Flight of the FINCH through the Java Wilderness

Michael Orlov and Moshe Sipper

***Abstract*—We describe Fertile Darwinian Bytecode Harvester (FINCH), a methodology for evolving Java bytecode, enabling the evolution of *extant*, *unrestricted* Java programs, or programs in other languages that compile to Java bytecode. Our approach is based upon the notion of *compatible crossover*, which produces correct programs by performing operand stack-based, local variables-based, 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. We demonstrate FINCH's unqualified success at solving a host of problems, including simple and complex regression, trail navigation, image classification, array sum, and tic-tac-toe. FINCH exploits the richness of the Java virtual machine architecture and type system, ultimately evolving human-readable solutions in the form of Java programs. The ability to evolve Java programs will hopefully lead to a valuable new tool in the software engineer's toolkit.**

***Index Terms*—Automatic programming, genetic programming (GP), Java bytecode, software evolution.**

I. Introduction

In a recent comprehensive monograph surveying the field of genetic programming (GP), Poli *et al.* noted the following.

> 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 [27, 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 [27].

In this paper, 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 GP involves the definition of functions and terminals appropriate to the problem at hand, after which evolution of expressions using these definitions takes place [11], [27]. This approach does not, however, suit us, since we seek 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 (more on this in Section II-B). We therefore turn to yet another well-explored alternative: evolution of machine code [22].

Java compilers almost never produce machine code directly, but instead compile source code to platform-independent *bytecode*, to be interpreted in software or, rarely, to be executed in hardware by a Java virtual machine (JVM) [15]. The JVM is free to apply its own optimization techniques, such as just-in-time, 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 [7], [15]. Fig. 1 shows a recursive Java program for computing the factorial of a number, and its corresponding bytecode.

The JVM architecture, illustrated in Fig. 2, 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 [18].

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. This paper extends significantly upon our preliminary results reported in [25].

Manuscript received November 17, 2009; revised February 17, 2010, April 27, 2010, and May 26, 2010. Date of publication January 10, 2011; date of current version March 30, 2011. The work of M. Orlov is supported in part by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities, and in part by the Lynn and William Frankel Center for Computer Sciences.

The authors are with the Department of Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel (e-mail: orlovm@cs.bgu.ac.il; sipper@cs.bgu.ac.il).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TEVC.2010.2052622

```
class F {                          0 iconst_1
  int fact(int n) {                1 istore_2
    // offsets 0-1                 2 iload_1
    int ans = 1;                   3 ifle 16
                                   6 iload_1
    // offsets 2-3                 7 aload_0
    if (n > 0)                     8 iload_1
      // offsets 6-15              9 iconst_1
      ans = n *                   10 isub
        fact(n-1);                11 invokevirtual #2
                                   14 imul
    // offsets 16-17               15 istore_2
    return ans;                    16 iload_2
}}                                 17 ireturn
```

(a) (b)

Fig. 1. Recursive factorial function in (a) Java, and its (b) corresponding bytecode. The argument to the virtual method invocation (#2) references the `int F.fact(int)` method via the constant pool. (a) Original Java source code. Each line is annotated with the corresponding code array offsets range. (b) Compiled bytecode. Offsets in the code array are shown in (a).

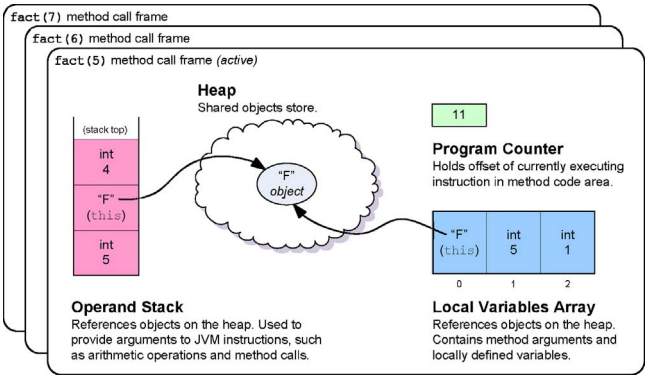


Fig. 2. Call frames in the architecture of the JVM, during execution of the recursive factorial function code shown in Fig. 1, with parameter $n = 7$. The top call frame is in a state preceding execution of **invokevirtual**. This instruction will pop a parameter and an object reference from the operand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

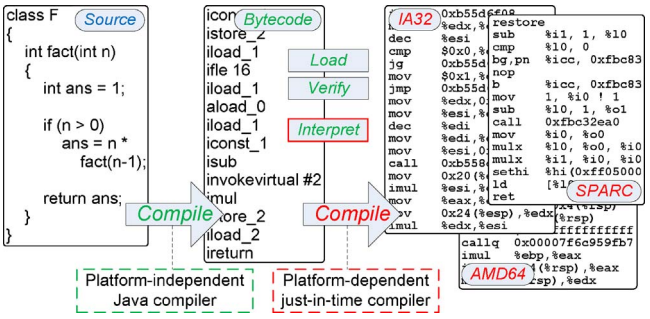


Fig. 3. Java source code is first compiled to *platform-independent* bytecode by a Java compiler. The JVM only loads the bytecode, which it verifies for correctness, and raises an exception in case the verification fails. After that, the JVM typically interprets the bytecode until it detects that it 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 (see Section II-D).

Note that we do not wish to invent a language to improve upon some aspect or other of GP (efficiency, terseness, readability, and so on), as has been amply done (and partly summarized in Section II-E and [25]). Nor do we wish to extend standard GP to become Turing-complete, an issue which has also been addressed [34]. 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 FINCH system, which affords the evolution of unrestricted bytecode, is described in Section II. Section III then presents the application of FINCH to evolving solutions to several hard problems: simple and complex regression, trail navigation, intertwined spirals (image classification), array sum, and tic-tac-toe. We end with concluding remarks and future work in Section IV.

II. Bytecode Evolution

Bytecode is the intermediate, platform-independent representation of Java programs, created by a Java compiler. Fig. 3 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.

A. Why Target Bytecode for 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 [30]—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 (see Fig. 2). 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. Below we provide a summary of our previous work on defining bytecode crossover—full details are available in [25].

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 fitness value—or, alternatively, must be assigned the worst possible value. 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 [6]?

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 [25] is one mechanism by which good crossover can be implemented.

As an example of compatible crossover, consider two identical programs with the same bytecode as in Fig. 1, which are reproduced as parents A and B in Fig. 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 Fig. 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 Fig. 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 Fig. 4, 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.

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 [25]; then a segment with the chosen size is uniformly selected. Whenever the chosen segments result in *bad*

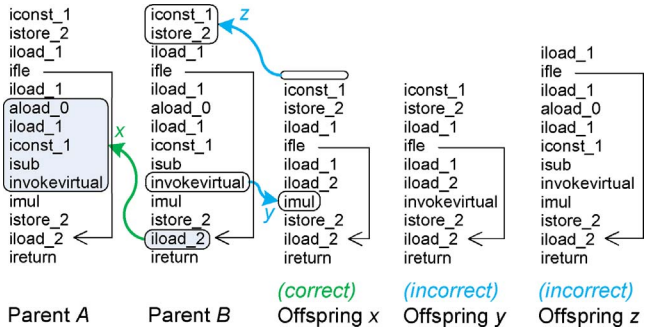


Fig. 4. Example of good and bad crossovers. The two identical individuals A and B represent a recursive factorial function (see Fig. 1; here we use an arrow instead of branch offset). In parent A, the bytecode sequence that corresponds to the **fact**$(n-1)$ call that leaves an integer value on the stack, 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.

```
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)

Fig. 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 [9] is considered. Snippet (b), though Java-compliant syntactically, is revealed to be ill-formed when semantics are thrown into play. (a) Correct Java snippet. (b) Incorrect Java snippet.

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. More intelligent choices of bytecode segments are possible, as will be discussed in Section IV.

The full formal specification of compatible bytecode crossover is provided in [25]. Note that we have not implemented (nor found necessary for the problems tackled so far) sophisticated mutation operators, a task we leave for future work, as described in Section IV. Only a basic constant mutator (Section III-C) was implemented.

B. 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 [24] 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 bound-

ary between *syntactic* and *semantic* constraints [27, 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 [33]. 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 Fig. 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 Fig. 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.* [19] implemented the complete grammar of structured function description language (SFL) [21] 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. [19]

Arcuri [2] 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 (see [2, 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 (see the 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 five 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) [2].

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

C. Halting Issue

An important issue that must be considered when dealing with the evolution of unrestricted programs is whether they halt—or not [14]. 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 central processing unit (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).[1]

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). That is, 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.

D. (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

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

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.[2] Moreover, even the GNU compiler for Java, which is part of the highly optimizing GNU Compiler Collection (GCC), does not optimize at the 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 [10].

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 [3] (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).

E. Related Work

In [25], we discussed at length several related works. Herein we complement that discussion by elaborating upon a number of works which we did not previously cover (due to space restrictions, or simply because they appeared after the publication of [25]).

Stack-based GP (stack GP) was introduced by Perkis [26]. In stack GP, instructions operate on a numerical stack, and whenever a stack underflow occurs (i.e., an operand 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 GP [20]. 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 content ourselves with JVM's single multi-type data stack and general-purpose multi-type registers (see Fig. 2).

Spector and Robinson [30] provide an interesting treatment of a stack-based architecture using the elaborately designed, expressive, Turing-complete Push programming language, which also supports autoconstructive evolution (where individual programs are responsible for the construction of their own children). Push maintains multiple type-specific stacks, while at the same time placing virtually no syntax restrictions on the evolving code. For example, instructions with an insufficient number of operands on the stack are simply ignored, following the "permissive execution" *modus operandi*. Our above remarks regarding stack GP [26] mostly apply to [30] as well, given the similarity of the two approaches. Moreover, the stack-per-type approach cannot handle evolution of object-oriented programs with hierarchical types very well.

Another line of recent research related to ours is that of software repair by evolutionary means. Forrest *et al.* [8] automatically repair C programs by evolving abstract syntax tree nodes on the faulty execution path, where at least one negative test case is assumed to be known. The resulting programs are then compiled before evaluation. Unlike FINCH, which works with individuals in compiled form while seamlessly handling semantic bytecode constraints, the approach by Forrest *et al.* is bound to be problematic when handling large faulty execution paths, or multiple-type, object-oriented programs. Additionally, finding precise negative test cases highlighting a short faulty execution path is typically the most difficult debugging problem a human programmer faces—fixing the bug therefrom is usually straightforward. This approach is not an alternative to FINCH, therefore, which can take an existing program as a whole, and evolutionarily improve it—free from various compilability requirements, which are relevant to abstract syntax trees, but not to Java bytecode. We shall demonstrate this capability of FINCH in Section III-E, where the programmer is in possession of only an approximate implementation of an optimal algorithm—a "correct" execution path does not exist prior to the evolutionary process.

There is also the recent work by Arcuri [2], mentioned earlier in Section II-B.

III. Results

We now turn to testing the feasibility of bytecode evolution, i.e., we need to support our hypothesis that evolution of unrestricted bytecode can be driven by the compatible crossover operator proposed in [25]. For this purpose we integrated our framework, which uses ASM [4], with the ECJ evolutionary framework [16], with ECJ providing the evolutionary engine. Now we are ready to apply FINCH to a selection of problems of interest.

Throughout this section we describe typical results (namely, Java programs) produced by our evolutionary runs. The consistency of FINCH's successful yield is shown in Table I, wherein we provide a summary of all runs performed. Table I

[2]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`.*

TABLE I
SUMMARY OF EVOLUTIONARY RUNS

Section	Problem		Yield
III-A	Simple symbolic regression		99%
	Complex symbolic regression		100%
III-B	Artificial ant on Santa Fe trail		2%
		Population of 2000	11%
		Population of 10 000, 101 generations	35%
III-C	Intertwined spirals		10%
III-D	Array sum		77%
		List-based seed individual	97%
		List-recursive seed individual	100%
III-E	Tic-tac-toe	`alpha/save imperfection`	96%
		`unary "-" imperfection`	88%
		`false/save imperfection`	100%
		`alpha-beta imperfection`	88%

Yield is the percentage of runs wherein a successful individual was found (where "success" is measured by the *success predicate*, defined for each problem in the appropriate section). The number of runs per problem was 100, except for 25 runs for each tic-tac-toe imperfection.

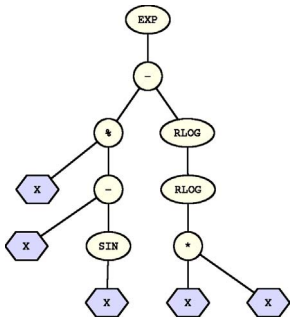


Fig. 6. Tree representation of the worst generation-0 individual in the original *simple symbolic regression* experiment of Koza [11]. Functions are represented by inner tree nodes, and terminals by leaves. The corresponding mathematical expression for $x \neq 0, 1$ is $e^{\frac{x}{x-\sin x} - \ln|\ln x^2|}$, due to protected division and logarithm operators `%` and `RLOG`. These operators protect against 0 in denominators and logarithms (`RLOG` protects against negative arguments as well).

shows that the run yield of FINCH, i.e., the fraction of successful runs, is high, thereby demonstrating both consistency and repeatability. No significant difference in evolutionary execution time was found between our bytecode evolution and reported typical times for tree-based GP (most of the runs described herein took a few minutes on a dual-core 2.6 GHz Opteron machine, with the exception of tic-tac-toe, which took on the order of one hour; note that this latter would also be quite costly using tree-based GP due to the game playing-based fitness function [29]). Of course, increasing population size (see Table I) incurred a time increase.

A. Symbolic Regression: Simple and Complex

We begin with a classic test case in GP—*simple symbolic regression* [11]—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. FINCH needs an existing program to begin with, so we *seeded* the initial population with copies of a single individual [13], [27], [28]. We selected the *worst* generation-0 individual in Koza's original experiment, shown in Fig. 6, and translated it into Java (Fig. 7).

TABLE II
SIMPLE SYMBOLIC REGRESSION: PARAMETERS

Parameter	Koza [11, ch. 7.3]	FINCH
Objective	Symbolic regression: $x^4 + x^3 + x^2 + x$	
Fitness	Sum of errors on 20 random data points in $[-1, 1]$	
Success predicate	All errors are less than 0.01	
Input	`X` (a terminal)	**Number** num
Functions	+, −, ∗, % (protected division), `SIN, COS, EXP, RLOG` (protected log)	Built-in arithmetic and **Math** functions present in the seed individual (Fig. 7)
Population	500 individuals	
Generations	51, or less if ideal individual was found	
Probabilities	$p_{\text{cross}} = 0.9$, $p_{\text{mut}} = 0$	
Selection	Fitness-proportionate	Binary tournament
Elitism	*Not used*	
Growth limit	Tree depth of 17	*No limit*
Initial population	Ramped half-and-half with maximal depth 6	Copies of seed program given in Fig. 7
Crossover location	Internal nodes with $p_{\text{int}} = 0.9$, otherwise a terminal	Uniform distribution over segment sizes

```
class SimpleSymbolicRegression {
  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 */ }}
```

Fig. 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 [11], as 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 {+, −, *, %, SIN, COS, EXP, 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 (enforcement of closure) is unnecessary in FINCH because evaluation of bytecode individuals uses Java's built-in exception-handling mechanism.

The worst generation-0 individual (Fig. 6) is the Lisp S-expression

```
(EXP (- (% X (- X (SIN X))) (RLOG (RLOG (* X X)))))
```

where `X` is the numeric input (a terminal), and {+, −, *, %, SIN, COS, EXP, RLOG} represents the function set. Whereas the function set includes protected division and logarithm operators `%` and `RLOG`, FINCH needs no protection of functions, since evaluation of bytecode individuals uses Java's built-in exception-handling mechanism. Therefore, individuals can be coded in the most straightforward manner. However, to demonstrate the capability of handling different primitive and reference types, we added an additional constraint whereby the **`simpleRegression`** method accepts and returns a general **Number** object. Moreover, in order to match the original experiment's function set, we added extra code that corresponds to the + and `COS` functions. In classical (tree-based) GP, the function and terminal sets must be *sufficient* in order to drive the evolutionary process; analogously, in FINCH, the initial (cloned) individual must contain a sufficient mixture of primitive components—the

```
class SimpleSymbolicRegression_0_7199 {
  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 */ }}
```

Fig. 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 onto 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.

```
class SimpleSymbolicRegression_0_2720 {
  Number simpleRegression(Number num) {
    double d = num.doubleValue();
    d = d; d = d;
    double d1 = Math.exp(d - d);
    return Double.valueOf(num.doubleValue() *
      (num.doubleValue() *
        (d * d + d) + d) + d);
/* Rest of class unchanged */ }}
```

Fig. 9. Decompiled contents of method **simpleRegression** that evolved after 13 generations in another experiment. Here, the evolutionary result is more straightforward, and the computation proceeds as $x \cdot (x \cdot (x \cdot x + x) + x) + x$, where x is the method's input. (Note: Both here and in Fig. 8, the name of the num parameter produced by the decompiler was different—and manually corrected by us—since we do not preserve debugging information during bytecode evolution; in principle, this adjustment could be done automatically.)

bytecode equivalents of function calls, arithmetic operations, conditional operators, casts, and so forth.

To remain faithful to Koza's original experiment, we used the same parameters where possible, as shown in Table II: a population of 500 individuals, crossover probability of 0.9, and no mutation. We used binary tournament selection instead of fitness-proportionate selection.

We chose bytecode segments randomly using a uniform probability distribution for segment sizes, with up to 1 000 retries (a limit reached in extremely rare cases, the average number of retries typically ranging between 16 and 24), as discussed in Section II-A.

An ideal individual was found in nearly every run. Typical evolutionary results are shown in Figs. 8 and 9.

Can FINCH be applied to a more complex case of symbolic regression? To test this we considered the recent work by Tuan-Hao *et al.* [31], where polynomials of the form $\sum_{i=1}^{n} x^i$, up to $n = 9$, were evolved using *incremental evaluation*. Tuan-Hao *et al.* introduced the DEVTAG evolutionary algorithm, which employs a multi-stage comparison between individuals to compute fitness. This fitness evaluation method is based on rewarding individuals that compute partial solutions of the target polynomial, i.e., polynomials $\sum_{i=1}^{n} x^i$, where $n < 9$ ($n = 9$ being the ultimate target polynomial).

To ascertain whether FINCH can tackle such an example of complex symbolic regression, we adapted our above evolutionary regression setup by introducing a fitness function in the spirit of Tuan-Hao *et al.* [31], based on the highest degree n computed by an evolving individual.

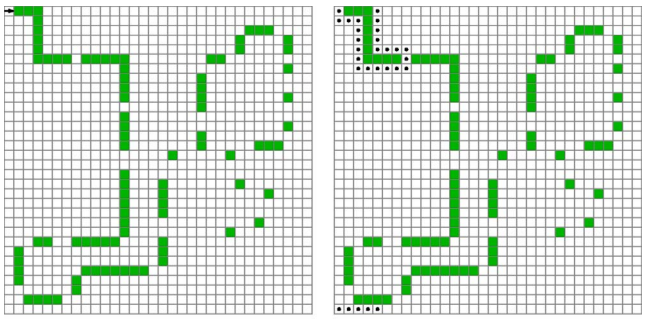
TABLE III
COMPLEX SYMBOLIC REGRESSION: PARAMETERS

Parameter	Tuan-Hao *et al.* [31]	FINCH
Objective	Symbolic regression: $x^9 + x^8 + \cdots + x^2 + x$	
Fitness	Sum of errors on 20 random samples in $[-1, 1]$, multi-stage incremental evaluation of polynomials $\sum_{i=1}^{n} x^i$ in DEVTAG	Degree n of polynomial $\sum_{i=1}^{n} x^i$ for which errors on all 20 random samples in $[-1, 1]$ are $< 10^{-8}$ + inverse of method size
Success predicate	All errors are less than 0.01	$n = 9$
Terminals	X, 1.0	**Number** num (an input)
Functions	+, -, *, /, SIN, COS, LOG, EXP (/ and LOG may return *Inf* and *NaN*)	Built-in arithmetic and **Math** functions present in the seed individual (Fig. 7)
Population	250 individuals	500 individuals
Generations	*Unspecified* (MAXGEN)	51, or less if ideal individual was found
Probabilities	$p_{\text{cross}} = 0.9$, $p_{\text{mut}} = 0.1$	$p_{\text{cross}} = 0.9$, $p_{\text{mut}} = 0$
Selection	Tournament of size 3	Tournament of size 7
Elitism	*Not used*	
Growth limit	Tree depth of 30	Maximal growth factor 5.0
Initial population	Random individuals with initial size 2–1000	Copies of seed program given in Fig. 7
Crossover location	Sub-tree crossover and sub-tree mutations	Gaussian distribution over segment sizes with 3σ = method size

```
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);
}
```

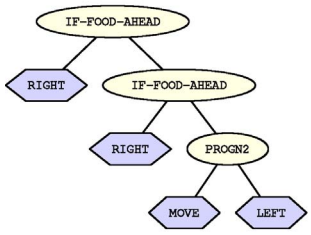
Fig. 10. Decompiled contents of method **simpleRegression** that evolved after 19 generations in the complex symbolic regression experiment. The evolutionary result proceeds as $x + (x \cdot (x \cdot (x + (x + (((x \cdot x + x) \cdot x + x) \cdot x) \cdot x) + x) + x) \cdot x$, where x is the method's input, which is computationally equivalent to $x^9 + \cdots + x^2 + x$. Observe the lack of unnecessary code due to parsimony pressure during evolution. Note that the regression problem tackled herein is not actually "simple"—we just used the same initial method as in the simple symbolic regression experiment.

We ran FINCH with clones of the same worst-case **simpleRegression** method used previously [Fig. 7] serving as the initial population. The evolutionary parameters are shown in Table III: a population of 500 individuals, crossover probability of 0.9, and no mutation. We used tournament selection with tournament size 7. Fitness was defined as the degree n computed by **simpleRegression** (or zero if no such n exists) plus the inverse of the evolved method size (the latter is a minor component we added herein to provide lexicographic parsimony pressure [17] for preferring smaller methods). The degree n is computed as follows: 20 random input samples in $[-1, 1]$ are generated, and the individual (method) computes all 20 outputs; if all 20 are within a 10^{-8} distance of a degree-n polynomial's outputs over these sample



(a) (b)

Fig. 11. Santa Fe food trail for the artificial ant problem, and the corresponding path taken by the randomly-generated *Avoider* individual in the experiment by Koza [11]. Note that the grid is toroidal. (a) Santa Fe trail. The ant starts at the upper-left corner facing right, its objective being to consume 89 food pellets. (b) Path taken by the *Avoider* individual, shown in Fig. 12, around the food pellets that are marked by colored cells.



(a)

```
void step() {
  if (foodAhead())
    right();
  else if (foodAhead())
    right();
  else {
    move(); left();
  }
}
```

(b)

Fig. 12. *Avoider* individual in the original experiment of Koza [11, ch. 7.2] is given by the S-expression `(IF-FOOD-AHEAD (RIGHT) (IF-FOOD-AHEAD (RIGHT) (PROGN2 (MOVE) (LEFT))))`. (a) Original Lisp individual. (b) Translation into Java.

points, then n is the highest degree, otherwise this fitness component is zero.

Bytecode segments were chosen randomly before checking them for crossover compatibility, with preference for smaller segments: we used $|\mathcal{N}(0, n/3)|$ as the distribution for segment sizes, where n is the number of instructions in the **`simpleRegression`** method of a given individual, and $\mathcal{N}(\mu, \sigma)$ is the Gaussian distribution specified by given mean and standard deviation. Finally, a maximal growth factor of 5.0 was specified, to limit the evolved method to a multiple of the original worst-case **`simpleRegression`** method (i.e., a limit of five times the number of bytecode instructions in the initial method).

An ideal individual was found in every run. A typical evolved method is shown (without the wrapper class) in Fig. 10. As a side issue, we also tested whether straightforward, non-incremental fitness evaluation can be used—a test which proved successful. We were able to evolve degree-9 polynomials directly, using a simple initial-population individual consisting only of instructions computing $(x + x) \cdot x$.

B. Artificial Ant

The artificial ant problem is a popular learning problem, where the objective is for an artificial ant to navigate along an irregular trail that contains 89 food pellets (the trail is known as the Santa Fe trail). Here we consider Koza's well-known experiment [11], where Lisp trees with simple terminal and function sets were evolved. The terminal set contained a `MOVE` operation that moves the ant in the direction it currently faces (possibly consuming the food pellet at the new location), and `LEFT` and `RIGHT` operations that rotate the ant by 90°. The function set consisted of `PROGN2` and `PROGN3`, for 2- and 3-sequence operations, respectively, and the `IF-FOOD-AHEAD` test that checks the cell directly ahead of the ant and executes its *then* and *else* branches according to whether the cell contains a food pellet or not. The Santa Fe trail is shown in Fig. 11(a), where the ant starts at the upper-left corner and faces right.

TABLE IV
ARTIFICIAL ANT: PARAMETERS

Parameter	Koza [11, ch. 7.2]	FINCH
Objective	Single step function for artificial ant that moves and eats food pellets on Santa Fe trail [Fig. 11(a)]	
Fitness	Food pellets consumed up to limit of 400 moves (probably any move is counted)	Food pellets consumed up to limit of 100 non-eating moves + inverse of **`step`** method size
Success predicate	The ant consumed all 89 food pellets	
Terminals	`LEFT`, `RIGHT`, `MOVE`	N/A
Functions	`IF-FOOD-AHEAD`, `PROGN2` (sequence of 2), `PROGN3` (sequence of 3)	Built-in control flow and the functions present in the seed individual [Fig. 12(b)]
Population	500 individuals	
Generations	51, or less if ideal individual was found	
Probabilities	$p_{\text{cross}} = 0.9$, $p_{\text{mut}} = 0$	
Selection	Fitness-proportionate	Tournament of size 7
Elitism	*Not used*	Five individuals
Growth limit	Tree depth of 17	Maximal growth factor 4.0
Initial population	Ramped half-and-half with maximal depth 6	Copies of seed program given in Fig. 12(b)
Crossover location	Internal nodes with $p_{\text{int}} = 0.9$, otherwise a terminal	Gaussian distribution over segment sizes with 3σ = method size

Koza reported that in one experiment the initial population contained a peculiar randomly generated *Avoider* individual that actively *eschewed* food pellets, as shown in Fig. 11(b). We chose this zero-fitness individual for our initial population, implementing Avoider as a straightforward and unconstrained Java function called **`step`**, as shown in Fig. 12 (along with the original Lisp-tree representation). The complete artificial ant implementation is listed in the Appendix.

During implementation of the artificial ant problem in FINCH, we were faced with some design choices that were not evident in the original experiment's description. One of them is a limit on the number of operations, in order to prevent evolution of randomly moving ants that cover the grid without using any logic. Koza reported using a limit of 400 operations, where each `RIGHT`, `LEFT`, and `MOVE` counts as an operation. However, an optimal ant would still take 545 operations to consume all 89 food pellets. Therefore, we opted for a limit of 100 *non-eating* moves instead, the necessary minimum being 55.

We ran FINCH with clones of the Java implementation of *Avoider* [Fig. 7] serving as the initial population. Again, we used the same parameters of Koza where possible, as shown in

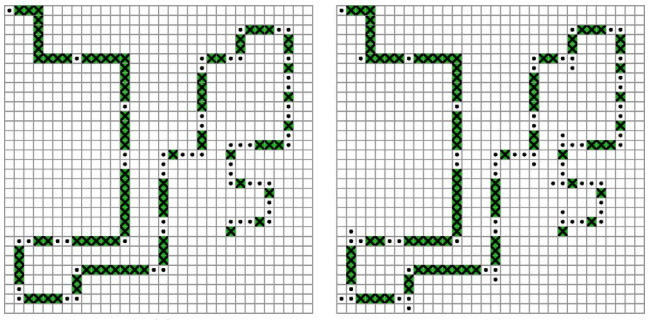
```
void step() {
  if (foodAhead()) {
    move();
    right();
  }
  else {
    right();
    right();
    if (foodAhead())
      left();
    else {
      right();
      move();
      left();
    }
    left();
    left();
  }
}
```

(a)

```
void step() {
  if (foodAhead()) {
    move(); move();
    left(); right();
    right(); left();
    right();
  } else {
    right(); right();
    if (foodAhead()) {
      move(); right();
      right(); move();
      move(); right();
    } else {
      right(); move();
      left();
    }
    left(); left();
  }
}
```

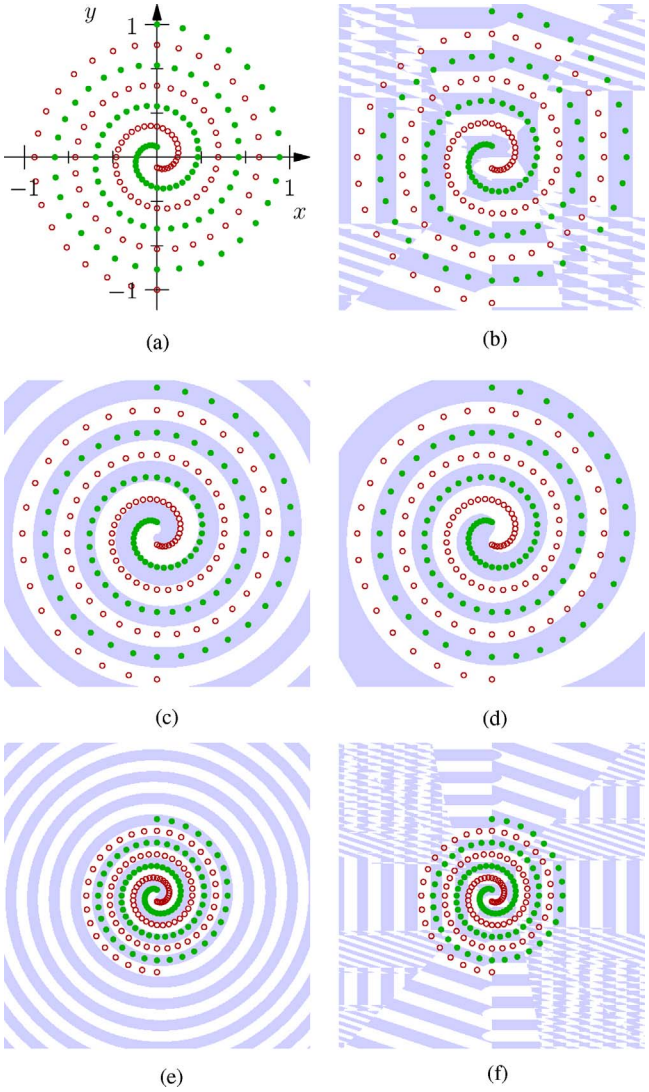
(b)

Fig. 13. **`Step`** methods of two solutions to the artificial ant problem that were evolved by FINCH. The corresponding ant trails are shown in Fig. 14. (a) Optimal individual that appeared in generation 45. It makes no unnecessary moves, as can be seen in the corresponding ant trail in Fig. 14(a). (b) Solution that appeared in generation 31. It successfully consumes all the food pellets, but makes some unnecessary moves, as shown in Fig. 14(b).



(a) (b)

Fig. 14. Ant trails that result from executing the artificial ant programs that contain the evolved **`step`** methods shown in Fig. 13. (a) Trail of the optimal individual shown in Fig. 13(a). (b) Trail of the non-optimal (though all-consuming) individual shown in Fig. 13(b).



(a) (b) (c) (d) (e) (f)

Fig. 15. Intertwined spirals dataset (a) and the visualizations of two evolved (perfect) solutions (c), (d) contrasted with the result produced by Koza (b). Note the smoothness of FINCH's solutions as compared with the jaggedness of Koza's. This is even more striking when "zooming out," as shown in (e) and (f). (a) Intertwined spirals, as described by Koza [11]. The two spirals, containing 97 points each, encircle the axes' origin three times. The first spiral (filled circles) belongs to class +1, and the second spiral (empty circles) belongs to class −1. The farthest point on each spiral is at unit distance from the origin. (b) Visualization of the solution evolved by Koza [11] (shown in Fig. 18), re-created by running this individual (taking into account the different scale used by Koza—the farthest points are at distance 6.5 from the origin). Note the jaggedness of the solution, due to 11 conditional nodes in the genotype. (c) Visualization of the solution in Fig. 17, found by FINCH. Points for which the evolved program returns true are indicated by a dark background. After manual simplification of the program, we see that it uses the sign of $\sin\left(\frac{9}{4}\pi^2\sqrt{x^2+y^2} - \tan^{-1}\frac{y}{x}\right)$ as the class predictor of (x, y). (d) Visualization of another snail-like solution to the intertwined spirals problem, evolved by FINCH. Note the phenotypic smoothness of the result, which is also far terser than the bloated individual that generated (b), all of which points to our method's producing a more general solution. (e) Solution (c) scaled to span the $[-2, 2]$ interval on both axes. (f) Koza's solution (b), scaled similarly to (e).

Table IV: a population of 500 individuals, crossover probability of 0.9, and no mutation. We used tournament selection with tournament size 7 instead of fitness-proportionate selection, and elitism of five individuals. Fitness was defined as the number of food pellets consumed within the limit of 100 non-eating moves, plus the inverse of the evolved method size (the former component is similar to the original experiment, the latter is a minor parsimony pressure component as in the complex symbolic regression problem). Bytecode segments were chosen randomly before checking them for crossover compatibility, with preference for smaller segments, as described previously. Finally, a maximal growth factor of 4.0 was specified, to limit the evolved method to a multiple of the original *Avoider* **`step`** method.

Fig. 13 shows two typical, maximal-fitness solutions to the Santa Fe artificial ant problem, as evolved by FINCH. The corresponding ant trails are shown in Fig. 14. Table I shows the success rate (i.e., percentage of runs producing all-consuming individuals) for runs using the settings in Table IV, and also the yield after increasing the population size and removing parsimony pressure (the inverse **`step`** method size component of the fitness function).

TABLE V
INTERTWINED SPIRALS: PARAMETERS

Parameter	Koza [11, ch. 7.3]	FINCH
Objective	Two-class classification of intertwined spirals [Fig. 15(a)]	
Fitness	The number of points that are correctly classified	
Success predicate	All 194 points are correctly classified	
Terminals	X, Y, ℜ (ERC in [−1, 1])	double x, y (inputs)
Functions	+, −, *, % (protected division), IFLTE (four-argument *if*), SIN, COS	Built-in control flow and the functions present in the seed individual (Fig. 16)
Population	10 000 individuals	2000 individuals
Generations	51, or less if ideal individual was found	251, or less if ideal individual was found
Probabilities	$p_{\text{cross}} = 0.9$, $p_{\text{mut}} = 0$	$p_{\text{cross}} = 0.8$, $p_{\text{mut}} = 0.1$, Gaussian constants mutation with $3\sigma = 1$
Selection	Fitness-proportionate	Tournament of size 7
Elitism	Not used	
Growth limit	Tree depth of 17	Maximal growth factor 4.0
Initial population	Ramped half-and-half with maximal depth 6	Copies of seed program given in Fig. 16
Crossover location	Internal nodes with $p_{\text{int}} = 0.9$, otherwise a terminal	Gaussian distribution over segment sizes with 3σ = method size

```
boolean isFirst(double x, double y) {
  double a = Math.hypot(x, y);
  double b = Math.atan2(y, x);
  double c = -a + b * 2;
  double d = -b * Math.sin(c) / a;
  double e = c - Math.cos(d) - 1.2345;
  boolean res = e >= 0;
  return res;
}
```

Fig. 16. Method of the intertwined spirals individual of the initial population. A return value of `true` indicates class +1. This method serves as a repository of components to be used by evolution: floating-point arithmetic operators, trigonometric functions, and functions usable in polar-rectangular coordinates conversion—all arbitrarily organized. Note that the inequality operator translates to a conditional construct in the bytecode.

```
boolean isFirst(double x, double y) {
  double a, b, c, e;
  a = Math.hypot(x, y);  e = y;
  c = Math.atan2(y, b = x) +
    -(b = Math.atan2(a, -a))
    * (c = a + a) * (b + (c = b));
  e = -b * Math.sin(c);
  if (e < -0.0056126487018762772) {
    b = Math.atan2(a, -a);
    b = Math.atan2(a * c + b, x);  b = x;
    return false;
  }
  else
    return true;
}
```

Fig. 17. Decompiled contents of method **`isFirst`** that evolved after 86 generations from the Java program in Fig. 16. The variable names have been restored—a bit of manual tinkering with an otherwise automatic technique. This method returns `true` for all points of class +1, and `false` for all points of class −1. This is an "elegant" generalizable solution, unlike the one reported by Koza [11], where the evolved individual contains 11 conditionals and 22 constants. Note that the only constant here is an approximation to 0, and $\tan^{-1}\frac{a}{-a} = \frac{3}{4}\pi$, since a is a positive magnitude value.

C. Intertwined Spirals

In the intertwined spirals problem the task is to correctly classify 194 points on two spirals, as shown in Fig. 15(a). The points on the first spiral are given in polar coordinates by

$$r_n = \frac{8+n}{104} \qquad \alpha_n = \frac{8+n}{16} \cdot \pi$$

for $0 \leqslant n \leqslant 96$, and the Cartesian coordinates are

$$x_n^+ = r_n \cos\alpha_n \qquad x_n^- = -x_n^+$$
$$y_n^+ = r_n \sin\alpha_n \qquad y_n^- = -y_n^+$$

where (x_n^+, y_n^+) are points on the first spiral, and (x_n^-, y_n^-) lie on the second spiral [5].

A classic machine-learning case study, the intertwined spirals problem was treated by Koza [11] using the parameters shown in Table V, with his best-of-run individual including 11 conditionals and 22 constants (shown in Fig. 18). Whereas Koza used a slew of ERCs (ephemeral random constants) in the initial population, FINCH's initial population is seeded with clones of a single program (shown in Fig. 16), containing very few constants. We therefore implemented mutation of constants, as follows. Before an individual is added to the new population, each floating-point constant in the bytecode is modified with probability p_{mut} by adding an $\mathcal{N}(0, 1/3)$ Gaussian-distributed random value.

We seeded the population with an individual containing the method shown in Fig. 16. In addition to the usual arithmetic functions, we added some trigonometric functions that seemed to be useful—since a "nice" solution to the intertwined spirals problem is likely to include a manipulation of polar coordinates of the points on the two spirals. This assumption proved to be right: Fig. 17 shows a typical evolved result, visualized in Fig. 15(c). Unlike the jagged pattern of the extremely precision-sensitive solution evolved by Koza (with slight changes in constants notably degrading performance), we observe a smooth curvature founded on an elegant mathematical equation. This is likely a result of incremental evolution that starts with clones of a single individual, and lacks an initial "wild" phase of crossovers of randomly-generated individuals with different ERCs. In addition, Koza used a much higher growth limit (see Table V), with his best-of-run comprising 91 internal nodes. This individual, shown in Fig. 18, can be observed to be far "quirkier" than solutions evolved by FINCH (e.g., Fig. 17). Fig. 15(d) shows another visualization of an evolved solution.

As done by Koza [12], we also retested our ten best-of run individuals on sample points chosen twice as dense (i.e., 386 points), and ten times more dense (1922 points). For seven individuals, 100% of the points in both denser versions of the intertwined spirals problem were correctly classified; for the remaining three individuals, 99% of the points were correctly classified on average, for both denser versions. Koza reported 96% and 94% correct classification rates for doubly dense and tenfold dense versions, respectively, for the single best-of-run individual. Hence, our solutions exhibit generality as well.

```
(sin (iflte (iflte (+ Y Y) (+ X Y) (- X Y) (+ Y
Y)) (* X X) (sin (iflte (% Y Y) (% (sin (sin (% Y
0.30400002))) X) (% Y 0.30400002) (iflte (iflte
(% (sin (% (% Y (+ X Y)) 0.30400002)) (+ X Y))
(% X -0.10399997) (- X Y) (* (+ -0.12499994
-0.15999997) (- X Y))) 0.30400002 (sin (sin (iflte
(% (sin (% (% Y 0.30400002) 0.30400002)) (+
X Y)) (% (sin Y) Y) (sin (sin (sin (% (sin X) (+
-0.12499994 -0.15999997))))) (% (+ (+ X Y) (+
Y Y)) 0.30400002)))) (+ (+ X Y) (+ Y Y)))))
(sin (iflte (iflte Y (+ X Y) (- X Y) (+ Y Y)) (* X X)
(sin (iflte (% Y Y) (% (sin (sin (% Y 0.30400002)))
X) (% Y 0.30400002) (sin (sin (iflte (iflte (sin (%
(sin X) (+ -0.12499994 -0.15999997))) (% X
-0.10399997) (- X Y) (+ X Y)) (sin (% (sin X)
(+ -0.12499994 -0.15999997))) (sin (sin (% (sin X)
(+ -0.12499994 -0.15999997)))) (+ (+ X Y) (+ Y
Y)))))))) (% Y 0.30400002)))))
```

Fig. 18. Best-of-run S-expression evolved by Koza [11] at generation 36, visualized in Fig. 15(b), containing 88 terminals (where 22 are constants), and 91 functions (where 11 are conditional operators). This result is extremely sensitive to the exact values of the constants, the intertwined spirals dataset, and the floating-point precision of the S-expression evaluator.

TABLE VI
ARRAY SUM: PARAMETERS

Parameter	Withall *et al.* [32]	FINCH
Objective	Summation of numbers in an input array	
Fitness	Negative total of differences from array sums on ten predefined test inputs	As in [32], + inverse of **`sumlist`** method size
Success predicate	The sums calculated for ten test inputs and ten verification inputs are correct	
Variables	`Sum` (read-write), `size`, `tmp`, `list[tmp]` (read-only, list index is taken modulo list size)	`Sum`, `size`, `tmp` (read-write), `list` (read-only array accesses)
Statements	Variable assignment, four arithmetical operations, **if** comparing two variables, **for** loop over all list elements	
Population	Seven individuals	500 individuals
Generations	50 000, or less if ideal individual was found	51, or less if ideal individual was found
Probabilities	$p_{\text{cross}} = 1$, $p_{\text{mut}} = 0.1$	$p_{\text{cross}} = 0.8$, $p_{\text{mut}} = 0$
Selection	Fitness-proportionate	Tournament of size 7
Elitism	One individual	
Growth limit	*Fixed length*	Maximal growth factor 2.0
Time limit	1000 loop iterations	5000 backward branches
Initial population	Randomly-generated integer vector genomes	Copies of seed program given in Fig. 19
Crossover location	Uniform crossover between fixed-length genomes	Gaussian distribution over segment sizes with 3σ = method size

D. Array Sum

So far all our programs have consisted of primitive Java functions along with a forward jump in the form of a conditional statement. Moving into Turing-complete territory, we ask in this subsection whether FINCH can handle two of the most important constructs in programming: loops and recursion. Toward this end we look into the problem of computing the sum of values of an integer array.

Withall *et al.* [32] recently considered a set of problems described as "more traditional programming problems than traditional GP problems," for the purpose of evaluating an improved representation for GP. This representation, which resembles the one used in grammatical evolution [24], maintains a genome comprising blocks of integer values, which are mapped to predefined statements and variable references through a given genotype-to-phenotype translation. The statements typically differentiate between read-only and read-write (assignable) variables—in contrast to FINCH, where a variable that is not assigned to in the seed individual is automatically "write-protected" during evolution.

```
int sumlist(int[] list) {
  int sum  = 0;
  int size = list.length;
  for (int tmp = 0; tmp < list.length; tmp++) {
    sum = sum - tmp * (list[tmp] / size);
    if (sum > size || tmp == list.length + sum)
      sum = tmp - list[size/2];
  }
  return sum;
}
```

Fig. 19. Evolving method of the seed individual for the array sum problem. Note that loop variable `tmp` is assignable, and thus the **for** loop can "deteriorate" during evolution. The array indexes are not taken modulo list size like in [32]—an exception is automatically thrown by the JVM in case of an out-of-bounds index use.

```
int sumlist(int list[]) {
  int sum  = 0;
  int size = list.length;
  for (int tmp = 0; tmp < list.length; tmp++) {
    size = tmp;
    sum = sum - (0 - list[tmp]);
  }
  return sum;
}
```

Fig. 20. Decompiled ideal individual that appeared in generation 11, correctly summing up all the test and validation inputs. Variable names were manually restored for clarity.

Table VI shows the evolutionary setups used by Withall *et al.* [32] and by us for the array sum problem (called *sumlist* by Withall *et al.*). During evaluation, individuals are given ten predefined input lists of lengths 1–4, and if a program correctly computing all the ten sums is found, it is also validated on ten predefined verification inputs of lengths 1–9. Our initial population was seeded with copies of the blatantly unfit Java program in Fig. 19. This program includes a **for** statement, for use by evolution, and a loop body that is nowhere near the desired functionality—but merely serves to provide some basic evolutionary components. An important new criterion in Table VI—*time limit*—regards the CPU resources allocated to a program during its evaluation, a measure discussed in Section II-C.

FINCH encountered little difficulty in finding solutions to the array sum problem (see Table I). One evolved solution is shown in Fig. 20. FINCH's ability to handle this problem gracefully is all the more impressive when one considers the vastly greater search space in comparison to other systems. For instance, Withall *et al.* defined the **for** statement as an elemental (unbreakable) "chunk" in the genome, specified only by the read-only iteration variable. In addition, array indexes

```
int sumlist(List<Integer> list) {
  int sum  = 0;
  int size = list.size();
  for (int tmp: list) {
    sum = sum - tmp * (tmp / size);
    if (sum > size || tmp == list.size() + sum)
      sum = tmp;
  }
  return sum;
}
```

Fig. 21. Evolving method of the seed individual for the **List** version of the array sum problem. Note that although the new Java 5.0 container iteration syntax is simple to use, it is translated to sophisticated iterators machinery [9], as is evident in the best-of-run result in Fig. 22.

```
int sumlist(List list) {
  int sum = 0;
  int size = list.size();
  for (Iterator iterator = list.iterator();
                        iterator.hasNext(); ) {
    int tmp = ((Integer) iterator.next())
                        .intValue();
    tmp = tmp + sum;
    if (tmp == list.size() + sum)
      sum = tmp;
    sum = tmp;
  }
  return sum;
}
```

Fig. 22. Decompiled ideal individual that appeared in generation 12. Variable names were manually restored for the purpose of clarity, but Java 5.0 syntax features (generic classes, unboxing, and enhanced **for**) were not restored.

```
int sumlistrec(List<Integer> list) {
  int sum = 0;
  if (list.isEmpty())
    sum *= sumlistrec(list);
  else
    sum += list.get(0)/2 + sumlistrec(
           list.subList(1, list.size()));
  return sum;
}
```

Fig. 23. Evolving method of the seed individual for the recursive **List** version of the array sum problem. The call to the **get** method returns the first list element, and **subList** returns the remainder of the list (the two methods are known as *car* and *cdr* in Lisp). Some of the obstacles evolution must overcome herein are the invalid stop condition that causes infinite recursion, and a superfluous operation on the first list element.

were taken modulo array size. FINCH, however, has no such abstract model of the bytecode (nor does it need one). A **for** loop is compiled to a set of conditional branches and variables comparisons, and array access via an out-of-bound index raises an exception.

Of course, FINCH is not limited to dealing with integer arrays—it can easily handle different list abstractions, such as those that use the types defined in the powerful Java standard library. Fig. 21 shows the seed method used in a slightly modified array sum class, where the **List** abstraction is used for a list of numbers. Solutions evolve just as readily as in the integer array approach—see Fig. 22 for one such individual.

Having demonstrated FINCH's ability to handle loops—we now turn to recursion. Fig. 23 shows the seed individual

```
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;
}
```

Fig. 24. Decompiled ideal individual for the recursive **List** array sum problem version, which appeared in generation 2. Java 5.0 syntax features were not restored.

Input : a minimax tree node node, search depth limit d, α-β pruning parameters, player color $c \in \{1, -1\}$

if node *is a terminal node* $\vee$ $d = 0$ **then**
  return $c \cdot$ UTILITY(node)
else
  foreach succ $\in$ *successors of* node **do**
    $\alpha \leftarrow \max\big(\alpha, -\text{NEGAMAX}(\text{succ}, d-1, -\beta, -\alpha, -c)\big)$
    if $\alpha \geqslant \beta$ **then**
      return α
  return α

Fig. 25. NEGAMAX (node, d, α, β, c): an α-β-pruning variant of the classic minimax algorithm for zero-sum, two-player games, as formulated at the Wikipedia site, wherein programmers might find it. The initial call for the root minimax tree node is d, $-\infty$, ∞, 1. The function UTILITY returns a heuristic node value for the player with color $c = 1$.

used to evolve recursive solutions to the array sum problem. Note that this method enters a state of infinite recursion upon reaching the end of the list, a situation which in no way hinders FINCH, due to its use of the instruction limit-handling mechanism described in Section II-C. Solutions evolve readily for the recursive case as well—see Fig. 24 for an example.

E. Tic-Tac-Toe

Having shown that FINCH can evolve programmatic solutions to hard problems, along the way demonstrating the system's ability to handle many complex features of the Java language, we now take a different stance, that of *program improvement*. Specifically, we wish to generate an optimal program to play the game of tic-tac-toe, based on the negamax algorithm.[3]

Fig. 25 shows the negamax algorithm, a variant of the classic minimax algorithm used to traverse game trees, thus serving as the heart of many programs for two-player games—such as tic-tac-toe. Whereas in the previous examples we seeded FINCH with rather "deplorable" seeds, programs whose main purpose was to inject the basic evolutionary ingredients, herein our seed is a highly functional—yet *imperfect* program.

We first implemented the negamax algorithm, creating an optimal tic-tac-toe strategy, i.e., one that never loses. We then seeded FINCH with four imperfect versions thereof,

[3]Tic-tac-toe is a simple noughts and crosses game, played on a 3×3 grid, where the two players **X** (who plays first) and **O** strive to place three marks in a horizontal, vertical, or diagonal row.

```
1 int negamaxAB(TicTacToeBoard board,
2       int alpha, int beta, boolean save) {
3   Position[] free = getFreeCells(board);
4   // utility is derived from the number of free cells left
5   if (board.getWinner() != null)
6     alpha = utility(board, free);
7   else if (free.length == 0)
8     alpha = 0;
9   else for (Position move: free) {
10    TicTacToeBoard copy = board.clone();
11    copy.play(move.row(), move.col(),
12                      copy.getTurn());
13    int utility = -negamaxAB(copy,
14                      -beta, -alpha, false);
15    if (utility > alpha) {
16      alpha = utility;
17      if (save)
18        // save the move into a class instance field
19        chosenMove = move;
20      if (alpha >= beta)
21        break;
22    }
23  }
24  return alpha;
25 }
```

Fig. 26. FINCH setup for improving imperfect tic-tac-toe strategies. Shown above is the key Java method in a perfect implementation of the negamax algorithm (Fig. 25) that a seasoned programmer might write—*if* she got everything right. However, we consider four possible single-error lapses, or *imperfections*, as it were, which the programmer could easily have introduced into the Java code. Here, the **`utility`** method computes the deterministic board value for the player whose turn it is (i.e., the *color* variable of Fig. 25 is unnecessary), assigning higher values to boards with more free cells. The **`negamaxAB`** method represents an optimal player that wins (or draws) in as few turns as possible.

TABLE VII

TIC-TAC-TOE: PARAMETERS

Parameter	Angeline and Pollack [1]	FINCH
Objective	Learn to play tic-tac-toe	
Fitness	Number of rounds won in single-elimination tournament	
Success predicate	*Not defined*	Same performance against RAND and BEST as an optimal player
Terminals	$\texttt{pos}_{00}, \ldots, \texttt{pos}_{22}$ (board positions)	Primitive and object parameters, and local variables in Fig. 26, including the **Position** enum
Functions	`And`, `or`, `if` (three-argument if), `open`, `mine`, `yours` (position predicates), `play-at` (position action)	All the control flow and methods used in Fig. 26, including the **`play`** tic-tac-toe board instance method
Population	256 individuals	2048 individuals
Generations	150	16
Probabilities	$p_{\text{cross}} = 0.9$, $p_{\text{compr}} = 0.1$	$p_{\text{cross}} = 0.8$, $p_{\text{mut}} = 0$
Selection	Fitness-proportionate, with linear scaling to $0 \sim 2$	Tournament of size 7
Elitism	*Not used*	Seven individuals
Growth limit	Tree depth of 15	Maximal growth factor 2.0
Time limit	*Not used*	500 000 back-branches
Initial population	*Grow* with maximal depth 4	Copies of seed program given in Fig. 26
Crossover location	Internal nodes with $p_{\text{int}} = 0.9$, otherwise a terminal	Gaussian distribution over segment sizes, $\sigma = 3.0$

TABLE VIII

TIC-TAC-TOE

Line	Single-Error Imperfection	RAND W	RAND D	RAND L	BEST D	BEST L
	RAND	44	12	44	13	87
	BEST	87	13	0	100	0
8	Put `save= false` instead of `alpha = 0`	87	2	11	50	50
13	Remove unary "–" preceding the recursive call to **`negamaxAB`** method	25	35	40	16	84
14	Pass `save` instead of `false` to the recursive call	72	10	18	32	68
20	Swap `alpha` and `beta` in the conditional test	45	11	44	13	87

Four different single-error imperfections and their effect on the resulting player's performance over a 2000-game match. Line numbers refer to the perfect code of Fig. 26. Performance is shown as percentage of wins, draws, and losses vs. two players: RAND and BEST. (Note that BEST never loses.)

demonstrating four distinct, plausible, single-error slips that a good human programmer might make. We asked whether FINCH could improve our imperfect programs, namely, evolve the perfect, optimally performing negamax algorithm, given each one of the four imperfect versions. Our setup is illustrated in Fig. 26.

Given a good-but-not-perfect tic-tac-toe program, i.e., an imperfect version of Fig. 26, we set FINCH loose. In their work on evolving tic-tac-toe players, Angeline and Pollack [1] computed fitness by performing a single-elimination tournament among individuals in the evolving population, demonstrating this method's superiority over using "expert" players, in terms of the evolved players' ability to compete against the optimal player. Table VII details the analogous evolutionary setup we used. Note that we used a fixed standard deviation for segment sizes in order to focus the search on small modifications to the evolving programs.

In single-elimination tournament, as applied to our setup, 2^k players are arbitrarily partitioned into 2^{k-1} pairs. Each pair competes for one round and the winner moves on to the next tournament level—which has 2^{k-1} players. A single round consists of two games, each player thus given the chance to be **X**, i.e., to make the first move. The round winner is determined according to $\text{sgn}(\frac{1}{m_1} - \frac{1}{m_2})$, where m_i is the number of moves player i made to win the game it played as **X** (m_i is negative if player **O** won, or ∞ in case of a draw).[4] The fitness value of an individual is simply the number of rounds won, and is in the range $\{0, \ldots, k\}$. Ties are broken randomly. This approach gives preference to players that take less moves to win.

Table VIII lists four distinct imperfections an experienced programmer might have realistically created while implementing the non-trivial **`negamaxAB`** method, and the impact of these imperfections on the resulting tic-tac-toe player's

[4]The absolute value of m_i is actually the number of moves plus 1, to accommodate the possibility of a win in 0 moves, as is the case when **X** fails to make the first move, thus forfeiting (and losing) the game.

```
1  int negamaxAB(TicTacToeBoard board,
2        int alpha, int beta, boolean save) {
3    Position free[] = getFreeCells(board);
4    if (board.getWinner() != null)
5      alpha = utility(board, free);
6    else if (free.length == 0)
7      alpha = 0;
8    else {
9      Position free1[];
10     int l = (free1 = free).length;
11     for (int k = 0; k < l; k++) {
12       Position pos = free[k];
13       TicTacToeBoard copy = board.clone();
14       copy.play(pos.row(), pos.col(),
15                       copy.getTurn());
16       int utility = -negamaxAB(copy,
17                      -beta, -alpha, false);
18       if (utility > alpha) {
19         alpha = utility;
20         if (save)
21           chosenMove = pos;
22         if (-beta >= -alpha)
23           break;
24       }
25       pos = free1[k];
26     }
27   }
28   return alpha;
29 }
```

Fig. 27. Decompiled Java method in a solution evolved from the `alpha-beta` swap imperfect seed in Table VIII. Compare line 22 above with line 20 of Fig. 26. Variable names were manually restored according to Fig. 26 (Java 5.0 for-each loop was not restored).

performance against two of the standard players defined by Angeline and Pollack [1]: RAND and BEST. The former plays randomly and the latter is an optimal player, based on the correct negamax implementation shown in Fig. 26 (so in our case it also minimizes the number of moves to win or draw). We see that although each of the four single-error flaws is minute and subtle at the source-code level (and therefore likely to be made), the imperfections have a varying (detrimental) impact on the player's performance.

Our experiments, summarized in Table I, show that FINCH easily unravels these unfortunate imperfections in the completely unrestricted, real-world Java code. The evolved bytecode plays at the level of the BEST optimal player, never losing to it. Fig. 27 shows one interesting, subtle example of a solution evolved from the `alpha-beta` swap imperfect seed (last case of Table VIII). FINCH discovered this solution by cleverly reusing unrelated code through the compatible crossover operator: stack pushes of the `beta` and `alpha` parameters for the **if_icmplt** comparison instruction were replaced by stack pushes of `-beta` and `-alpha` from the parameters passing section of the recursive **`negamaxAB`** method call.

IV. Conclusion

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 employed *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.

It is important to keep in mind the scope and limitations of FINCH. No software development method is a "silver bullet," and FINCH is no exception to this rule. Evolving as-is software still requires a suitably defined fitness function, and it is quite plausible that manual improvement might achieve better results in some cases. That being said, we believe that automatic software evolution will eventually become an integral part of the software engineer's toolbox.

A recent study commissioned by the U.S. 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" [23, 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 (with the proof-of-concept described herein focusing on relatively small software systems). Differences aside, both our work and this paper share the vision of true software evolution.

Is good crossover 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 good 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 evolutionarily significant components 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.

Several avenues of future research present themselves, including:

1) defining a process by which consistent bytecode segments can be *found* during compatibility checks, thus improving preservation of evolutionary components during evolution;
2) supporting class-level evolution, such as cross-method crossover and introduction of new methods;
3) development of mutation operators, currently lacking (except for the constant mutator of Section III-C);
4) applying FINCH to additional hard problems, along

the way garnering further support for our approach's efficacy;
5) directly handling high-level bytecode constructs such as try/catch clauses and monitor enter/exit pairs;
6) designing an integrated development environment (IDE) plugin to enable the use of FINCH in software projects by non-specialists;
7) applying FINCH to meta-evolution, in order to discover better evolutionary algorithms;
8) applying unrestricted bytecode evolution to the automatic improvement of existing applications, establishing the relevance of FINCH to the realm of extant software.

Ultimately, one might be able to relax and forget about the Java *programming language*, concentrating instead on the *beverage* to be enjoyed, as evolution blithely works to produce working programs.

APPENDIX
ARTIFICIAL ANT: AVOIDER

Below, we detail the implementation of the Santa Fe artificial ant problem in Java, after slight simplifications, such as removing assertions intended for debugging. *Avoider*, a zero-fitness, generation-0 individual from the experiment by Koza [11], was implemented as the **`step`** method. Note that this is a standard, unrestricted Java class, with static and instance fields, an inner class, and a virtual function override. The compiled bytecode was then provided as-is to FINCH for the purpose of evolution.

```
public class ArtificialAnt {
  // Exception for exceeding operations limit
  public static class OperationsLimit
        extends RuntimeException {
    public final int ops;
    public OperationsLimit(int ops) {
      super("Operations limit of " + ops
          + " reached");
      this.ops = ops;
    }
  }

  // Map loader, also provides ASCII representation
  private static final ArtificialAntMap antMap =
    new ArtificialAntMap(ArtificialAntMap.class
        .getResource("santafe.trl"));

  private final int maxOps;
  private int     opsCount;

  private final boolean[][] visitMap;
  private int     eaten;     // pellets counter
  private int     x,  y;     // col, row
  private int     dx, dy;    // { -1, 0, +1 }

  public ArtificialAnt(int maxOps) {
    this.maxOps = maxOps;
    opsCount    = 0;

    boolean[][] model = antMap.foodMap;
    visitMap = new boolean[model.length][];

    // Initialized to "false"
    for (int row = 0;  row < visitMap.length;
         ++row)  visitMap[row] =
      new boolean[model[row].length];

    eaten = 0;
    x  = 0;  y = 0;
    dx = 1; dy = 0;
    visit();
  }

  // Perform as many steps as possible
  public void go()
  { while (!ateAll()) step(); }

  // Avoider (Koza I, p.151)
  public void step() {
    if (foodAhead()) right();
    else if (foodAhead()) right();
    else { move(); left(); }
  }

  // Visits current cell
  private void visit() {
    if (! visitMap[y][x]) {
      visitMap[y][x] = true;
      // Don't count eating as a move
      if (antMap.foodMap[y][x])
        { ++eaten; --opsCount; }
    }
  }

  // Moves to next cell in current direction
  private void move() {
    x = (x + dx + antMap.width)
      % antMap.width;
    y = (y + dy + antMap.height)
      % antMap.height;
    visit();  operation();
  }

  // Turns counter-clockwise
  private void left() {
    if (dy == 0)  { dy = -dx; dx = 0; }
    else          { dx = dy;  dy = 0; }
  }

  // Turns clockwise
  private void right() {
    if (dy == 0)  { dy = dx;  dx = 0; }
    else          { dx = -dy; dy = 0; }
  }

  private void operation() {
    if (++opsCount >= maxOps)
      throw new OperationsLimit(opsCount);
  }

  // Checks whether a food pellet is at next cell
  private boolean foodAhead() {
    int xx = (x + dx + antMap.width)
          % antMap.width;
    int yy = (y + dy + antMap.height)
          % antMap.height;
    return antMap.foodMap[yy][xx]
      && !visitMap[yy][xx];
  }

  // Returns number of eaten food pellets
  public int getEatenCount()
  { return eaten; }
```

```
  // Returns true if all food pellets were eaten
  public boolean ateAll()
  { return eaten == antMap.totalFood; }

  @Override
  public String toString()
  { return antMap.toString(visitMap); }
}
```

REFERENCES

[1] P. J. Angeline and J. B. Pollack, "Competitive environments evolve better solutions for complex tasks," in *Proc. 5th ICGA*, Jul. 1993, pp. 264–270.

[2] A. Arcuri, "Automatic software generation and improvement through search based techniques," Ph.D. dissertation, School Comput. Sci., Univ. Birmingham, Birmingham, U.K., Dec. 2009 [Online]. Available: http://etheses.bham.ac.uk/400

[3] M. F. Brameier and W. Banzhaf, *Linear Genetic Programming* (Genetic and Evolutionary Computation). New York: Springer, Dec. 2006.

[4] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A code manipulation tool to implement adaptable systems (Un outil de manipulation de code pour la réalisation de systèmes adaptables)," in *Proc. ACM SIGOPS France Journées Composants 2002: Systèmes à Composants Adaptables et Extensibles (Adaptable and Extensible Component Systems) (Systèmes à Composants Adaptables et Extensibles)*, Oct. 2002, pp. 184–195 [Online]. Available: http://asm.objectweb.org/current/asm-eng.pdf

[5] Carnegie Mellon University. (1993, Feb.). *CMU Neural Network Benchmark Database* [Online]. Available: http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/cmu

[6] C. Darwin, *On the Origin of Species by Means of Natural Selection, or the Preservation of Favored Races in the Struggle for Life*. London, U.K.: John Murray, 1859.

[7] J. Engel, *Programming for the Java™ Virtual Machine*. Reading, MA: Addison-Wesley, Jul. 1999.

[8] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proc. 11th GECCO*, Jul. 2009, pp. 947–954.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java™ Language Specification* (The Java™ Series, 3rd ed.). Boston, MA: Addison-Wesley, May 2005 [Online]. Available: http://java.sun.com/docs/books/jls

[10] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, "Design of the Java HotSpot™ client compiler for Java 6," *ACM Trans. Architecture Code Optimization*, vol. 5, no. 7, pp. 7:1–7:32, May 2008.

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

[12] J. R. Koza, "A genetic approach to the truck backer upper problem and the inter-twined spiral problem," in *Proc. IJCNN*, vol. 4. Jul. 1992, pp. 310–318.

[13] W. B. Langdon and P. Nordin, "Seeding genetic programming populations," in *Proc. 3rd EuroGP*, LNCS 1802. Apr. 2000, pp. 304–315.

[14] W. B. Langdon and R. Poli, "The halting probability in von Neumann architectures," in *Proc. 9th EuroGP*, LNCS 3905. Apr. 2006, pp. 225–237.

[15] T. Lindholm and F. Yellin, *The Java™ Virtual Machine Specification* (The Java™ Series, 2nd ed.). Boston, MA: Addison-Wesley, Apr. 1999 [Online]. Available: http://java.sun.com/docs/books/jvms

[16] S. Luke and L. Panait. (2004, Mar.). *A Java-Based Evolutionary Computation Research System* [Online]. Available: http://cs.gmu.edu/∼ eclab/projects/ecj

[17] S. Luke and L. Panait, "Lexicographic parsimony pressure," in *Proc. 4th GECCO*, Jul. 2002, pp. 829–836.

[18] J. Miecznikowski and L. Hendren, "Decompiling Java bytecode: Problems, traps and pitfalls," in *Proc. Compiler Construction 11th Int. Conf. ETAPS*, LNCS 2304. Apr. 2002, pp. 111–127.

[19] J. Mizoguchi, H. Hemmi, and K. Shimohara, "Production genetic algorithms for automated hardware design through an evolutionary process," in *Proc. 1st ICEC*, vol. 2. Jun. 1994, pp. 661–664.

[20] D. J. Montana, "Strongly typed genetic programming," *Evol. Comput.*, vol. 3, no. 2, pp. 199–230, 1995.

[21] Y. Nakamura, K. Oguri, and A. Nagoya, "Synthesis from pure behavioral descriptions," in *High-Level VLSI Synthesis*, R. Camposano and W. H. Wolf, Eds. Norwell, MA: Kluwer, May 1991, pp. 205–229 [Online]. Available: http://www-lab09.kuee.kyoto-u.ac.jp/parthenon/NTT

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

[23] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau, *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Pittsburgh, PA: Carnegie Mellon University, Jul. 2006 [Online]. Available: http://www.sei.cmu.edu/uls

[24] M. O'Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language* (Genetic Programming, vol. 4). Norwell, MA: Kluwer, May 2003.

[25] M. Orlov and M. Sipper, "Genetic programming in the wild: Evolving unrestricted bytecode," in *Proc. 11th GECCO*, Jul. 2009, pp. 1043–1050.

[26] T. Perkis, "Stack-based genetic programming," in *Proc. 1st ICEC*, vol. 1. Jun. 1994, pp. 148–153.

[27] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming* (with contributions by J. R. Koza). London, U.K.: Lulu Enterprises, Mar. 2008 [Online]. Available: http://www.gp-field-guide.org.uk

[28] M. D. Schmidt and H. Lipson, "Incorporating expert knowledge in evolutionary search: A study of seeding methods," in *Proc. 11th GECCO*, Jul. 2009, pp. 1091–1098.

[29] M. Sipper, Y. Azaria, A. Hauptman, and Y. Shichel, "Designing an evolutionary strategizing machine for game playing and beyond," *IEEE Trans. Syst. Man Cybern. Part C: Applicat. Rev.*, vol. 37, no. 4, pp. 583–593, Jul. 2007.

[30] L. Spector and A. Robinson, "Genetic programming and autoconstructive evolution with the Push programming language," *Genet. Program. Evolvable Mach.*, vol. 3, no. 1, pp. 7–40, Mar. 2002.

[31] H. Tuan-Hao, R. I. B. McKay, D. Essam, and N. X. Hoai, "Solving symbolic regression problems using incremental evaluation in genetic programming," in *Proc. CEC*, Jul. 2006, pp. 2134–2141.

[32] M. S. Withall, C. J. Hinde, and R. G. Stone, "An improved representation for evolving programs," *Genet. Program. Evolvable Mach.*, vol. 10, no. 1, pp. 37–70, Mar. 2009.

[33] M. L. Wong and K. S. Leung, *Data Mining Using Grammar Based Genetic Programming and Applications* (Genetic Programming, vol. 3). Norwell, MA: Kluwer, Feb. 2000.

[34] J. R. Woodward, "Evolving Turing complete representations," in *Proc. CEC*, vol. 2. Dec. 2003, pp. 830–837.

Michael Orlov received the B.Sc. degree (*summa cum laude*) and the M.Sc. degree (*cum laude*), both in computer science, from Ben-Gurion University, Beer-Sheva, Israel. He is currently pursuing the Ph.D. degree in computer science from Ben-Gurion University.

His current research interests include the application of evolutionary algorithms to software development.

Mr. Orlov was awarded the prestigious Adams Fellowship by the Israeli Academy of Sciences in 2008.

Moshe Sipper received the B.A. degree from the Technion-Israel Institute of Technology, Haifa, Israel, and the M.Sc. and Ph.D. degrees from Tel Aviv University, Ramat Aviv, Israel, all in computer science.

From 1995 to 2001, he was a Senior Researcher with the Swiss Federal Institute of Technology, Lausanne, Switzerland. He is currently a Professor of Computer Science with the Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel. He has published over 130 scientific papers. He is the author of two books: *Machine Nature: The Coming Age of Bio-Inspired Computing*, and *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*. His current research interests include evolutionary computation, mainly as applied to games and software development.

Dr. Sipper is an Associate Editor of the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, the IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, and *Genetic Programming and Evolvable Machines*. He is an editorial board member of *Memetic Computing*. He received the 1999 EPFL Latsis Prize, the 2008 BGU Toronto Prize for Academic Excellence in Research, and four HUMIE Awards in 2005, 2007, 2008, and 2009 from the Human-Competitive Results Produced by Genetic and Evolutionary Computation Competition.