Genetic Programming in the Wild: Evolving Unrestricted Bytecode

Michael Orlov
Department of Computer Science
Ben-Gurion University, PO Box 653
Beer-Sheva 84105, Israel
orlovm@cs.bgu.ac.il

Moshe Sipper
Department of Computer Science
Ben-Gurion University, PO Box 653
Beer-Sheva 84105, Israel
sipper@cs.bgu.ac.il

ABSTRACT

We describe a methodology for evolving Java bytecode, enabling the evolution of *extant*, *unrestricted* Java programs, or programs in other languages that compile to Java bytecode. Bytecode is evolved directly, without any intermediate genomic representation. Our approach is based upon the notion of *compatible crossover*, which produces correct programs by performing operand stack-, local variables-, and control flow-based compatibility checks on source and destination bytecode sections. 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. Given the huge universe of unrestricted Java bytecode, *as is* programs, our work enables the application of evolution within this realm. We experimentally validate our methodology both by extensively testing the correctness of compatible crossover on arbitrary bytecode, and by running evolution on a program that exploits the richness of the Java virtual machine architecture and type system.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation, program modification*;
D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Algorithms, Languages

Keywords

Java bytecode, software evolution

1. INTRODUCTION

Genetic programming is mostly used as a means to define a sophisticated genomic representation for a given problem, rather than to evolve a bona fide, full-fledged computer program. This is (arguably) true not only where traditional LISP-based GP is concerned, but also for other forms of GP, such as linear GP and grammatical evolution [18]. Herein, we propose a method 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—a highly popular programming language—or can be compiled to Java bytecode.

The established approach in genetic programming involves the definition of functions and terminals appropriate to the problem, after which evolution of expressions using these definitions takes place [7, 18]. This approach does not, however, suit us, since we want to be able to evolve extant Java programs. Evolving the source code directly is not a viable option, either. The source code is intended for humans to write and modify, and is thus abundant in syntactic constraints. This makes it very hard to produce viable offspring with enough variation to drive the evolutionary process. We therefore turn to yet another well-explored alternative: evolution of machine code [15].

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) [9]. 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 [3, 9]. Figure 1 shows a recursive Java program for computing the factorial of a number, and its corresponding bytecode.

The Java virtual machine architecture, illustrated in Fig. 2, is successful enough that numerous programming languages compile directly to Java bytecode. Moreover, Java decompilers are available that facilitate restoration of the Java source code from compiled bytecode. Since the design of 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 [13].

We choose 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-

GECCO'09, July 8–12, 2009, Montréal Québec, Canada.

```
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) The original Java source code. Each line is annotated with the corresponding code array offsets range.

```
 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) The compiled bytecode. Offsets in the code array are shown on the left.

Figure 1: A recursive factorial function in Java (a), and its corresponding bytecode (b). The argument to the virtual method invocation (#2) references the `int F.fact(int)` method via the constant pool.

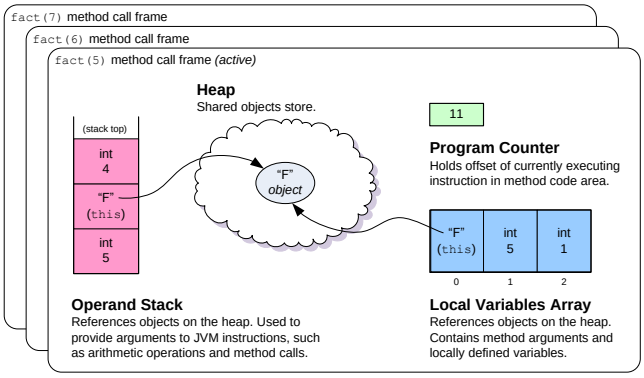

Figure 2: Call frames in the architecture of the Java virtual machine, during execution of the recursive factorial function code shown in Fig. 1, with parameter $n = 7$. The top call frame is in a state preceding execution of invokevirtual. This instruction will pop a parameter and an object reference from the operand stack, invoke the method `fact` of class `F`, and open a new frame for `fact(4)` call. When that frame closes, the returned value will be pushed onto the operand stack.

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 our intent here is *not to provide a killer application* (though we do provide a proof-of-concept example), but rather to describe in detail the *underlying methodology by which unrestricted bytecode can be evolved successfully.* Furthermore, we do not wish to invent a language to improve upon some aspect or other of genetic programming (efficiency, terseness, readability, etc.), as has been amply done (and partly summarized in Section 2.1). Nor do we wish to extend standard GP to become Turing complete, an issue which has also been addressed [23]. Rather, conversely, our point of departure is an *extant*, highly popular, general-purpose language, with our aim being to render it evolvable. We wish to bring evolution to the (Java) masses. The ability to evolve Java programs might lead to a valuable new tool in the software engineer's toolkit. This paper presents a sledgehammer—the nails will surely follow suit in future work.

Evolution of unrestricted bytecode is described in Section 2. The methodology is experimentally validated in Section 3. We present our conclusions in Section 4.

2. BYTECODE EVOLUTION

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 Spector and Robinson [20]—but rather aim to handle the Java programming language in particular.

Evolving the bytecode instead of the source code alleviates this issue, but not completely. Java bytecode must be *correct* with respect to dealing with stack and local variables (cf. Fig. 2). Values that are read and written should be type-compatible, and stack underflow must not occur. The Java virtual machine 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 JVM to perform genetic programming. Therefore, basic evolutionary operations, such as bytecode crossover and mutation, should produce correct individuals.

We define a *good* crossover of two parents as one where the offspring is a *correct* bytecode program, meaning one that compiles with no errors; conversely, a *bad* crossover of two parents is one where the offspring is an *incorrect* bytecode program, meaning one whose compilation 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 fitness value—or, alternatively, must be assigned a value of 0. Too many bad crossovers will hence produce a population with little variability, on whose vital role Darwin averred:

> If then we have under nature variability and a powerful agent always ready to act and select, why should we doubt that variations in any way useful to beings, under their excessively complex relations of life, would be preserved, accumulated, and inherited? [2]

Indeed, we performed empirical studies of good vs. bad crossover, concluding that the use of the latter is highly inef-

ficient (at least ten times slower). To wit, we have found that it is far more efficient (i.e., faster) to use a good crossover operator (as described below), rather than perform a series of bad crossovers until a good result chances to come along.

2.1 Previous Work

A number of researchers previously described bytecode evolution, though as an extension of the standard GP concept, namely, that of using bytecode, or some variant thereof, as a representation for solving a particular problem, rather than considering extant programs with the aim of evolving them directly. That is, none of the research surveyed below allows the treatment of existing unrestricted bytecode as the evolving genotype. It should be noted that some of the bytecode-related papers appeared as brief summaries, without peer review, during the time frame when Java started to gain popularity.

Stack-based genetic programming (Stack GP) was introduced by Perkis [17]. In Stack GP, instructions operate on a numerical stack, and whenever a stack underflow occurs (i.e., an argument for the operation is unavailable), the respective instruction is ignored. Whenever multiple data types are desired, multiple stacks are proposed as an alternative to strongly typed genetic programming [14]. Stack GP possesses a number of disadvantages with respect to our aims: First, ignoring stack underflows will produce incorrect bytecode segments with ambiguous decompilation results. Second, allowing such code will unnecessarily enlarge the search space, which is already huge—after all, we are evolving extant, real-world programs, and not evolving programs from scratch using a limited instruction set. Lastly, our approach assumes absolutely no control over the JVM architecture: we do not create stacks at will but are content with JVM's single multi-type data stack and general-purpose multi-type registers (see Fig. 2).

An early introduction to Java bytecode genetic programming (JBGP) was given by Lukschandl et al. [12], who evolved bytecode sequences with a small set of simple arithmetic and custom branch-like macro instructions. Lukschandl et al. evolved very limited individuals with a single floating-point type in one local variable and no control structures, and therefore only needed to consider effects of instruction blocks on operand stack depth in order to avoid stack overflow and underflow errors. A later work by Lukschandl et al. [11] used this method in a distributed bytecode evolutionary system (DJBGP), and presented its application to a telecom routing problem. Similar approaches were independently introduced by other researchers, as bcGP [4] (which also handles branching instructions, but does not discuss crossover compatibility) and Japhet [6] (which seems to leave compatibility checks to the Java verifier, although too few details are provided). The aforementioned approaches are conceptually limited to using Java bytecode as yet another genotype representation for GP. None can be applied to evolving correct individuals based on unrestricted bytecode—which we show how to do in the following section.

Once we consider non-bytecode stack-based GP, Tchernev [21] offered a more thorough treatment of requirements for crossover in the programming language Forth, arguing that ensuring same-stack depth at crossover points is not only better than GP's popular subtree crossover, but is an engine for combining building blocks that is strictly different from a macromutation. However, similar to the works discussed previously, Tchernev considers only the stack depth in synthetic individuals with restricted primitives. Tchernev and Phatak [22] later introduced a similar technique for correct crossover of high-level control structures. This work is not applicable at all to Java bytecode evolution, since control structures are not expressed as such in bytecode, and are instead translated into simpler **goto** instructions.

Evolutionary program induction using binary machine code (AIM-GP) was introduced by Nordin [15] as the fastest known genetic programming method. Although Nordin et al. [16] later mentioned Java as a possible evolutionary target, the paper is scarce on details. As of now, DISCIPULUS, the commercial successor to AIM-GP, can only produce Java source code as a decompilation result from an evolved native machine code individual, as opposed to our goal of evolving the intermediate-level bytecode. In AIM-GP, the creation of viable offspring individuals from parent programs is realized via a careful multi-granularity crossover process. It is interesting to contrast this work with the attempt by Kühling et al. [8] to forgo any constraints on code and on evolutionary operators, and instead trap all exceptions of code that is executed as a separate encapsulated entity. We do not expect this approach to overcome the huge search space that results from evolving Java programs with unrestricted crossover and mutation operators. However, since we decided that the evolutionary process should stay close to the JVM, we cannot completely safeguard bytecode execution from exceptional conditions, as done e.g, by Huelsbergen [5]. Thus, evaluating evolving bytecode individuals in an encapsulated environment—a *sandbox*—is still necessary.

More recently, Servant et al. [19] introduced JEB, an open-source tool for Java byte-code evolution, as an extension to the ECJ evolutionary computation software package [10]. In JEB, genotype and phenotype bytecodes are separate entities, and stack underflows are corrected during genotype-phenotype translation. Other limitations that we discussed previously—restricted instruction set, no handling of types, etc.—apply to this work as well. This is however an interesting approach—yet it is undesirable for evolving extant bytecode, since it introduces a separate representation for evolving programs and increases the search space. Reduction of search-space size is better achieved with properly-defined compatible evolutionary operators, as discussed next.

In summary, although all these works touched upon some aspect or other of Java bytecode (or, at least, machine code) evolution, they did so in a restricted way, the ultimate goal being that of affording a beneficial genomic representation for problem solving with genetic programming. Our departure point may be seen as one diametrically opposed: given the huge universe of unrestricted Java bytecode, *as is* programs, we aim to perform evolution within this realm.

2.2 Bytecode Evolution Principles

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 [3]. 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

Table 1: Operand stack and local variables array requirements during execution of the factorial method. An 'a' denotes a type-annotated object reference, and an 'i' denotes an integer type. Pop lists are given in reverse order. A list of types on the stack is given after each instruction, with stack top at the right of the list. x:y stands for read or write access to local variable x with type y. The 17 instructions are divided into four parts, each part corresponding to a single source line in Fig. 1(a). The argument to invokevirtual instruction (#2) references a value in the constant pool that resolves to the `int F.fact(int)` method signature.

Offset	Instruction	Description	Stack pops	Stack pushes	Stack state	Vars read	Vars written
0	**iconst_1**	push 1 on the stack		i	i		
1	**istore_2**	pop stack to the local variable **ans**	i		∅		2:i
2	**iload_1**	push n on the stack		i	i	1:i	
3	**ifle** 16	pop stack, and jump to **iload_2** if value $\leqslant 0$; note that the encoded offset is relative (+13)	i		∅		
6	**iload_1**	push n on the stack		i	i	1:i	
7	**aload_0**	push **this** on the stack		a/F	i, a/F	0:a/F	
8	**iload_1**	push n on the stack		i	i, a/F, i	1:i	
9	**iconst_1**	push 1 on the stack		i	i, a/F, i, i		
10	**isub**	pop two values, subtract, and push result	i, i	i	i, a/F, i		
11	**invoke-virtual #2**	pop object reference and parameter from the stack, and invoke virtual method; returned value is on the stack	a/F, i	i	i, i		
14	**imul**	pop two values, multiply, and push result	i, i	i	i		
15	**istore_2**	pop stack to the local variable **ans**	i		∅		2:i
16	**iload_2**	push the local variable **ans** on the stack		i	i	2:i	
17	**ireturn**	pop stack, and return value to the calling frame	i		∅		

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 Fig. 2

To demonstrate the operation of the JVM, consider a simple recursive program for computing the factorial of a number, shown in Fig. 1. Table 1 shows a step-by-step execution of the bytecode. The operand stack is initially empty, and the local variables array contains a reference to **this** (the current class instance) at index 0, and the parameter n at index 1. The local variable `ans` is allocated the index 2, but the corresponding cell is uninitialized.

In our evolutionary setup, the individuals are bytecode sequences annotated with all the stack and variables information shown in Table 1. This information is gathered in one pass over the bytecode, using the ASM bytecode manipulation and analysis library [1]. Afterwards, similar information for any sequential code segment in the individual can be aggregated separately—Table 2 shows this information for several bytecode segments. This preprocessing step allows us to realize compatible two-point crossover on bytecode sequences. 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 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 Fig. 1, which are reproduced as parents α and β in Fig. 3. We replace bytecode instructions at offsets 7–11 in parent α with the single **iload_2** instruction at offset 16 from parent β. Off-

Table 2: Operand stack and local variables array requirements for several bytecode segments of the compiled factorial method. The code array offsets are given according to Table 1. Object references are annotated with types that are inferred by data-flow analysis. Pop lists are given in reverse order—the topmost value is shown at the right-hand side. Note that the 14–17 fragment does not require a ready 2:i value, since write precedes read in this segment. A *potential* write, marked with '?', is not guaranteed to occur.

Offsets	Stack pops	Stack pushes	Vars read	Vars written
6–15			0:a/F, 1:i	2:i
8–15	i, a/F		1:i	2:i
3–10	i	i, a/F, i	0:a/F, 1:i	
14–17	i, i			2:i
3–15	i		0:a/F, 1:i	2:i?

[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 [9, §3.11.1].

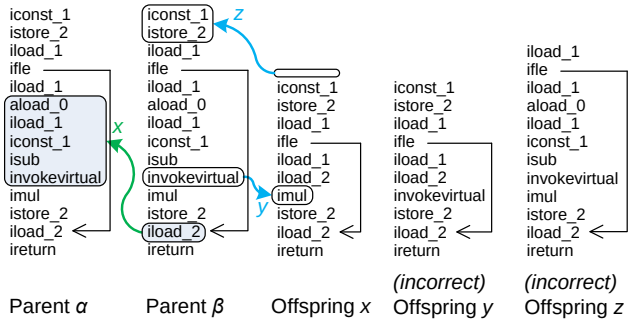

Figure 3: An example of good and bad crossovers. The two identical individuals α and β represent a recursive factorial function (see Fig. 1; here we use an arrow instead of branch offset). In parent α, the bytecode sequence that corresponds to the `fact(n-1)` call that leaves an integer value on the stack, is replaced with the single instruction in β that corresponds to pushing the local variable `ans` on the stack. The resulting correct offspring x and the original parent β are then considered as two new parents. We see that either replacing the first two instructions in β with an empty section, or replacing the imul instruction in x with the invokevirtual instruction from β, result in incorrect bytecode, shown as offspring y and z—see main text for full explanation.

```
// Queue initially contains instruction a
Q ← {a}
while Q ≠ ∅ do
    // Remove the front of queue
    c ← DEQUEUE(Q)
    // Sets of locations are initially empty
    {b_i} ← recorded locations branching to c
    compute δ*_{a,c} from {δ*_{a,b_i}} and δ*_c
    if δ*_{a,c} is new or changed then
        foreach c_j ∈ branch destinations of c do
            if ⟨c, c_j⟩ ≠ ⟨b, b+1⟩ then
                // Insert at end of queue
                Q ← ENQUEUE(Q, c_j)
                record c as location branching to c_j
```

Figure 4: COMPUTE-ACCESSES(a, b): A BFS-like traversal of the data-flow graph starting at location a and ending at b, in order to compute variables accessed in the code segment $[a, b]$. $\delta^*_{x,y}$ denotes the three access sets $\delta^r_{x,y}$, $\delta^w_{x,y}$, and $\delta^{w!}_{x,y}$. Here, a branch denotes natural transitions to subsequent instruction as well as transitions resulting from conditional and unconditional branching instructions. The inner if clause ensures that a "natural" transition at the end of segment $[a, b]$ is not unnecessarily followed.

sets 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 Fig. 3, 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 β. This crossover is *bad*, as illustrated by incorrect offspring y in Fig. 3. 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 β (that correspond to the `int ans=1` statement) with an empty segment. In this case, illustrated by incorrect offspring z in Fig. 3, variable `ans` is no longer guaranteed to be initialized when it is read immediately prior to the function's return, and the resulting bytecode is therefore incorrect.

The 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. We do not define a mutation operator in this paper—this we leave for future work.

2.3 Compatible Bytecode Crossover

As discussed in the beginning of the section, 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 variables 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.

2.3.1 *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; this set of variables is the *vars read* column in Table 2;
- $\delta^w_{a,b}$: set of local variables that are written to through at least one potential execution path; the corresponding column in Table 2 is *vars written*;
- $\delta^{w!}_{a,b}$: set of local variables that are guaranteed to be written to, no matter which execution path is taken; in Table 2, non-*potential* writes in the *vars written* column correspond to this set.

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:

Good	α	β
pre-stack	`****`	`****`
post-stack	`***`	`***`
depth	1	2

(a) Case 1(a). Whereas β has necessary stack depth of 2 (two pops and one push), α has a stack depth of 1 (one pop). However, α has more stack available, and can be viewed as having a stack depth of 2.

Bad	α	β
pre-stack	`*`	`****`
post-stack	$\emptyset$	`***`
depth	1	2

(b) Case 1(a). Here, α cannot be viewed as having a stack depth of 2, since the whole stack depth before α is 1.

Good	α	β
pre-stack	`**AB`	`**AA`
post-stack	`**B`	`**C`
depth	3	2

(c) Case 1(b). Stack pops "`AB`" (2 stack frames) are narrower than "`AA`", whereas stack push "`C`" is narrower than "`B`".

Bad	α	β
pre-stack	`**AB`	`**Af`
post-stack	`**B`	`**A`
depth	3	2

(d) Case 1(b). Stack pops "`AB`" are not narrower than "`Af`", since the object reference `B` and the primitive type `f` are incompatible. Also, stack push "`A`" is not narrower than "`B`".

Good	α	β
pre-stack	`iB**`	`****`
post-stack	`iA**`	`****`
depth	4	2

(e) Case 1(c). Stack pops "`iB`" (extra 2 stack frames) are narrower than stack pushes "`iA`".

Bad	α	β
pre-stack	`*A**`	`***`
post-stack	`*B**`	`***`
depth	3	2

(f) Case 1(c). Stack pop "`A`" (extra 1 stack frame) is not narrower than stack push "`B`".

Figure 5: Illustrations to the operand stack requirements in bytecode crossover constraints. Here, we assume that class `B` extends class `A`, and `B` is thus a narrower type than `A`, and that class `C` similarly extends class `B`. The symbols `i` and `f` denote the primitive types `int` and `float`. The `*` symbol is used in cases where the precise type does not matter. For stacks, the topmost value is shown at the right-hand side. *Pre-stack* and *post-stack* are states of stack before and after execution of code segment.

- $\delta^r_{a,c}$ 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 $\delta^{w!}_c = \delta^w_c$).

We therefore traverse the data-flow graph as shown in Fig. 4, 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—such as the number of pops in Table 2—are also updated. The requirements for compatible bytecode crossover can now be specified.

[2]The data-flow traversal is similar in nature to the data-flow analyzer's loop in [9, §4.9.2].

Good	
β^w	2:`i`, 3:`C`, 1:`*`
post-α^r	2:`i`, 3:`B`, 4:`*`

(a) Case 2(a). Variables 2:`i`, 3:`C` that can be written by β are narrower than variables 2:`i`, 3:`B` that can be read after α.

Bad	
β^w	2:`i`, 3:`A`
post-α^r	2:`f`, 3:`B`

(b) Case 2(a). Variable 2:`i` is not compatible with 2:`f`, and variable 3:`A` is not narrower than 3:`B`.

Good	
pre-$\alpha^{w!}$	2:`*`, 1:`C`, 4:`f`
$\beta^{w!}$	2:`*`, 3:`*`
post-α^r	2:`*` 3:`*` 1:`A` 4:`f`

(c) Case 2(b). Variables 1:`A`, 4:`f` that can be read after α and are not necessarily written by β, are certainly written before α as narrower types 1:`C`, 4:`f`.

Bad	
pre-$\alpha^{w!}$	2:`f`, 1:`A`
$\beta^{w!}$	3:`*`
post-α^r	2:`i` 3:`*` 1:`B` 4:`*`

(d) Case 2(b). Variable 2:`f` is not compatible with 2:`i`, variable 1:`A` is not narrower than 1:`B`, and variable 4 is not necessarily written either by β or before α.

Good	
pre-$\alpha^{w!}$	2:`i`, 3:`C`, 1:`*`
β^r	2:`i`, 3:`B`

(e) Case 2(c). Variables 2:`i`, 3:`B` that can be read by β, are necessarily written before α as narrower types 2:`i`, 3:`C`.

Bad	
pre-$\alpha^{w!}$	2:`i`, 3:`C`, 1:`*`
β^r	2:`f`, 3:`B`, 4:`*`

(f) Case 2(c). Variable 2:`i` is not compatible with 2:`f`, and variable 4 is not necessarily written before α.

Figure 6: Illustrations to the local variables requirements in bytecode crossover constraints. Notation and types used are similar to Fig. 5; x:`y` stands for read or write access to local variable x with type `y`. For example, 2:`i` in β^r means that segment β reads variable 2 as an `int`.

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

1. Operand stack (illustrated in Fig. 5):
   (a) 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;
   (b) 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;
   (c) α has compatible stack frames deeper than p_β: stack pops of α have identical or narrower types as corresponding stack pushes of α.
2. Local variables (illustrated in Fig. 6):
   (a) local variables written by β (β^w) have identical or narrower types as corresponding variables that are read after α (*post-*α^r);
   (b) local variables read after α (*post-*α^r) and not necessarily written by β ($\beta^{w!}$) must be written before

```
class Gecco {
  Number simpleRegression(Number num) {
    double x     = num.doubleValue();
    double llsq  = Math.log(Math.log(x*x));
    double dv    = x / (x - Math.sin(x));
    double worst = Math.exp(dv - llsq);
    return Double.valueOf(worst + Math.cos(1));
/* Rest of class omitted */ }}
```

Figure 7: *Simple Symbolic Regression* in Java. Worst-of-generation individual in generation 0 of the $x^4 + x^3 + x^2 + x$ regression experiment of Koza [7, ch. 7.3], as freely translated by us into a Java instance method with primitive and reference types. Since the archetypal individual (EXP (- (% X (- X (SIN X))) (RLOG (RLOG (* X X))))) does not contain the complete function set $\{+, -, *, \%, \mathtt{SIN}, \mathtt{COS}, \mathtt{EXP}, \mathtt{RLOG}\}$, we added a smattering of extra code in the last line, providing analogs of + and COS, and, incidentally, the constant 1. Protecting function arguments is unnecessary in our approach.

```
class Gecco {
  Number simpleRegression(Number num) {
    double d = num.doubleValue();
    d = num.doubleValue();
    double d1 = d; d = Double.valueOf(d + d * d *
          num.doubleValue()).doubleValue();
    return Double.valueOf(d +
          (d = num.doubleValue()) * num.doubleValue());
/* Rest of class unchanged */ }}
```

Figure 8: Decompiled contents of method simpleRegression that evolved after 17 generations from the Java program in Fig. 7. It is interesting to observe that because the evolved bytecode does not adhere to the implicit rules by which typical Java compilers generate code, the decompiled result is slightly incorrect: the assignment to variable d in the return statement occurs *after* it is pushed on the stack. This is a quirk of the decompiler—the evolved bytecode is perfectly correct and functional. The computation thus proceeds as $(x+x\cdot x\cdot x)+(x+x\cdot x\cdot x)\cdot x$, where x is the method's input.

α (*pre*-$\alpha^{w!}$), or provided as arguments for call to A, as identical or narrower types;

(c) 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.

3. Control flow:
   (a) no branch instruction outside of α has branch destination in α, and no branch instruction in β has branch destination outside of β;
   (b) code before α has transition to the first instruction of α, and code in β has transition to the first instruction after β.

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, 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.

3. EXPERIMENTAL VALIDATION

In order to validate our approach to evolving unrestricted Java bytecode, we need to ensure that whenever a potential unidirectional crossover between two programs is judged compatible, as outlined in Section 2.3, the resulting individuals are correct—i.e., pass verification ("compile") when the new program is loaded by the Java virtual machine. It is easy to see that the requirements for compatible crossover are necessary, as illustrated by "*bad*" counterexamples in Figs. 5 and 6. However, no amount of formalism will convince us that these requirements are sufficient to handle replacement of segments across genuine unrestricted functions compiled to bytecode.

We have extensively tested evolutionary operators on unrestricted bytecode with arbitrarily complex Java class files, leaving out nary a sophisticated Java feature: control flow using loops, **if**, **switch** statements, variables of different types exploiting the hierarchy of types and interfaces, object creation and construction using the **new** operator, and so on—all were tested. We have also manually written bytecode using the Jasmin Java assembler, to reach borderline cases unattainable by compiling from regular Java source code—thus ensuring that the methodology does not assume specific patterns of bytecode generation. The resulting individuals were then loaded and instantiated using the Java class loader. In all cases, without exception, instantiation passed without a verification error, implying that handling of types on operand stack and in local variables is correct. We are thus confident that our methodology is an industrial-strength design, applicable to automatic programming within the big, wild universe of extant Java bytecode programs.

We also need to test the *feasibility* of bytecode evolution. That is, we need to see that evolution of unrestricted bytecode can be driven by the compatible crossover operator. For this purpose, we integrated our framework, which uses ASM [1], with the ECJ evolutionary framework [10]. We then considered a classic test case in GP—namely, *simple symbolic regression* [7], where individuals with a single numeric input and output are tested on their ability to approximate the polynomial $x^4 + x^3 + x^2 + x$ on 20 random samples. Since our approach needs an existing program, we initialized the initial population with copies of a single individual translated into Java. To be sure, we picked the *worst possible* individual, as described in Fig. 7.

To remain faithful to the original experiment, we used the same parameters where possible: a population of 500 individuals, crossover probability of 0.9, and no mutation. To facilitate implementation we used tournament selection with tournament size 2 instead of fitness-proportionate selection, which was used originally. We chose bytecode segments randomly before checking them for crossover compatibility. An ideal individual was found in every run. A typical evolutionary result, shown in Fig. 8, establishes without a doubt: unrestricted bytecode evolution *works* as a comprehensive evolutionary computation method.

4. CONCLUSIONS

We presented a powerful tool by which *extant* software, written in the Java programming language, or in a language

that compiles to Java bytecode, can be evolved *directly*, without an intermediate genomic representation, and with *no restrictions* on the constructs used. We introduced *compatible crossover*, a fundamental evolutionary operator that produces correct programs by performing operand stack-, local variables-, and control flow-based compatibility checks on source and destination bytecode sections.

Are compatible crossover requirements necessary for evolving correct bytecode? After all, the JVM includes a verifier that signals upon instantiation of a problematic class, a condition easily detected. There are several reasons that compatible evolutionary operators are crucial to unrestricted bytecode evolution. One reason is that precluding bad crossovers avoids synthesizing, loading, and verifying a bad individual. In measurements we performed, the naive approach (allowing bad crossover) is at least ten times slower than our unoptimized implementation of compatible crossover. However, this reason is perhaps the least important. Once we rely on the JVM verifier to select compatible bytecode segments, we lose all control over which segments are considered consistent. The built-in verifier is more permissive than strictly necessary, and will thus overlook building blocks in given bytecode. Moreover, the evolutionary computation practitioner might want to implement stricter requirements on crossover, or select alternative segments *during* compatibility checking—all this is impossible using the naive verifier-based approach.

We intend to pursue two important avenues of research that present themselves. First, we aim to define a process by which consistent bytecode segments can be *found* during compatibility checks, thus improving preservation of building blocks during evolution. Second, we wish to apply unrestricted bytecode evolution to the automatic improvement of existing applications, establishing the relevance of this methodology to the realm of extant software.

Acknowledgments

Michael Orlov is supported by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities, and is partially supported by the Lynn and William Frankel Center for Computer Sciences.

5. REFERENCES

[1] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems, Oct. 17–18, Grenoble, France*, pp. 184–195, 2002.

[2] C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, London, 1859.

[3] J. Engel. *Programming for the Java^{TM} Virtual Machine*. Addison-Wesley, Reading, MA, USA, 1999.

[4] B. Harvey, J. Foster, and D. Frincke. Towards byte code genetic programming. In W. Banzhaf et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, Orlando, FL, USA, July 13–17*, vol. 2, p. 1234, 1999. Morgan Kaufmann.

[5] L. Huelsbergen. Fast evolution of custom machine representations. In D. Corne et al., editors, *The 2005 IEEE Congress on Evolutionary Computation, 2–5 Sep., Edinburgh, Scotland, UK*, vol. 1, pp. 97–104. IEEE Press.

[6] S. Klahold, S. Frank, R. E. Keller, and W. Banzhaf. Exploring the possibilites and restrictions of genetic programming in Java bytecode. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference, Madison, WI, USA, July 22–25*, pp. 120–124, 1998. Omni Press.

[7] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, MA, USA, 1992.

[8] F. Kühling, K. Wolff, and P. Nordin. A brute-force approach to automatic induction of machine code on CISC architectures. In J. A. Foster et al., editors, *Genetic Programming: 5th European Conference, EuroGP 2002, Kinsale, Ireland, Apr. 3–5*, vol. 2278 of *LNCS*, pp. 288–297, 2002. Springer-Verlag.

[9] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, Boston, MA, USA, second edition, 1999.

[10] S. Luke and L. Panait. A Java-based evolutionary computation research system. Online, 2004. `http://cs.gmu.edu/~eclab/projects/ecj`.

[11] E. Lukschandl, H. Borgvall, L. Nohle, M. Nordahl, and P. Nordin. Distributed Java bytecode genetic programming with telecom applications. In R. Poli et al., editors, *Genetic Programming: European Conference, EuroGP 2000, Edinburgh, Scotland, UK, Apr. 15–16*, vol. 1802 of *LNCS*, pp. 316–325, 2000. Springer-Verlag.

[12] E. Lukschandl, M. Holmlund, E. Modén, M. Nordahl, and P. Nordin. Induction of Java bytecode with genetic programming. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference, Madison, WI, USA, July 22–25*, pp. 135–142, 1998. Omni Press.

[13] J. Miecznikowski and L. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In R. N. Horspool, editor, *Compiler Construction: 11th International Conference, CC 2002, Grenoble, France, Apr. 8–12*, vol. 2304 of *LNCS*, pp. 111–127, 2002. Springer-Verlag.

[14] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[15] P. Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications*. Krehl Verlag, Münster, Germany, 1997.

[16] P. Nordin, W. Banzhaf, and F. D. Francone. Efficient evolution of machine code for CISC architectures using blocks and homologous crossover. In L. Spector et al., editors, *Advances in Genetic Programming*, vol. 3, chap. 12, pp. 275–299. The MIT Press, Cambridge, MA, USA, 1999.

[17] T. Perkis. Stack-based genetic programming. In Z. Michalewicz et al., editors, *Proceedings of the First IEEE Conference on Evolutionary Computation, June 27–29, Orlando, FL, USA*, vol. 1, pp. 148–153. IEEE Neural Networks, 1994.

[18] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK, Mar. 2008.

[19] F. Servant, D. Robilliard, and C. Fonlupt. JEB: Java evolutionary byte-code — implementation and tests. In *Artificial Evolution, 7th International Conference, EA 2005, Lille, France, Oct. 26–28*, 2005.

[20] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the Push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, 2002.

[21] E. B. Tchernev. Forth crossover is not a macromutation? In J. R. Koza et al., editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference, July 22–25, Madison, WI, USA*, pp. 381–386, 1998. Morgan Kaufmann.

[22] E. B. Tchernev and D. S. Phatak. Control structures in linear and stack-based genetic programming. In M. Keijzer, editor, *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference, June 26–30, Seattle, WA, USA*. Distributed on CD-ROM, 2004.

[23] J. R. Woodward. Evolving Turing complete representations. In R. Sarker et al., editors, *The CEC 2003 Congress on Evolutionary Computation, Canberra, Australia, 8–12 Dec.*, vol. 2, pp. 830–837. IEEE Press.