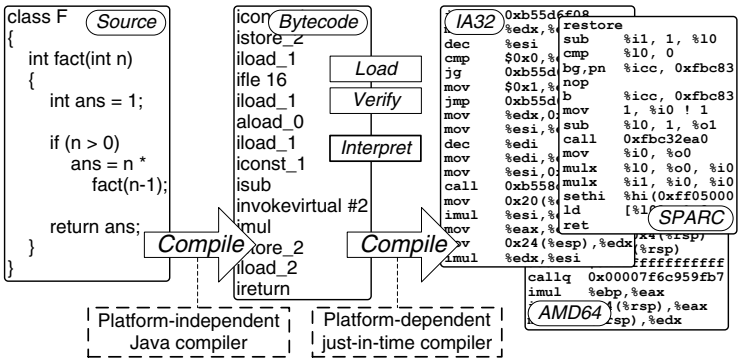

Figure 1-2. Java source code is first compiled to *platform-independent* bytecode by a Java compiler. The JVM only loads the bytecode, which it verifies for correctness, and raises an exception in case the verification fails. After that, the JVM typically interprets the bytecode until it detects that it would be advantageous to compile it, with optimizations, to native, *platform-dependent* code. The native code is then executed by the CPU as any other program. Note that no optimization is performed when Java source code is compiled to bytecode. Optimization only takes place during compilation from bytecode to native code.

(Spector and Robinson, 2002)—but rather aim to handle the Java programming language in particular.

Evolving bytecode instead of source code alleviates the issue of producing non-compilable programs to some extent—but not completely. Java bytecode must be *correct* with respect to dealing with stack and local variables (cf. Figure 1-3). Values that are read and written should be type-compatible, and stack underflow must not occur. The JVM performs bytecode verification and raises an exception in case of any such incompatibility.

We wish not merely to evolve bytecode, but indeed to evolve *correct* bytecode. This task is hard, because our purpose is to evolve given, unrestricted code, and not simply to leverage the capabilities of the JVM to perform GP. Therefore, basic evolutionary operations, such as bytecode crossover and mutation, should produce correct individuals.

3. Bytecode Evolution Principles

We define a *good* crossover of two parents as one where the offspring is a *correct* bytecode program, meaning one that passes verification with no errors; conversely, a *bad* crossover of two parents is one where the offspring is an *incorrect* bytecode program, meaning one whose verification produces errors. While it is easy to define a trivial slice-and-swap crossover operator on two programs, it is far more arduous to define a *good* crossover operator. This latter is necessary in order to preserve variability during the evolutionary process, because incorrect programs cannot be run, and therefore cannot be ascribed a

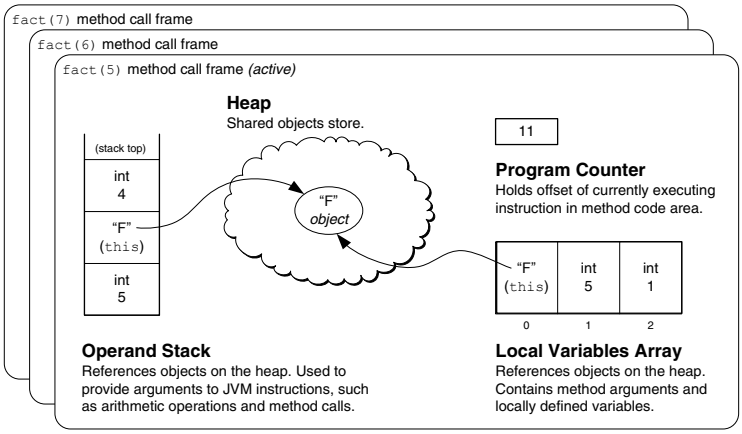

Figure 1-3. Call frames in the architecture of the Java Virtual Machine, during execution of the recursive factorial function code shown in Figure 1-1, with parameter $n = 7$. The top call frame is in a state preceding execution of `invokevirtual`. This instruction will pop a parameter and an object reference from the operand stack, invoke the method `fact` of class `F`, and open a new frame for the `fact(4)` call. When that frame closes, the returned value will be pushed onto the operand stack.

fitness value—or, alternatively, must be assigned the worst possible value. Too many bad crossovers will hence produce a population with little variability.

Note that we use the term *good* crossover to refer to an operator that produces a viable offspring (i.e., one that passes the JVM verification) given two parents; *compatible* crossover, defined below, is one mechanism by which good crossover can be implemented.

The Java Virtual Machine is a stack-based architecture for executing Java bytecode. The JVM holds a stack for each execution thread, and creates a frame on this stack for each method invocation. The frame contains a code array, an operand stack, a local variables array, and a reference to the constant pool of the current class (Engel, 1999). The code array contains the bytecode to be executed by the JVM. The local variables array holds all method (or function) parameters, including a reference to the class instance in which the current method executes. In addition, the variables array also holds local-scope variables. The operand stack is used by stack-based instructions, and for arguments when calling other methods. A method call moves parameters from the caller's operand stack to the callee's variables array; a return moves the top value from the callee's stack to the caller's stack, and disposes of the callee's frame. Both the operand stack and the variables array contain typed items, and instructions always act on a specific type. The relevant bytecode instructions are prefixed accordingly: '`a`' for an object or array reference, '`i`' and '`l`' for integral types **int** and **long**, and

'f' and 'd' for floating-point types **float** and **double**.[1] Finally, the constant pool is an array of references to classes, methods, fields, and other unvarying entities. The JVM architecture is illustrated in Figure 1-3.

In our evolutionary setup, the individuals are bytecode sequences annotated with all the necessary stack and variables information. This information is gathered in one pass over the bytecode, using the ASM bytecode manipulation and analysis library (Bruneton et al., 2002). Afterwards, similar information for any sequential code segment in the individual can be aggregated separately. This preprocessing step allows us to define compatible two-point crossover on bytecode sequences (Orlov and Sipper, 2009). Code segments can be replaced only by other segments that use the operand stack and the local variables array in a depth-compatible and type-compatible manner. The compatible crossover operator thus maximizes the viability potential for offspring, preventing type incompatibility and stack underflow errors that would otherwise plague indiscriminating bytecode crossover. Note that the crossover operation is *unidirectional*, or asymmetric—the code segment compatibility criterion as described here is not a symmetric relation. An ability to replace segment α in individual A with segment β in individual B does not imply an ability to replace segment β in B with segment α.

As an example of compatible crossover, consider two identical programs with the same bytecode as in Figure 1-1, which are reproduced as parents A and B in Figure 1-4. We replace bytecode instructions at offsets 7–11 in parent A with the single `iload_2` instruction at offset 16 from parent B. Offsets 7–11 correspond to the `fact(n-1)` call that leaves an integer value on the stack, whereas offset 16 corresponds to pushing the local variable `ans` on the stack. This crossover, the result of which is shown as offspring x in Figure 1-4, is *good*, because the operand stack is used in a compatible manner by the source segment, and although this segment reads the variable `ans` that is not read in the destination segment, that variable is guaranteed to have been written previously, at offset 1.

Alternatively, consider replacing the `imul` instruction in the newly formed offspring x with the single `invokevirtual` instruction from parent B. This crossover is *bad*, as illustrated by incorrect offspring y in Figure 1-4. Although both `invokevirtual` and `imul` pop two values from the stack and then push one value, `invokevirtual` expects the topmost value to be of reference type `F`, whereas `imul` expects an integer. Another negative example is an attempt to replace bytecode offsets 0–1 in parent B (that correspond to the `int ans=1` statement) with an empty segment. In this case, illustrated by incorrect offspring z in Figure 1-4, variable `ans` is no longer guaranteed to be initialized

[1] The types **boolean**, **byte**, **char** and **short** are treated as the computational type **int** by the Java Virtual Machine, except for array accesses and explicit conversions (Lindholm and Yellin, 1999).

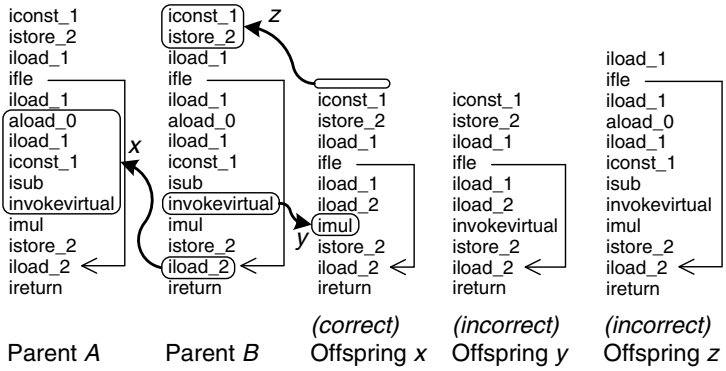

Figure 1-4. An example of good and bad crossovers. The two identical individuals A and B represent a recursive factorial function (see Figure 1-1; here we use an arrow instead of branch offset). In parent A, the bytecode sequence that corresponds to the `fact(n-1)` call that leaves an integer value on the stack, is replaced with the single instruction in B that corresponds to pushing the local variable `ans` on the stack. The resulting correct offspring x and the original parent B are then considered as two new parents. We see that either replacing the first two instructions in B with an empty section, or replacing the `imul` instruction in x with the `invokevirtual` instruction from B, result in incorrect bytecode, shown as offspring y and z—see main text for full explanation.

when it is read immediately prior to the function's return, and the resulting bytecode is therefore incorrect.

A mutation operator employs the same constraints as compatible crossover, but the constraints are applied to variations of the same individual. The requirements for correct bytecode mutation are thus derived from those of compatible crossover. To date, we did not use this type of mutation as it proved unnecessary, and instead implemented a restricted form of constants-only point mutation, where each constant in a new individual is modified with a given probability.

4. Compatible Bytecode Crossover

As discussed above, compatible bytecode crossover is a fundamental building block for effective evolution of correct bytecode. In order to describe the formal requirements for compatible crossover, we need to define the meaning of variable accesses for a segment of code. That is, a section of code (that is not necessary linear, since there are branching instructions) can be viewed as reading and writing some local variables, or as an aggregation of reads and writes by individual bytecode instructions. However, when a variable is written before being read, the write "shadows" the read, in the sense that the code executing prior to the given section does not have to provide a value of the correct type in the variable.

Variables Access Sets. We define variables access sets, to be used ahead by the compatible crossover operator, as follows: Let a and b be two locations in the same bytecode sequence. For a set of instructions $\delta_{a,b}$ that could potentially be executed starting at a and ending at b, we define the following access sets.

$\delta^r_{a,b}$: set of local variables such that for each variable v, there exists a *potential* execution path (i.e., one not necessarily taken) between a and b, in which v is read before any write to it.

$\delta^w_{a,b}$: set of local variables that are written to through at least one potential execution path.

$\delta^{w!}_{a,b}$: set of local variables that are guaranteed to be written to, no matter which execution path is taken.

These sets of local variables are incrementally computed by analyzing the data flow between locations a and b. For a single instruction c, the three access sets for δ_c are given by the Java bytecode definition. Consider a set of (normally non-consecutive) instructions $\{b_i\}$ that branch to instruction c or have c as their immediate subsequent instruction. The variables accessed between a and c are computed as follows:

$\delta^r_{a,c}$ is the union of all reads δ^r_{a,b_i}, with the addition of variables read by instruction c—unless these variables are guaranteed to be written before c. Formally, $\delta^r_{a,c} = \left(\bigcup_i \delta^r_{a,b_i}\right) \cup \left(\delta^r_c \setminus \bigcap_i \delta^{w!}_{a,b_i}\right)$.

$\delta^w_{a,c}$ is the union of all writes δ^w_{a,b_i}, with the addition of variables written by instruction c: $\delta^w_{a,c} = \left(\bigcup_i \delta^w_{a,b_i}\right) \cup \delta^w_c$.

$\delta^{w!}_{a,c}$ is the set of variables guaranteed to be written before c, with the addition of variables written by instruction c: $\delta^{w!}_{a,c} = \left(\bigcap_i \delta^{w!}_{a,b_i}\right) \cup \delta^{w!}_c$ (note that $\delta^{w!}_c = \delta^w_c$). When $\delta^{w!}_{a,c}$ has already been computed, its previous value needs to be a part of the intersection as well.

We therefore traverse the data-flow graph starting at a, and updating the variables access sets as above, until they stabilize—i.e., stop changing.[2] During the traversal, necessary stack depths are also updated. The requirements for compatible bytecode crossover can now be specified.

Bytecode Constraints on Crossover. In order to attain viable offspring, several conditions must hold when performing crossover of two bytecode programs. Let A and B be functions in Java, represented as bytecode sequences. Consider segments α and β in A and B, respectively, and let p_α and p_β be the necessary depth of stack for these segments—i.e., the minimal number of

[2] The data-flow traversal process is similar to the data-flow analyzer's loop in (Lindholm and Yellin, 1999).

elements in the stack required to avoid underflow. Segment α can be replaced with β if the following conditions hold.

- Operand stack: (1) it is possible to ensure that $p_\beta \leqslant p_\alpha$ by prefixing stack pops and pushes of α with some frames from the stack state at the beginning of α; (2) α and β have compatible stack frames up to depth p_β: stack pops of α have identical or narrower types as stack pops of β, and stack pushes of β have identical or narrower types as stack pushes of α; (3) α has compatible stack frames deeper than p_β: stack pops of α have identical or narrower types as corresponding stack pushes of α.

- Local variables: (1) local variables written by β (β^w) have identical or narrower types as corresponding variables that are read after α (*post*-α^r); (2) local variables read after α (*post*-α^r) and not necessarily written by β ($\beta^{w!}$) must be written before α (*pre*-$\alpha^{w!}$), or provided as arguments for call to A, as identical or narrower types; (3) local variables read by β (β^r) must be written before α (*pre*-$\alpha^{w!}$), or provided as arguments for call to A, as identical or narrower types.

- Control flow: (1) no branch instruction outside of α has branch destination in α, and no branch instruction in β has branch destination outside of β; (2) code before α has transition to the first instruction of α, and code in β has transition to the first instruction after β; (3) last instruction in α implies transition to the first instruction after α.

Detailed examples of the above conditions can be found in (Orlov and Sipper, 2009).

Compatible bytecode crossover prevents verification errors in offspring, in other words, all offspring *compile* sans error. As with any other evolutionary method, however, it does not prevent production of non-viable offspring—in our case, those with runtime errors. An exception or a timeout can still occur during an individual's evaluation, and the fitness of the individual should be reset accordingly.

We chose bytecode segments randomly before checking them for crossover compatibility as follows: For a given method, a segment size is selected using a given probability distribution among all bytecode segments that are branch-consistent under the first control-flow requirement; then a segment with the chosen size is uniformly selected. Whenever the chosen segments result in *bad* crossover, bytecode segments are chosen again (up to some limit of retries). Note that this selection process is very fast (despite the retries), as it involves fast operations—and, most importantly, we ensure that crossover *always* produces a viable offspring.

```
float x;  int y = 7;
if (y >= 0)
    x = y;
else
    x = -y;
System.out.println(x);
```

(a)

```
int x = 7;  float y;
if (y >= 0) {
    y = x;
    x = y;
}
System.out.println(z);
```

(b)

Figure 1-5. Two Java snippets that comply with the context-free grammar rules of the programming language. However, only snippet (a) is legal once the full Java Language Specification (Gosling et al., 2005) is considered . Snippet (b), though Java-compliant syntactically, is revealed to be ill-formed when semantics are thrown into play.

5. The Grammar Alternative

One might ask whether it is really necessary to evolve bytecode in order to support the evolution of unrestricted Java software. After all, Java is a programming language with strict, formal rules, which are precisely defined in Backus-Naur form (BNF). One could make an argument for the possibility of providing this BNF description to a grammar evolutionary system (O'Neill and Ryan, 2003) and evolving away.

We disagree with such an argument. The apparent ease with which one might apply the BNF rules of a real-world programming language in an evolutionary system (either grammatical or tree-based) is an illusion stemming from the blurred boundary between *syntactic* and *semantic* constraints (Poli et al., 2008, ch. 6.2.4). Java's formal (BNF) rules are purely syntactic, in no way capturing the language's type system, variable visibility and accessibility, and other semantic constraints. Correct handling of these constraints in order to ensure the production of viable individuals would essentially necessitate the programming of a full-scale Java compiler—a highly demanding task, not to be taken lightly. This is not to claim that such a task is completely insurmountable—e.g., an extension to context-free grammars (CFGs), such as logic grammars, can be taken advantage of in order to represent the necessary contextual constraints (Wong and Leung, 2000). But we have yet to see such a GP implementation in practice, addressing real-world programming problems.

We cannot emphasize the distinction between syntax and semantics strongly enough. Consider, for example, the Java program segment shown in Figure 1-5(a). It is a seemingly simple syntactic structure, which belies, however, a host of semantic constraints, including: type compatibility in variable assignment, variable initialization before read access, and variable visibility. The similar (and CFG-conforming) segment shown in Figure 1-5(b) violates all these constraints: variable `y` in the conditional test is uninitialized during a read access, its subsequent assignment to `x` is type-incompatible, and variable `z` is undefined.

It is quite telling that despite the popularity and generality of grammatical evolution, we were able to uncover only a single case of evolution using a real-world, unrestricted phenotypic language—involving a semantically simple *hardware* description language (HDL). (Mizoguchi et al., 1994) implemented the complete grammar of SFL (Structured Function description Language) (Nakamura et al., 1991) as production rules of a rewriting system, using approximately 350(!) rules for a language far simpler than Java. The semantic constraints of SFL—an object-oriented, register-transfer-level language—are sufficiently weak for using its BNF directly:

> By designing the genetic operators based on the production rules and by performing them in the chromosome, a grammatically correct SFL program can be generated. This eliminates the burden of eliminating grammatically incorrect HDL programs through the evolution process and helps to concentrate selective pressure in the target direction. (Mizoguchi et al., 1994)

(Arcuri, 2009) recently attempted to repair Java source code using syntax-tree transformations. His JAFF system is not able to handle the entire language—only an explicitly defined subset (Arcuri, 2009, Table 6.1), and furthermore, exhibits a host of problems that evolution of correct Java bytecode avoids inherently: individuals are compiled at each fitness evaluation, compilation errors occur despite the *syntax*-tree modifications being legal (cf. discussion above), lack of support for a significant part of the Java syntax (inner and anonymous classes, labeled `break` and `continue` statements, Java 5.0 syntax extensions, etc.), incorrect support of method overloading, and other problems:

> The constraint system consists of 12 basic node types and 5 polymorphic types. For the functions and the leaves, there are 44 different types of constraints. For each program, we added as well the constraints regarding local variables and method calls. Although the constraint system is quite accurate, it does not completely represent yet all the possible constraints in the employed subset of the Java language (i.e., a program that satisfies these constraints would not be necessarily compilable in Java). (Arcuri, 2009)

FINCH, through its clever use of Java bytecode, attains a scalability leap in evolutionarily manageable programming language complexity.

6. The Halting Issue

An important issue that must be considered when dealing with the evolution of unrestricted programs is whether they halt—or not (Langdon and Poli, 2006). Whenever Turing-complete programs with arbitrary control flow are evolved, a possibility arises that computation will turn out to be unending. A program that has acquired the undesirable non-termination property during evolution is executed directly by the JVM, and FINCH has nearly no control over the process.

A straightforward approach for dealing with non-halting programs is to limit the execution time of each individual during evaluation, assigning a minimal fitness value to programs that exceed the time limit. This approach, however, suffers from two shortcomings: First, limiting execution time provides coarse-time granularity at best, is unreliable in the presence of varying CPU load, and as a result is wasteful of computer resources due to the relatively high time-limit value that must be used. Second, applying a time limit to an arbitrary program requires running it in a separate thread, and stopping the execution of the thread once it exceeds the time limit. However, externally stopping the execution is either unreliable (when interrupting the thread that must then eventually enter a blocked state), or unsafe for the whole application (when attempting to kill the thread).[3]

Therefore, in FINCH we exercise a different approach, taking advantage of the lucid structure offered by Java bytecode. Before evaluating a program, it is temporarily *instrumented* with calls to a function that throws an exception if called more than a given number of times (steps). A call to this function is inserted before each backward branch instruction and before each method invocation. Thus, an infinite loop in any evolved individual program will raise an exception after exceeding the predefined steps limit. Note that this is not a coarse-grained (run)time limit, but a precise limit on the number of steps.

7. (No) Loss of Compiler Optimization

Another issue that surfaces when bytecode genetic operators are considered is the apparent loss of compiler optimization. Indeed, most native-code producing compilers provide the option of optimizing the resulting machine code to varying degrees of speed and size improvements. These optimizations would presumably be lost during the process of bytecode evolution.

Surprisingly, however, bytecode evolution does *not* induce loss of compiler optimization, since there is no optimization to begin with! The common assumption regarding Java compilers' similarity to native-code compilers is simply incorrect. As far as we were able to uncover, with the exception of the IBM Jikes Compiler (which has not been under development since 2004, and which does not support modern Java), no Java-to-bytecode compiler is optimizing. Sun's Java Compiler, for instance, has not had an optimization switch since version 1.3.[4] Moreover, even the GNU Compiler for Java, which is part of the highly optimizing GNU Compiler Collection (GCC), does not optimize at the

[3] For the intricacies of stopping Java threads see `http://java.sun.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`.

[4] See the old manual page at `http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/javac.html`, which contains the following note in the definition of the `-O` (Optimize) option: *the `-O` option does nothing in the current implementation of `javac`.*

bytecode-producing phase—for which it uses the Eclipse Compiler for Java as a front-end—and instead performs (optional) optimization at the native code-producing phase. The reason for this is that optimizations are applied at a later stage, whenever the JVM decides to proceed from interpretation to just-in-time compilation (Kotzmann et al., 2008).

The fact that Java compilers do not optimize bytecode does not preclude the possibility of doing so, nor render it particularly hard in various cases. Indeed, in FINCH we apply an automatic post-crossover bytecode transformation that is typically performed by a Java compiler: dead-code elimination. After crossover is done, it is possible to get a method with unreachable bytecode sections (e.g., a forward `goto` with no instruction that jumps into the section between the `goto` and its target code offset). Such dead code is problematic in Java bytecode, and it is therefore automatically removed from the resulting individuals by our system. This technique does not impede the ability of individuals to evolve introns, since there is still a multitude of other intron types that can be evolved (Brameier and Banzhaf, 2007) (e.g., any arithmetic bytecode instruction not affecting the method's return value, which is not considered dead-code bytecode, though it is an intron nonetheless).

8. A Summary of Results

Due to space limitations we only provide a brief description of our results, with the full account available in (Orlov and Sipper, 2009; Orlov and Sipper, 2010). To date, we have successfully tackled several problems:

- *Simple and complex symbolic regression*: Evolve programs to approximate the simple polynomial $x^4 + x^3 + x^2 + x$ and the more complex polynomial $\sum_{i=1}^{9} x^i$.
- *Artificial ant problem*: Evolve programs to find all 89 food pellets on the Santa Fe trail.
- *Intertwined spirals problem*: Evolve programs to correctly classify 194 points on two spirals.
- *Array sum*: Evolve programs to compute the sum of values of an integer array, along the way demonstrating FINCH's ability to handle loops and recursion.
- *Tic-tac-toe*: Evolve a winning program for the game, starting from a *flawed* implementation of the negamax algorithm. This example shows that programs can be improved.

Figure 1-6 shows two examples of Java programs evolved by FINCH.

```
Number simpleRegression(Number num) {
  double d = num.doubleValue();
  return Double.valueOf(d + (d * (d * (d +
      ((d = num.doubleValue()) +
          (((num.doubleValue() * (d = d) + d)
              * d + d) * d + d) * d)
      * d) + d) + d) * d);
}
```

(a)

```
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;
}
```

(b)

Figure 1-6. Examples of evolved programs for the degree-9 polynomial regression problem (a), and the recursive array sum problem (b). The Java code shown was produced by decompiling the respective evolved bytecode solutions.

9. Concluding Remarks

A recent study commissioned by the US Department of Defense on the subject of futuristic ultra-large-scale (ULS) systems that have billions of lines of code noted, among others, that, "Judiciously used, digital evolution can substantially augment the cognitive limits of human designers and can find novel (possibly counterintuitive) solutions to complex ULS system design problems" (Northrop et al., 2006, p. 33). This study does not detail any actual research performed but attempts to build a road map for future research. Moreover, it concentrates on huge, futuristic systems, whereas our aim is at current systems of any size. Differences aside, both our work and this study share the vision of true software evolution.

Turing famously (and wrongly...) predicted that, "in about fifty years' time it will be possible, to programme computers [...] to make them play the imitation game so well that an average interrogator will not have more than 70 per cent. chance of making the right identification after five minutes of questioning" (Turing, 1950). Recently, Harman wrote that, "... despite its current widespread use, there was, within living memory, equal skepticism about whether compiled code could be trusted. If a similar change of attitude to evolved code occurs over time..." (Harman, 2010).

We wish to offer our own prediction for fifty years hence, in the hope that we shall *not* be wrong: We believe that in about fifty years' time it will be possible, to program computers by means of evolution. Not merely *possible* but indeed *prevalent*.

References

Arcuri, Andrea (2009). *Automatic Software Generation and Improvement Through Search Based Techniques*. PhD thesis, University of Birmingham, Birmingham, UK.

Brameier, Markus and Banzhaf, Wolfgang (2007). *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer.

Bruneton, Eric, Lenglet, Romain, and Coupaye, Thierry (2002). ASM: A code manipulation tool to implement adaptable systems (Un outil de manipulation de code pour la réalisation de systèmes adaptables). In *Adaptable and Extensible Component Systems (Systèmes à Composants Adaptables et Extensibles), October 17–18, 2002, Grenoble, France*, pages 184–195.

Engel, Joshua (1999). *Programming for the Java™ Virtual Machine*. Addison-Wesley, Reading, MA, USA.

Gosling, James, Joy, Bill, Steele, Guy, and Bracha, Gilad (2005). *The Java™ Language Specification*. The Java™ Series. Addison-Wesley, Boston, MA, USA, third edition.

Harman, Mark (2010). Automated patching techniques: The fix is in. *Communications of the ACM*, 53(5):108.

Kotzmann, Thomas, Wimmer, Christian, Mössenböck, Hanspeter, Rodriguez, Thomas, Russell, Kenneth, and Cox, David (2008). Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–32.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Langdon, W. B. and Poli, R. (2006). The halting probability in von Neumann architectures. In Collet, Pierre, Tomassini, Marco, Ebner, Marc, Gustafson, Steven, and Ekárt, Anikó, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 225–237, Budapest, Hungary. Springer.

Lindholm, Tim and Yellin, Frank (1999). *The Java™ Virtual Machine Specification*. The Java™ Series. Addison-Wesley, Boston, MA, USA, second edition.

Miecznikowski, Jerome and Hendren, Laurie (2002). Decompiling Java bytecode: Problems, traps and pitfalls. In Horspool, R. Nigel, editor, *Compiler Construction: 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8–12, 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127, Berlin / Heidelberg. Springer-Verlag.

Mizoguchi, Jun'ichi, Hemmi, Hitoshi, and Shimohara, Katsunori (1994). Production genetic algorithms for automated hardware design through an evolutionary process. In *Proceedings of the First IEEE Conference on Evolutionary Computation, ICEC'94*, volume 2, pages 661–664.

Nakamura, Yukihiro, Oguri, Kiyoshi, and Nagoya, Akira (1991). Synthesis from pure behavioral descriptions. In Camposano, Raul and Wolf, Wayne Hendrix, editors, *High-Level VLSI Synthesis*, pages 205–229. Kluwer, Norwell, MA, USA.

Northrop, Linda et al. (2006). *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Carnegie Mellon University, Pittsburgh, PA, USA.

O'Neill, Michael and Ryan, Conor (2003). *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers.

Orlov, Michael and Sipper, Moshe (2009). Genetic programming in the wild: Evolving unrestricted bytecode. In Raidl, Günther et al., editors, *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, July 8–12, 2009, Montréal Québec, Canada*, pages 1043–1050, New York, NY, USA. ACM Press.

Orlov, Michael and Sipper, Moshe (2010). Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*. In press.

Poli, Riccardo, Langdon, William B., and McPhee, Nicholas Freitag (2008). *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

Spector, Lee and Robinson, Alan (2002). Genetic programming and autoconstructive evolution with the Push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40.

Turing, Alan Mathison (1950). Computing machinery and intelligence. *Mind*, 59(236):433–460.

Wong, Man Leung and Leung, Kwong Sak (2000). *Data Mining Using Grammar Based Genetic Programming and Applications*, volume 3 of *Genetic Programming*. Kluwer, Norwell, MA, USA.

Woodward, John R. (2003). Evolving Turing complete representations. In Sarker, Ruhul et al., editors, *The 2003 Congress on Evolutionary Computation, CEC 2003, Canberra, Australia, 8–12 December, 2003*, volume 2, pages 830–837. IEEE Press.