Darwinian Software Engineering

Thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

by

Michael Orlov

Submitted to the Senate of

Ben-Gurion University
of the Negev

September 2013

Beer-Sheva · Israel

Darwinian Software Engineering

Thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

by

Michael Orlov

Submitted to the Senate of

Ben-Gurion University
of the Negev

Approved by the advisor ____________________

Approved by the Dean of the Kreitman School for Advanced Graduate Studies ____________________

September 2013

Beer-Sheva · Israel

This work was carried out under the supervision of **Prof. Moshe Sipper**

In the Department of Computer Science

In the Faculty of Natural Sciences

Declaration

RESEARCH STUDENT'S AFFIDAVIT FOR
SUBMITTING THE DOCTORAL THESIS FOR REFEREEING:

I, Michael Orlov, whose signature appears below, hereby declare that:

- I have written this Thesis by myself, except for the help and guidance offered by my Thesis Advisors.
- The scientific materials included in this Thesis are products of my own research, *culled from the period during which I was a research student*.

Student's name: Michael Orlov
Date: 2.10.2013
Signature:

Acknowledgments

WRITING THIS thesis would not have been possible without the extraordinary support that I received from my advisors, colleagues, and collaborators.

First and foremost I would like to thank Prof. Moshe Sipper for his steadfast expert guidance during the years of research and development presented here, and his unconditional overall support for my Ph.D. studies.

I am grateful to Prof. Natalio Krasnogor and to Dr. Mayer Goldberg for providing valuable input at the research proposal stage of this work. I was fortunate to meet with Prof. Krasnogor several times later in the course of my studies, and he always had helpful and insightful advice for the research process.

I am very thankful to Dr. Matthew Hyde for inviting me to collaborate on extending FINCH at University of Nottingham and afterwards at Ben-Gurion University, which has been a wonderful experience.

My sincere gratitude goes to Mr. Marcel Adams and the Israeli Academy of Sciences for generously supporting my studies and conference travel with Adams Fellowship, and for providing the amazing support network of Adams conferences and meetings. I am also grateful to Dr. Yossi Friedman and the Department of Computer Science at Ben-Gurion University for the Friedman Scholarship award.

Finally, I would like to praise my wonderful wife Marina — without her unreserved support, receiving a Ph.D. degree would have been beyond the realm of possibility.

Contents

Declaration v

Acknowledgments vii

Abstract 1

1 Introduction 3

2 Related Work 7

3 Bytecode Evolution 11
- 3.1 Why Target Bytecode for Evolution? 11
- 3.2 The Grammar Alternative 15
- 3.3 The Halting Issue 17
- 3.4 (No) Loss of Compiler Optimization 18
- 3.5 Bytecode Evolution Principles 19
- 3.6 Compatible Bytecode Crossover 22

4 Experimental Validation 27
- 4.1 Symbolic Regression: Simple and Complex 28
- 4.2 Artificial Ant 33
- 4.3 Intertwined Spirals 37
- 4.4 Array Sum 41
- 4.5 Tic-Tac-Toe 46

5 Conclusions 53

A Source Listings 57
- A.1 Artificial Ant: Avoider 57

Bibliography 61

Index 67

Glossary **69**

Hebrew Abstract (תקציר) **H:1**

A Hebrew translation of the abstract is adjacent to the thesis' back cover.

List of Figures

1 Java source code compilation and execution. 12
2 A recursive factorial function in Java and its corresponding bytecode. 12
3 Call frames in the architecture of the JVM. 13
4 An example of good and bad crossovers. 15
5 Correct and incorrect CFG-compliant Java snippets. 16
6 A BFS-like traversal of the data-flow graph 24
7 Illustrations to operand stack requirements in bytecode crossover constraints. 25
8 Illustrations to the local variables requirements in bytecode crossover constraints. 26

9 Tree representation of the worst generation-0 individual in the simple symbolic regression experiment of Koza [1992a]. 28
10 Simple symbolic regression in Java, Koza's worst generation-0 individual. 30
11 Decompiled evolved simple symbolic regression method. 31
12 Decompiled evolved simple symbolic regression method, another experiment. 31
13 Decompiled evolved complex symbolic regression method. 33
14 The Santa Fe food trail for the artificial ant problem, and the path taken by the *Avoider* individual in Koza's experiment. 34
15 Tree representation and Java implementation of Koza's *Avoider* individual. 34
16 Two evolved solutions to the artificial ant problem. 36
17 Ant trails that result from executing the evolved solutions. 36
18 Intertwined spirals dataset, Koza's original result, and two evolved solutions. 38
19 "Zooming out" of Koza's and FINCH solution. 40
20 Intertwined spirals seed method. 40
21 Decompiled evolved intertwined spirals method. 41
22 Koza's intertwined spirals best-of-run S-expression. 42
23 Evolving method of the seed individual for the array sum problem. . 42

24 Decompiled ideal array sum individual. 44
25 Evolving method of the seed individual for the *List* version of the array sum problem. 44
26 Decompiled ideal array sum individual (*List* version). 45
27 Evolving method of the seed individual for the recursive *List* version of the array sum problem. 45
28 Decompiled ideal array sum individual (recursive *List* version). . . . 46
29 An α-β-pruning variant of the minimax algorithm. 47
30 FINCH setup for improving imperfect tic-tac-toe strategies. 48
31 Decompiled evolved solution to an imperfect tic-tac-toe seed. 51

List of Tables

1 Operand stack and local variables array requirements during execution of the factorial method. 20
2 Operand stack and local variables requirements for several bytecode segments of the factorial method. 22

3 Summary of evolutionary runs. 27
4 Simple symbolic regression: Parameters. 29
5 Complex symbolic regression: Parameters. 32
6 Artificial ant: Parameters. 35
7 Intertwined spirals: Parameters. 39
8 Array Sum: Parameters. 43
9 Tic-tac-toe: Parameters. 49
10 Tic-tac-toe: Four different single-error imperfections and their game performance effect. 50

Abstract

THIS THESIS presents FINCH (Fertile Darwinian Bytecode Harvester), a methodology for evolving Java bytecode, enabling the evolution of *extant*, *unrestricted* Java programs, or programs in other languages that compile to Java bytecode. The name of this thesis, "Darwinian Software Engineering", reflects our optimistic view of how this methodology may affect integration of evolutionary computation into software engineering practices.

An evolutionary computation system requires the key features of selection, recombination, and evaluation of individuals through generations. The most complex of these features, once we try to apply them to programs in extant, generic programming language, is recombination. Our approach is based upon the notion of *compatible crossover*, which produces correct programs by performing operand stack-, local variables-, and control flow-based compatibility checks on source and destination bytecode sections.

As far as we are aware, this approach to evolving extant software is unique. In the beginning of this thesis, we contrast our approach with existing work that uses restricted subsets of the Java bytecode instruction set as a representation language for individuals in genetic programming.

We subsequently describe FINCH's implementation, outlining the algorithms that achieve compatible crossover for producing correct individuals. We discuss viability of alternative implementations, and describe how obstacles such as nontermination are dealt with.

Afterwards, 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 (JVM) architecture and type system, ultimately evolving human-readable solutions in the form of Java programs.

We hope that the ability to evolve Java programs will lead to a valuable new tool in the software engineer's toolkit.

Work described herein has been previously published in [Orlov and Sipper, 2009, 2010, 2011, Orlov et al., 2011].

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation, program modification*; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.2 [**Software Engineering**]: Design Tools and Techniques.[1]

Keywords

Software evolution, genetic programming, evolutionary computation, Java bytecode.

[1]According to the ACM Computing Classification System: `http://www.acm.org/about/class/1998`

Chapter 1

Introduction

In a recent comprehensive monograph surveying the field of Genetic Programming (GP), Poli et al. noted that:

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

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

In this work, 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 [Koza, 1992a, Poli et al., 2008]. 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 3.2). We therefore turn to yet another well-explored alternative: evolution of machine code [Nordin, 1997].

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) [Lindholm and Yellin, 1999]. The JVM is free to apply its own optimization techniques, such as Just-in-Time (JIT), on-demand compilation to native machine code—a process that is transparent to the user. The JVM implements a stack-based architecture with high-level language features such as object management and garbage collection, virtual function calls, and strong typing. The bytecode language itself is a well-designed assembly-like language with a limited yet powerful instruction set [Engel, 1999, Lindholm and Yellin, 1999]. Figure 2 on p. 12 shows a recursive Java program for computing the factorial of a number, and its corresponding bytecode.

The JVM architecture, illustrated in fig. 3 on p. 13, is successful enough that several programming languages compile directly to Java bytecode (e.g., Scala, Groovy, Jython, Kawa, JavaFX Script, and Clojure). Moreover, Java *decompilers* are available, which facilitate restoration of the Java source code from compiled bytecode. Since the design of the JVM is closely tied to the design of the Java programming language, such decompilation often produces code that is very similar to the original source code [Miecznikowski and Hendren, 2002].

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

Note that we do not wish to invent a language to improve upon some aspect or other of GP (efficiency, terseness, readability, etc.), as has been amply done (and partly summarized in chapter 2 and [Orlov and Sipper, 2009]). Nor do we wish to extend standard GP to become Turing complete, an issue which has also been addressed [Woodward, 2003]. Rather, conversely, our point of departure is an *extant*, highly popular, general-purpose language, with our aim being to render it evolvable. The ability to evolve Java programs will hopefully lead to a valuable new tool in the software engineer's toolkit.

The FINCH system, which affords the evolution of unrestricted bytecode, is described in chapter 3, beginning with questioning what exactly should we evolve

— why not simply target source code, or abstract syntax trees? After designating bytecode as the optimal evolutionary material, we consider dealing with non-termination of programs. Why is a straightforward limit on execution time not good enough, and what better alternatives does working with Java bytecode offer? In addition, does mangling bytecode lose compiler optimizations? The chapter describes surprising insights into the optimization issue. We then show precisely how compatible bytecode sections can be detected without losing viable crossover possibilities.

Chapter 4 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 start by reproducing experiments with no need for advanced programming features (although different primitive and class types, as well as library calls, are nevertheless taken advantage of), and then progress further to show FINCH's ability to automatically handle control flow, loops, and recursion. We show that using data structures is completely transparent to FINCH when solving the array sum problem, and that the rich set of available language features enables production of phenotypically "interesting" individuals in intertwined spirals. Tic-tac-toe experiment offers a glimpse into automatic improvement of extant software.

We end with concluding remarks and future work avenues in chapter 5.

Chapter **2**

Related Work

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

Stack-based GP (Stack GP) was introduced by Perkis [1994]. 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 [Montana, 1995]. 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. 3).

An early introduction to Java bytecode genetic programming (JBGP) was given by Lukschandl et al. [1998], who evolved bytecode sequences with a small set of simple arithmetic and custom branch-like macro instructions. Lukschandl et al. evolved very limited individuals with a single floating-point type in one local vari-

able and no control structures, and therefore only needed to consider effects of instruction blocks on operand stack depth in order to avoid stack overflow and underflow errors. A later work by Lukschandl et al. [2000] used this method in a distributed bytecode evolutionary system (DJBGP), and presented its application to a telecom routing problem. Similar approaches were independently introduced by other researchers, as bcGP [Harvey et al., 1999] (which also handles branching instructions, but does not discuss crossover compatibility) and Klahold et al. [1998] (which seems to leave compatibility checks to the Java verifier, although too few details are provided). The aforementioned approaches are conceptually limited to using Java bytecode as yet another genotype representation for GP. None can be applied to evolving correct individuals based on unrestricted bytecode—which we show how to do in the following section.

Once we consider non-bytecode stack-based GP, Tchernev [1998] offered a more thorough treatment of requirements for crossover in the programming language Forth, arguing that ensuring same-stack depth at crossover points is not only better than GP's popular subtree crossover, but is an engine for combining building blocks that is strictly different from a macromutation. However, similar to the works discussed previously, Tchernev considers only the stack depth in synthetic individuals with restricted primitives. Tchernev and Phatak [2004] later introduced a similar technique for correct crossover of high-level control structures. This work is not applicable at all to Java bytecode evolution, since control structures are not expressed as such in bytecode, and are instead translated into simpler `goto` instructions.

Evolutionary program induction using binary machine code (AIM-GP) was introduced by Nordin [1997] as the fastest known genetic programming method. Although Nordin et al. [1999] later mentioned Java as a possible evolutionary target, the paper is scarce on details. As of now, Discipulus, the commercial successor to AIM-GP, can only produce Java source code as a decompilation result from an evolved native machine code individual, as opposed to our goal of evolving the intermediate-level bytecode. In AIM-GP, the creation of viable offspring individuals from parent programs is realized via a careful multi-granularity crossover process. It is interesting to contrast this work with the attempt by Kühling et al. [2002] to forgo any constraints on code and on evolutionary operators, and instead trap all exceptions of code that is executed as a separate encapsulated entity. We do not expect this approach to overcome the huge search space that results from evolving Java programs with unrestricted crossover and mutation operators. However, since we decided that the evolutionary process should stay close to the

JVM, we cannot completely safeguard bytecode execution from exceptional conditions, as done e.g, by Huelsbergen [2005]. Thus, evaluating evolving bytecode individuals in an encapsulated environment—a *sandbox*—is still necessary.

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

Spector and Robinson [2002] 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 [Perkis, 1994] mostly apply to [Spector and Robinson, 2002] 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.

More recently, Servant et al. [2005] introduced JEB, an open-source tool for Java byte-code evolution, as an extension to the ECJ evolutionary computation software package [Luke and Panait, 2004]. In JEB, genotype and phenotype bytecodes are separate entities, and stack underflows are corrected during genotype-phenotype translation. Other limitations that we discussed previously—restricted instruction set, no handling of types, etc.—apply to this work as well. This is however an interesting approach—yet it is undesirable for evolving extant bytecode, since it introduces a separate representation for evolving programs and increases the search space. Reduction of search-space size is better achieved with properly-defined compatible evolutionary operators, as discussed next.

Another line of recent research related to ours is that of software repair by evolutionary means. Forrest et al. [2009] 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 4.5, 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 [2009] that repairs Java source code using syntax-tree transformations. It is discussed later in section 3.2. Overall, due to availability of techniques such as fault localization, software repair is a much easier problem to tackle than unrestricted evolution of programs, and this line of research has seen a lot of quality research, e.g., by Schulte et al. [2013].

An interesting direction of research is optimizing *non-functional* software properties, explored by White et al. [2011]. We did not focus on this direction, although it is partially related to unexpected phenotype qualities observed in intertwined spirals experiment in section 4.3.

Chapter 3

Bytecode Evolution

Bytecode is the intermediate, platform-independent representation of Java programs, created by a Java compiler. Figure 1 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.

3.1 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 [2002]—but rather aim to handle the Java programming language in particular.

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

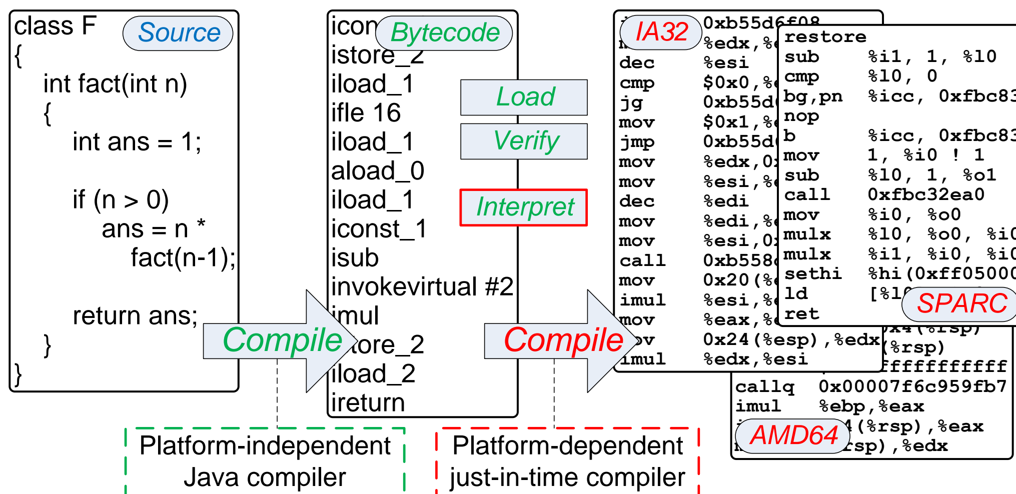


Figure 1. Java source code is first compiled to *platform-independent* bytecode by a Java compiler. The JVM only loads the bytecode, which it verifies for correctness, and raises an exception in case the verification fails. After that, the JVM typically interprets the bytecode until it detects that it 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 3.4).

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

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

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

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

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

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

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

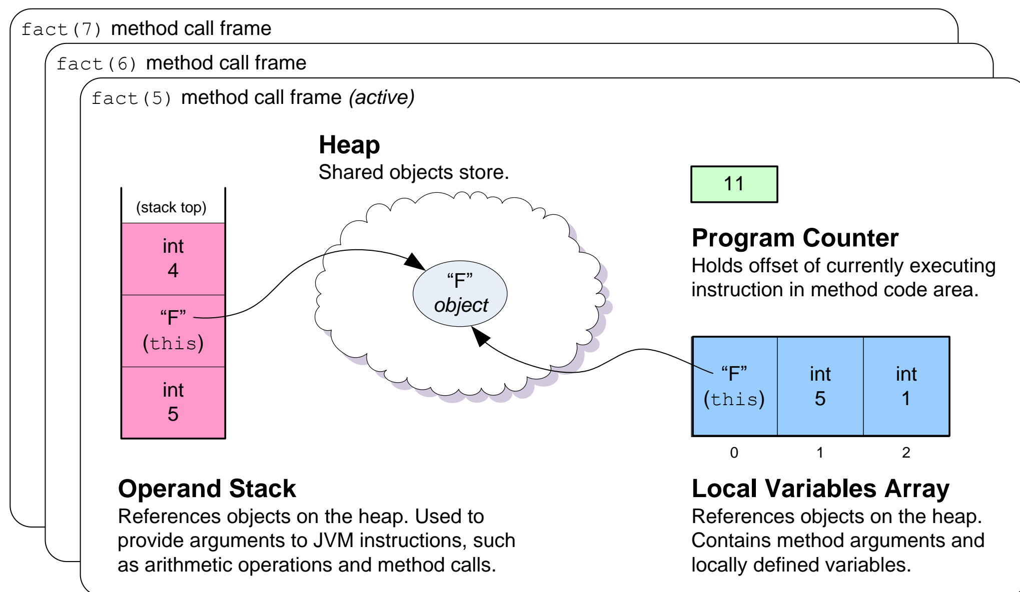


Figure 3. Call frames in the architecture of the Java Virtual Machine (JVM), during execution of the recursive factorial function code shown in fig. 2, with parameter $n = 7$. The top call frame is in a state preceding execution of **`invokevirtual`**. This instruction will pop a parameter and an object 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.

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 [Orlov and Sipper, 2009].

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? [Darwin, 1859]

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 [Orlov and Sipper, 2009] 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. 2, 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 [Orlov and Sipper, 2009]; then a segment with the chosen size is uniformly selected. Whenever the chosen segments result in *bad* crossover, bytecode segments are chosen again (up to some limit of retries). Note that this selection process is very fast (despite the retries), as it involves fast operations—and, most importantly, we ensure that crossover *always* produces a viable offspring. More

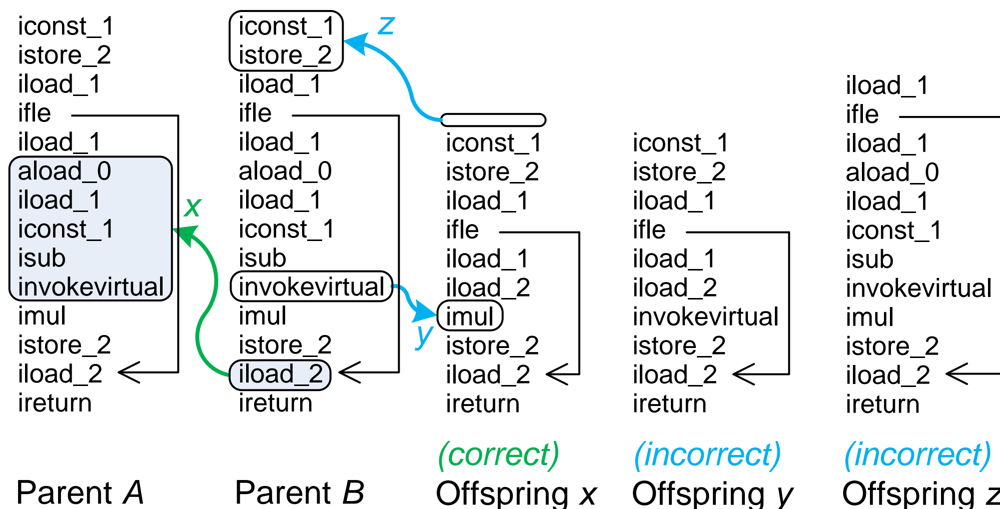


Figure 4. An example of good and bad crossovers. The two identical individuals *A* and *B* represent a recursive factorial function (see fig. 2; here we use an arrow instead of branch offset). In parent *A*, the bytecode sequence that corresponds to the `fact(n-1)` call that leaves an integer value on 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.

intelligent choices of bytecode segments are possible, as will be discussed in chapter 5.

The full formal specification of compatible bytecode crossover is provided in sections 3.5 to 3.6. 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 chapter 5. Only a basic constant mutator (section 4.3) was implemented.

3.2 The Grammar Alternative

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

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

(a) A correct Java snippet.

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

(b) An incorrect Java snippet.

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

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

We cannot emphasize the distinction between syntax and semantics strongly enough. Consider, for example, the Java program segment shown in 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. [1994] implemented the com-

plete grammar of SFL (Structured Function description Language) [Nakamura et al., 1991] as production rules of a rewriting system, using approximately 350(!) rules for a language far simpler than Java. The semantic constraints of SFL—an object-oriented, register-transfer-level language—are sufficiently weak for using its BNF directly:

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

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

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

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

3.3 The Halting Issue

An important issue that must be considered when dealing with the evolution of unrestricted programs is whether they halt—or not [Langdon and Poli, 2006].

Whenever Turing-complete programs with arbitrary control flow are evolved, a possibility arises that computation will turn out to be unending. A program that has acquired the undesirable non-termination property during evolution is executed directly by the JVM, and FINCH has nearly no control over the process.

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

3.4 (No) Loss of Compiler Optimization

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

Surprisingly, however, bytecode evolution does *not* induce loss of compiler optimization, since there is no optimization to begin with! The common assumption regarding Java compilers' similarity to native-code compilers is simply incorrect. As far as we were able to uncover, with the exception of the IBM Jikes

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

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 [Kotzmann et al., 2008].

The fact that Java compilers do not optimize bytecode does not preclude the possibility of doing so, nor render it particularly hard in various cases. Indeed, in FINCH we apply an automatic post-crossover bytecode transformation that is typically performed by a Java compiler: dead-code elimination. After crossover is done, it is possible to get a method with unreachable bytecode sections (e.g., a forward **`goto`** with no instruction that jumps into the section between the **`goto`** and its target code offset). Such dead code is problematic in Java bytecode, and it is therefore automatically removed from the resulting individuals by our system. This technique does not impede the ability of individuals to evolve introns, since there is still a multitude of other intron types that can be evolved [Brameier and Banzhaf, 2006] (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).

3.5 Bytecode Evolution Principles

The Java virtual machine is a stack-based architecture for executing Java bytecode. The JVM holds a stack for each execution thread, and creates a frame on this stack for each method invocation. The frame contains a code array, an operand stack, a local variables array, and a reference to the constant pool of the current class [Engel, 1999]. The code array contains the bytecode to be executed by the JVM. The local variables array holds all method (or function) parameters, including a reference to the class instance in which the current method executes. In addition, the variables array also holds local-scope variables. The operand stack is

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

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

used by stack-based instructions, and for arguments when calling other methods. A method call moves parameters from the caller's operand stack to the callee's variables array; a return moves the top value from the callee's stack to the caller's stack, and disposes of the callee's frame. Both the operand stack and the variables array contain typed items, and instructions always act on a specific type. The relevant bytecode instructions are prefixed accordingly: '`a`' for an object or array reference, '`i`' and '`l`' for integral types `int` and `long`, and '`f`' and '`d`' for floating-point types `float` and `double`.[3] Finally, the constant pool is an array of references to classes, methods, fields, and other unvarying entities. The JVM architecture is illustrated in fig. 3

To demonstrate the operation of the JVM, consider a simple recursive program for computing the factorial of a number, shown in fig. 2 on p. 12. Table 1 shows a step-by-step execution of the bytecode. The operand stack is initially empty, and the local variables array contains a reference to **`this`** (the current class instance) at index 0, and the parameter n at index 1. The local variable `ans` is allocated the index 2, but the corresponding cell is uninitialized.

In our evolutionary setup, the individuals are bytecode sequences annotated with all the stack and variables information shown in table 1. This information is gathered in one pass over the bytecode, using the ASM bytecode manipulation and analysis library [Bruneton et al., 2002]. Afterwards, similar information for any sequential code segment in the individual can be aggregated separately—table 2 shows this information for several bytecode segments. This preprocessing step allows us to realize compatible two-point crossover on bytecode sequences. Code segments can be replaced only by other segments that use the operand stack and the local variables array in a depth-compatible and type-compatible manner. The compatible crossover thus maximizes the viability potential for offspring, preventing type incompatibility and stack underflow errors that would otherwise plague indiscriminating bytecode crossover. Note that the crossover operation is *unidirectional*, or asymmetric—the code segment compatibility criterion as described here is not a symmetric relation. An ability to replace segment α in individual A with segment β in individual B does not imply an ability to replace segment β in B with segment α.

[3] The types `boolean`, `byte`, `char` and `short` are treated as the computational type `int` by the Java virtual machine, except for array accesses and explicit conversions [Lindholm and Yellin, 1999, §3.11.1].

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

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

3.6 Compatible Bytecode Crossover

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

Variables Access Sets

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

- $\delta^r_{a,b}$: set of local variables such that for each variable v, there exists a *potential* execution path (i.e., one not necessarily taken) between a and b, in which v is read before any write to it; this set of variables is the *vars read* column in table 2;
- $\delta^w_{a,b}$: set of local variables that are written to through at least one potential execution path; the corresponding column in table 2 is *vars written*;

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

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

- $\delta^r_{a,c}$ is the union of all reads δ^r_{a,b_i}, with the addition of variables read by instruction c—unless these variables are guaranteed to be written before c. Formally, $\delta^r_{a,c} = \left(\bigcup_i \delta^r_{a,b_i}\right) \cup \left(\delta^r_c \setminus \bigcap_i \delta^{w!}_{a,b_i}\right)$.
- $\delta^w_{a,c}$ is the union of all writes δ^w_{a,b_i}, with the addition of variables written by instruction c: $\delta^w_{a,c} = \left(\bigcup_i \delta^w_{a,b_i}\right) \cup \delta^w_c$.
- $\delta^{w!}_{a,c}$ is the set of variables guaranteed to be written before c, with the addition of variables written by instruction c: $\delta^{w!}_{a,c} = \left(\bigcap_i \delta^{w!}_{a,b_i}\right) \cup \delta^{w!}_c$ (note that $\delta^{w!}_c = \delta^w_c$). When $\delta^{w!}_{a,c}$ has already been computed, its previous value needs to be a part of the intersection as well.

We therefore traverse the data-flow graph as shown in fig. 6, starting at a, and updating the variables access sets as above, until they stabilize—i.e., stop changing.[4] During the traversal, necessary stack depths—such as the number of pops in table 2—are also updated. The requirements for compatible bytecode crossover can now be specified.

Bytecode Constraints on Crossover

In order to attain viable offspring, several conditions must hold when performing crossover of two bytecode programs. Let A and B be functions in Java, represented as bytecode sequences. Consider segments α and β in A and B, respectively, and let p_α and p_β be the necessary depth of stack for these segments—i.e, the minimal number of elements in the stack required to avoid underflow. Segment α can be replaced with β if the following conditions hold.

1. Operand stack (illustrated in fig. 7):
   (a) it is possible to ensure that $p_\beta \leqslant p_\alpha$ by prefixing stack pops and pushes of α with some frames from the stack state at the beginning of α;

[4] The data-flow traversal is similar in nature to the data-flow analyzer's loop in [Lindholm and Yellin, 1999, §4.9.2].

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

Figure 6. Compute-Accesses(a, b): A BFS-like traversal of the data-flow graph starting at location a and ending at b, in order to compute variables accessed in the code segment $[a, b]$. $\delta^*_{x,y}$ denotes the three access sets $\delta^r_{x,y}$, $\delta^w_{x,y}$, and $\delta^{w!}_{x,y}$. Here, a branch denotes natural transitions to subsequent instruction as well as transitions resulting from conditional and unconditional branching instructions. The inner **if** clause ensures that a "natural" transition at the end of segment $[a, b]$ is not unnecessarily followed.

   (b) α and β have compatible stack frames up to depth p_β: stack pops of α have identical or narrower types as stack pops of β, and stack pushes of β have identical or narrower types as stack pushes of α;
   (c) α has compatible stack frames deeper than p_β: stack pops of α have identical or narrower types as corresponding stack pushes of α.
2. Local variables (illustrated in fig. 8):
   (a) local variables written by β (β^w) have identical or narrower types as corresponding variables that are read after α (*post*-α^r);
   (b) local variables read after α (*post*-α^r) and not necessarily written by β ($\beta^{w!}$) must be written before α (*pre*-$\alpha^{w!}$), or provided as arguments for call to A, as identical or narrower types;
   (c) local variables read by β (β^r) must be written before α (*pre*-$\alpha^{w!}$), or provided as arguments for call to A, as identical or narrower types.
3. Control flow:
   (a) no branch instruction outside of α has branch destination in α, and no branch instruction in β has branch destination outside of β;
   (b) code before α has transition to the first instruction of α, and code in β has transition to the first instruction after β;
   (c) last instruction in α implies transition to the first instruction after α.

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

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

Bad	α	β
pre-stack	`*`	`****`
post-stack	∅	`***`
depth	1	2

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

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

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

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

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

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

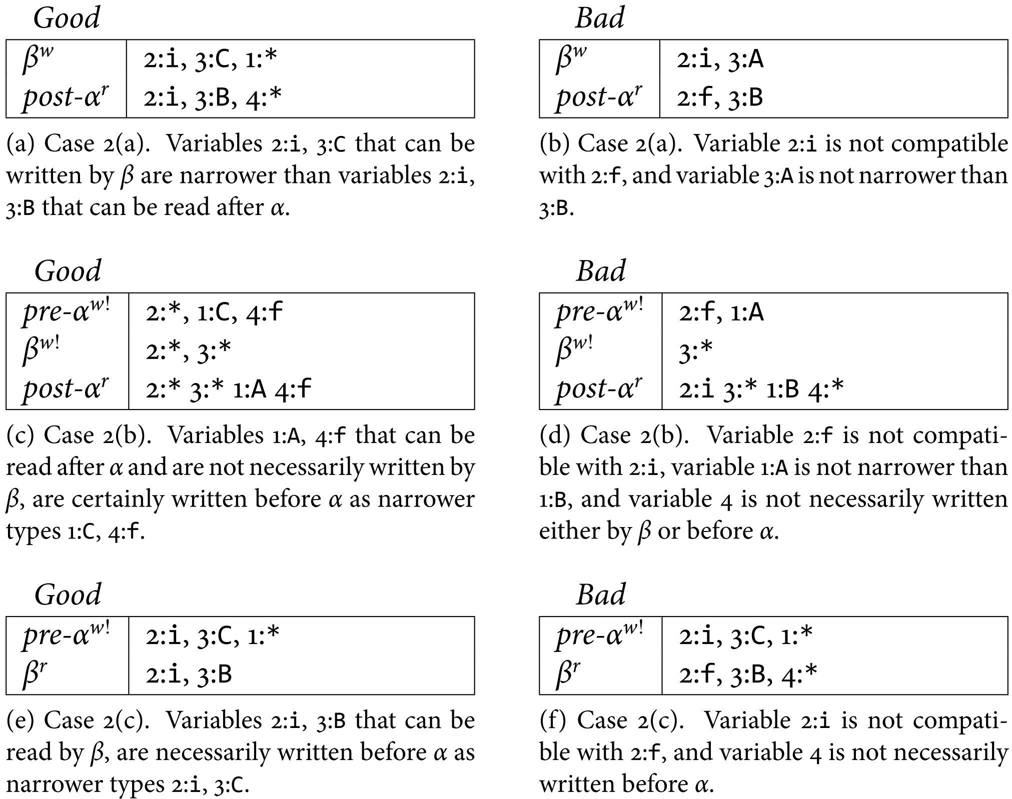
(e) Case 1(c). Stack pops "`iB`" (extra 2 stack frames) are narrower than stack pushes "`iA`".

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

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

Figure 7. Illustrations to the operand stack requirements in bytecode crossover constraints. Here, we assume that class `B` extends class `A`, and `B` is thus a narrower type than `A`, and that class `C` similarly extends class `B`. The symbols `i` and `f` denote the primitive types `int` and `float`. The `*` symbol is used in cases where the precise type does not matter. For stacks, the topmost value is shown at the right-hand side. *Pre-stack* and *post-stack* are states of stack before and after execution of code segment.

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



Good

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

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

Bad

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

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

Good

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

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

Bad

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

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

Good

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

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

Bad

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

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

Figure 8. Illustrations to the local variables requirements in bytecode crossover constraints. Notation and types used are similar to fig. 7; x:`y` stands for read or write access to local variable x with type `y`. For example, `2:i` in β^r means that segment β reads variable 2 as an `int`.

Chapter 4

Experimental Validation

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 [Orlov and Sipper, 2009]. For this purpose we integrated our framework, which uses ASM [Bruneton et al., 2002], with the ECJ evolutionary framework [Luke and Panait, 2004], with ECJ providing the evolutionary engine. Now we are ready to apply FINCH to a selection of problems of interest.

Table 3. Summary of evolutionary runs. 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.

Section	Problem		Yield
4.1	Simple symbolic regression		99%
	Complex symbolic regression		100%
4.2	Artificial ant on Santa Fe trail		2%
		population of 2 000	11%
		population of 10 000, 101 generations	35%
4.3	Intertwined spirals		10%
4.4	Array sum		77%
		`List`*-based seed individual*	97%
		`List`*-recursive seed individual*	100%
4.5	Tic-tac-toe:	`alpha`/`save` *imperfection*	96%
		unary "`-`*" imperfection*	88%
		`false`/`save` *imperfection*	100%
		`alpha-beta` *imperfection*	88%

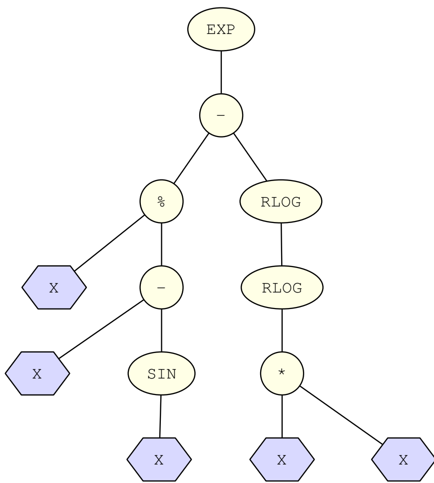


Figure 9. Tree representation of the worst generation-0 individual in the original *simple symbolic regression* experiment of Koza [1992a]. Functions are represented by inner tree nodes, and terminals by leaves. The corresponding mathematical expression for $x \neq 0, 1$ is $e^{\frac{x}{x-\sin x} - \ln|\ln x^2|}$, 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).

Throughout this chapter we describe typical results (namely, Java programs) produced by our evolutionary runs. The consistency of FINCH's successful yield is shown in table 3, wherein we provide a summary of all runs performed. Table 3 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 [Sipper et al., 2007]). Of course, increasing population size (see table 3) incurred a time increase.

4.1 Symbolic Regression: Simple and Complex

We begin with a classic test case in GP—*simple symbolic regression* [Koza, 1992a]—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 [Langdon and Nordin, 2000, Poli et al., 2008, Schmidt and Lipson, 2009]. We selected the *worst* generation-0 individual in Koza's original experiment, shown in fig. 9, and translated it into Java (fig. 10).

The worst generation-0 individual (fig. 9) is the Lisp S-expression

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

Table 4. Simple symbolic regression: Parameters. (Note: In the parameter tables shown throughout this paper we divide the parameters into four categories, separated by boldface lines: objective-related, structural, generic, and specific.)

Parameter	Koza [1992a, 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. 10)
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. 10
Crossover location	internal nodes with $p_{\text{int}} = 0.9$, otherwise a terminal	uniform distribution over segment sizes

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

```
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 */ }}
```

Figure 10. *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 [1992a], 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.

of primitive components—the 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 4: 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 16–24), as discussed in section 3.1.

An ideal individual was found in nearly every run. Typical evolutionary results are shown in figs. 11 and 12.

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. [2006], 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

```
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 */ }}
```

Figure 11. Decompiled contents of method `simpleRegression` that evolved after 17 generations from the Java program in fig. 10. 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 */ }}
```

Figure 12. 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. 11, 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.)

a fitness function in the spirit of Tuan-Hao et al. [2006], based on the highest degree n computed by an evolving individual.

We ran FINCH with clones of the same worst-case `simpleRegression` method used previously (fig. 10) serving as the initial population. The evolutionary parameters are shown in table 5: 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

Table 5. Complex symbolic regression: Parameters.

Parameter	Tuan-Hao et al. [2006]	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. 10)
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 $2 \sim 1000$	copies of seed program given in fig. 10
Crossover location	sub-tree crossover and sub-tree mutations	Gaussian distribution over segment sizes with 3σ = method size

a minor component we added herein to provide lexicographic parsimony pressure [Luke and Panait, 2002] 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 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 `sim-`

```
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);
}
```

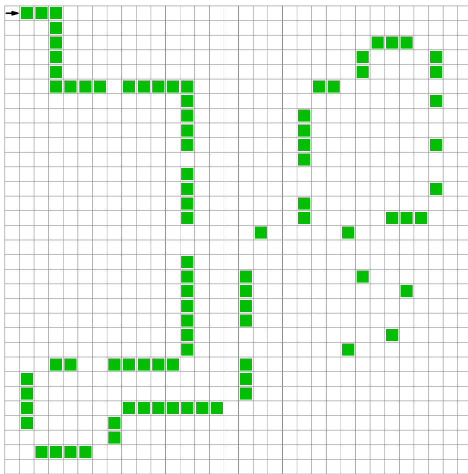
Figure 13. Decompiled contents of method `simpleRegression` that evolved after 19 generations in the complex symbolic regression experiment. The evolutionary result proceeds as $x + (x \cdot (x \cdot (x + (x + (((x \cdot x + x) \cdot x + x) \cdot x + x) \cdot x) \cdot x) + x) + x) \cdot x$, where x is 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.

`pleRegression` 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 5 times the number of bytecode instructions in the initial method).

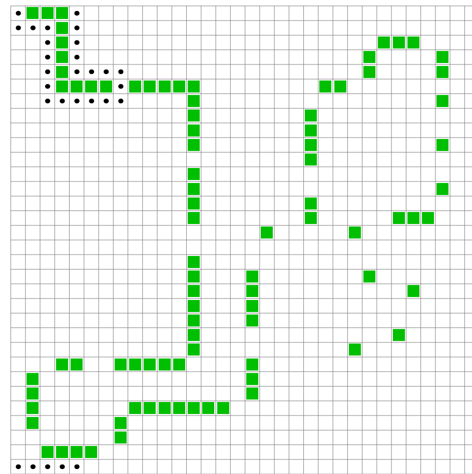
An ideal individual was found in every run. A typical evolved method is shown (without the wrapper class) in fig. 13. As a side issue, we also tested whether straightforward, non-incremental fitness evaluation can be used—a test which proved successful: We were able to evolve degree-9 polynomials directly, using a simple initial-population individual consisting only of instructions computing $(x + x) \cdot x$.

4.2 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 [Koza, 1992a], 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

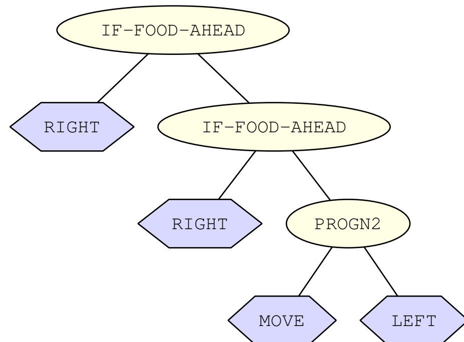


(a) The Santa Fe trail. The ant starts at the upper-left corner facing right, its objective being to consume 89 food pellets.



(b) Path taken by the *Avoider* individual, shown in fig. 15, around the food pellets that are marked by colored cells.

Figure 14. The 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 [1992a]. Note that the grid is toroidal.



(a) Tree representation of the Lisp individual.

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

(b) Implementation in Java. See section A.1 for class context.

Figure 15. The *Avoider* individual in the original experiment of Koza [1992a, 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.

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. 14(a), where the ant starts at the upper-left corner and faces right.

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. 14(b). We chose this zero-fitness individual for our initial population, implementing Avoider as a straightforward and unconstrained Java function

Table 6. Artificial ant: Parameters.

Parameter	Koza [1992a, ch. 7.2]	FINCH
Objective	single step function for artificial ant that moves and eats food pellets on Santa Fe trail (fig. 14(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. 15(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*	5 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. 15(b)
Crossover location	internal nodes with $p_{\text{int}} = 0.9$, otherwise a terminal	Gaussian distribution over segment sizes with 3σ = method size

called `step`, as shown in fig. 15 (along with the original Lisp-tree representation). The complete artificial ant implementation is listed in section A.1.

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.

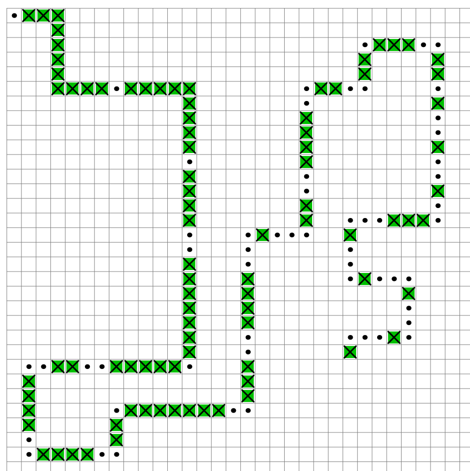
```java
void step() {
  if (foodAhead()) {
    move();
    right();
  }
  else {
    right();
    right();
    if (foodAhead())
      left();
    else {
      right();
      move();
      left();
    }
    left();
    left();
  }
}
```

(a) An optimal individual that appeared in generation 45. It makes no unnecessary moves, as can be seen in the corresponding ant trail in fig. 17(a).

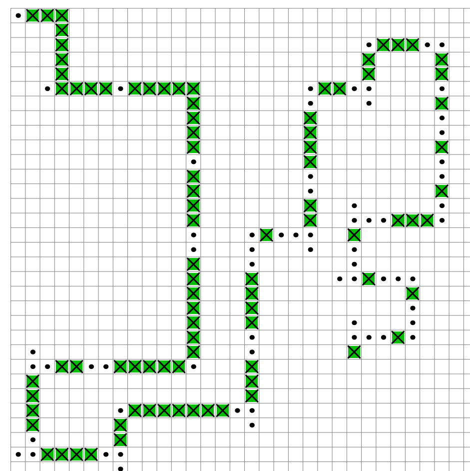
```java
void step() {
  if (foodAhead()) {
    move(); move();
    left(); right();
    right(); left();
    right();
  } else {
    right(); right();
    if (foodAhead()) {
      move(); right();
      right(); move();
      move(); right();
    } else {
      right(); move();
      left();
    }
 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

(b) A solution that appeared in generation 31. It successfully consumes all the food pellets, but makes some unnecessary moves, as shown in fig. 17(b).

Figure 16. The `step` methods of two solutions to the artificial ant problem that were evolved by FINCH. The corresponding ant trails are shown in fig. 17.



(a) The trail of the optimal individual shown in fig. 16(a).



(b) The trail of the non-optimal (though all-consuming) individual shown in fig. 16(b).

Figure 17. Ant trails that result from executing the artificial ant programs that contain the evolved `step` methods shown in fig. 16.

We ran FINCH with clones of the Java implementation of *Avoider* (fig. 15(b)) serving as the initial population. Again, we used the same parameters of Koza where possible, as shown in table 6: 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 5 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.

Figure 16 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. 17. Table 3 shows the success rate (i.e., percentage of runs producing all-consuming individual) for runs using the settings in table 6, and also the yield after increasing the population size and removing parsimony pressure (the inverse `step` method size component of the fitness function).

4.3 Intertwined Spirals

In the intertwined spirals problem the task is to correctly classify 194 points on two spirals, as shown in fig. 18(a). The points on the first spiral are given in polar coordinates by

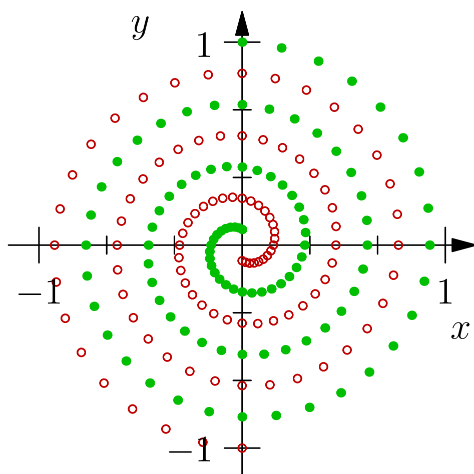
$$r_n = \frac{8+n}{104}\ , \qquad \alpha_n = \frac{8+n}{16}\cdot\pi\ ,$$

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

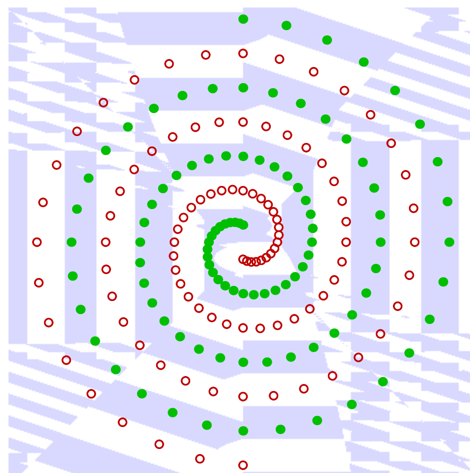
$$\begin{aligned} x_n^+ &= r_n \cos\alpha_n \\ y_n^+ &= r_n \sin\alpha_n \end{aligned}\ , \qquad \begin{aligned} x_n^- &= -x_n^+ \\ y_n^- &= -y_n^+ \end{aligned}\ ,$$

where (x_n^+, y_n^+) are points on the first spiral, and (x_n^-, y_n^-) lie on the second spiral [CMU, 1993].

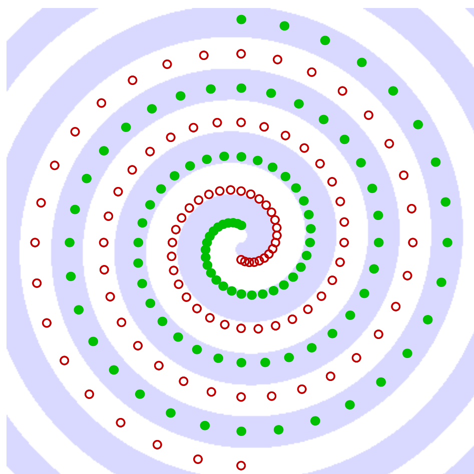
A classic machine-learning case study, the intertwined spirals problem was treated by Koza [1992a] using the parameters shown in table 7, with his best-of-run individual including 11 conditionals and 22 constants (shown in fig. 22). Whereas Koza used a slew of ERCs (ephemeral random constants) in the initial



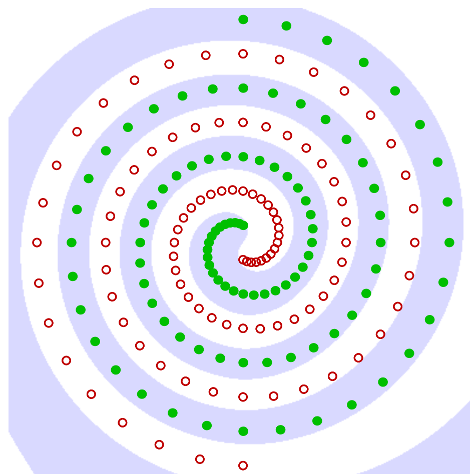
(a) Intertwined spirals, as described by Koza [1992a]. The two spirals, containing 97 points each, encircle the axes' origin three times. The first spiral (filled circles) belongs to class +1, and the second spiral (empty circles) belongs to class −1. The farthest point on each spiral is at unit distance from the origin.



(b) Visualization of the solution evolved by Koza [1992a] (shown in fig. 22), 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. 21, 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.

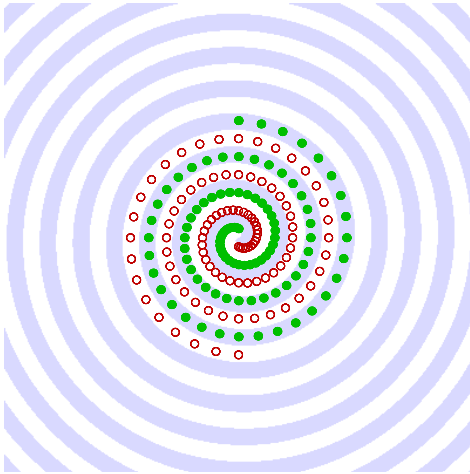
Figure 18. The intertwined spirals dataset (a) and the visualizations of two evolved (perfect) solutions (c), (d), contrasted with the result produced by Koza (b). Note the smoothness of FINCH's solutions as compared with the jaggedness of Koza's.

Table 7. Intertwined spirals: Parameters.

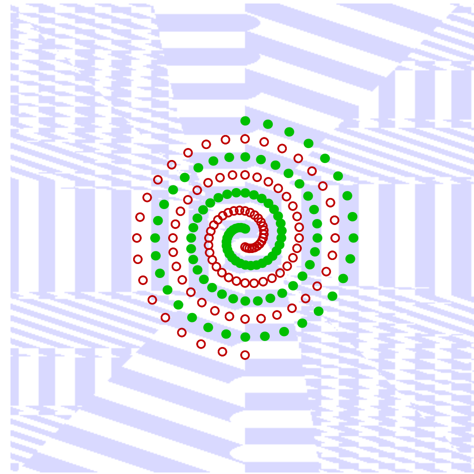
Parameter	Koza [1992a, ch. 17.3]	FINCH
Objective	two-class classification of intertwined spirals (fig. 18(a))	
Fitness	the number of points that are correctly classified	
Success predicate	all 194 points are correctly classified	
Terminals	`X`, `Y`, $\Re$ (ERC in $[-1, 1]$)	`double x`, `y` (inputs)
Functions	`+`, `-`, `*`, `%` (protected division), `IFLTE` (four-argument *if*), `SIN`, `COS`	built-in control flow and the functions present in the seed individual (fig. 20)
Population	10 000 individuals	2 000 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. 20
Crossover location	internal nodes with $p_{\text{int}} = 0.9$, otherwise a terminal	Gaussian distribution over segment sizes with 3σ = method size

population, FINCH's initial population is seeded with clones of a single program (shown in fig. 20), 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. 20. 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: Figure 21 shows a



(a) Solution in fig. 18(c) scaled to span the $[-2, 2]$ interval on both axes.



(b) Koza's solution in fig. 18(b), scaled similarly to (a).

Figure 19. Solutions smoothness is even more striking when "zooming out," as shown in (a) and (b).

```
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;
}
```

Figure 20. The 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.

typical evolved result, visualized in fig. 18(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 7), with his best-of-run comprising 91 internal nodes. This individual, shown in fig. 22, can be observed to be far "quirkier" than solutions evolved by FINCH (e.g., fig. 21). Figure 18(d)

```
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;
}
```

Figure 21. Decompiled contents of method `isFirst` that evolved after 86 generations from the Java program in fig. 20. 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 [1992a], 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.

shows another visualization of an evolved solution, and fig. 19 presents more intuition with regards to solution quality.

As done by Koza [1992b], 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.

4.4 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 section whether FINCH can handle two of the most im-

```
(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)))))
```

Figure 22. The best-of-run S-expression evolved by Koza [1992a] at generation 36, visualized in fig. 18(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.

```
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;
}
```

Figure 23. The 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 [Withall et al., 2009]—an exception is automatically thrown by the JVM in case of an out-of-bounds index use.

portant 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. [2009] 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 [O'Neill and Ryan, 2003], 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-

Table 8. Array Sum: Parameters.

Parameter	Withall et al. [2009]	FINCH
Objective	summation of numbers in an input array	
Fitness	negative total of differences from array sums on 10 predefined test inputs	as in [Withall et al., 2009], + inverse of `sumlist` method size
Success predicate	the sums calculated for 10 test inputs and 10 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	7 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	1 individual	
Growth limit	*fixed length*	maximal growth factor 2.0
Time limit	1 000 loop iterations	5 000 backward branches
Initial population	randomly-generated integer vector genomes	copies of seed program given in fig. 23
Crossover location	uniform crossover between fixed-length genomes	Gaussian distribution over segment sizes with 3σ = method size

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.

Table 8 shows the evolutionary setups used by Withall et al. [2009] and by us for the array sum problem (called *sumlist* by Withall et al.). During evaluation, individuals are given 10 predefined input lists of lengths 1–4, and if a program correctly computing all the 10 sums is found, it is also validated on 10 predefined verification inputs of lengths 1–9. Our initial population was seeded with

```
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;
}
```

Figure 24. Decompiled ideal individual that appeared in generation 11, correctly summing up all the test and validation inputs. Variable names were manually restored for clarity.

```
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;
}
```

Figure 25. The 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 [Gosling et al., 2005], as is evident in the best-of-run result in fig. 26.

copies of the blatantly unfit Java program in fig. 23. 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 8—*time limit*—regards the CPU resources allocated to a program during its evaluation, a measure discussed in section 3.3.

FINCH encountered little difficulty in finding solutions to the array sum problem (see table 3). One evolved solution is shown in fig. 24. 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 were taken modulo array size. FINCH, however, has no such abstract model

```java
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;
}
```

Figure 26. 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.

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

Figure 27. The 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.

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. Figure 25 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. 26 for one such 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;
}
```

Figure 28. Decompiled ideal individual for the recursive `List` array sum problem version, which appeared in generation 2. Java 5.0 syntax features were not restored.

Having demonstrated FINCH's ability to handle loops—we now turn to recursion. Figure 27 shows the seed individual 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 3.3. Solutions evolve readily for the recursive case as well—see fig. 28 for an example.

4.5 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.[1]

Figure 29 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, demonstrating four distinct, plausible, single-error slips that a

[1]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 a horizontal, vertical, or diagonal row.

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 ∨ d = 0 then
    return c · Utility(node)
else
    foreach succ ∈ successors of node do
        α ← max(α, −Negamax(succ, d − 1, −β, −α, −c))
        if α ⩾ β then
            return α
    return α
```

Figure 29. 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 Negamax(root, d, $-\infty$, ∞, 1). The function Utility returns a heuristic node value for the player with color $c = 1$.

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. 30.

Given a good-but-not-perfect tic-tac-toe program, i.e., an imperfect version of fig. 30, we set FINCH loose. In their work on evolving tic-tac-toe players, Angeline and Pollack [1993] 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 9 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 $\mathrm{sgn}(\frac{1}{m_1} - \frac{1}{m_2})$, 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).[2] The fitness value of an individual is simply the number of rounds won, and is in the range $\{0, \ldots, k\}$.

[2] 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   // 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 }
```

Figure 30. 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. 29) 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. 29 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.

Ties are broken randomly. This approach gives preference to players that take less moves to win.

Table 10 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 performance against two of the standard players defined by Angeline and Pollack [1993]: RAND and BEST. The former plays randomly and the latter is an optimal player, based on the correct negamax implementation shown in fig. 30 (so in our case it also minimizes the number of moves to win or draw). We see that although

Table 9. Tic-tac-toe: Parameters.

Parameter	Angeline and Pollack [1993]	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. 30, 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. 30, including the `play` tic-tac-toe board instance method
Population	256 individuals	2 048 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*	7 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. 30
Crossover location	internal nodes with $p_{\text{int}} = 0.9$, otherwise a terminal	Gaussian distribution over segment sizes, $\sigma = 3.0$

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 3, 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. Figure 31 shows one interesting, subtle example of a solution evolved from the `alpha-beta` swap imperfect seed (last case of table 10).

Table 10. Tic-tac-toe: 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. 30. Performance is shown as percentage of wins, draws, and losses vs. two players: RAND and BEST. (Note that BEST never loses.)

		RAND			BEST	
Line	Single-error imperfection	W	D	L	D	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

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.

```
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 }
```

Figure 31. Decompiled Java method in a solution evolved from the `alpha-beta` swap imperfect seed in table 10. Compare line 22 above with line 20 of fig. 30. Variable names were manually restored according to fig. 30 (Java 5.0 for-each loop was not restored).

Chapter 5

Conclusions

We presented a powerful tool by which extant software, written in the Java programming language, or in a language that compiles to Java bytecode, can be evolved directly, without an intermediate genomic representation, and with no restrictions on the constructs used. We 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.

It is also important to distinguish FINCH's methodology from approaches aimed at repairing software [Arcuri, 2009, Forrest et al., 2009]. Fixing programs typically relies on coarse-grained edits at the statement level of the abstract syntax tree, and on techniques such as fault localization using delta debugging [Le Goues, 2013]. Such approach, although certainly viable, is inherently limited in its applicability, as it requires precise positive and negative test cases for a certain bug in order to repair it by exploiting a drastically reduced search space — there is no evolution of a program as a whole.

We have shown that FINCH is applicable to both evolving "deplorable" first drafts (see sections 4.1 to 4.4) and to improving reasonable program drafts (see section 4.5), and in all cases the search space is completely unrestricted, and is only guided by the goal-driven fitness function. Of course, e.g., agile software

development methods such as test-driven development can and should be used with FINCH whenever possible, but demonstrating viability of such methods was never our goal. Instead, we strove to develop a generic methodology for evolving software in unrestricted search space, without relying on support of user-supplied infrastructure like precise positive and negative inputs, and without limiting the applicability of the methodology to repairing programs.

A recent study commissioned by the US Department of Defense on the subject of futuristic ultra-large-scale (ULS) systems that have billions of lines of code noted, among others, that, "Judiciously used, digital evolution can substantially augment the cognitive limits of human designers and can find novel (possibly counterintuitive) solutions to complex ULS system design problems" [Northrop et al., 2006, p. 33]. This study does not detail any actual research performed but attempts to build a road map for future research. Moreover, it concentrates on huge, futuristic systems, whereas our aim is at current systems of any size (with the proof-of-concept described herein focusing on relatively small software systems). Differences aside, both our work and this study 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: (a) defining a process by which consistent bytecode segments can be *found* during compatibility checks, thus improving preservation of evolutionary components during evolution; (b) supporting class-level evolution, such as cross-method crossover and introduction of new methods; (c) development of mutation operators, currently

lacking (except for the constant mutator of section 4.3); (d) applying FINCH to additional hard problems, along the way garnering further support for our approach's efficacy; (e) directly handling high-level bytecode constructs such as try/-catch clauses and monitor enter/exit pairs; (f) designing an IDE (integrated development environment) plugin to enable the use of FINCH in software projects by non-specialists; (g) applying FINCH to meta-evolution, in order to discover better evolutionary algorithms; (h) applying unrestricted bytecode evolution to the automatic improvement of existing applications, establishing the relevance of FINCH to the realm of extant software.

Progress along these directions can be already seen with automatic evolution of game heuristics [Orlov et al., 2011] and development of visualization plugins [Elyasaf et al., 2013].

It should be noted that the work presented here focused on evolving program *code*, and not software architecture as a whole. We consider it an open question whether Darwinian evolution of software *architecture* — i.e., introduction of new classes, new methods, refactoring code, changing inheritance relationships, modularization, etc. — is as important as evolution of code for achieving the ultimate goal of Darwinian software engineering, which is still a distant target. After all, software architecture is mostly intended to assist human software engineers, whereas code is executed by a CPU (or, in our case, a JVM). Automatic software evolution cannot be done without the latter, but can it avoid the former? Answering this question requires a major experimental investigation.

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 A

Source Listings

A.1 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 [1992a], 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); }
  }
```

Bibliography

CMU neural network benchmark database, Feb. 1993. URL `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/cmu`. 37

P. J. Angeline and J. B. Pollack. Competitive environments evolve better solutions for complex tasks. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, July 17–21, 1993, Urbana-Champaign, Illinois, USA*, pages 264–270, San Francisco, CA, USA, July 1993. Morgan Kaufmann. ISBN 1-55860-299-2. 47, 48, 49

A. Arcuri. *Automatic Software Generation and Improvement Through Search Based Techniques*. PhD thesis, University of Birmingham, Birmingham, UK, Dec. 2009. URL `http://etheses.bham.ac.uk/400/`. 10, 17, 53

M. F. Brameier and W. Banzhaf. *Linear Genetic Programming*. Genetic and Evolutionary Computation. Springer, New York, NY, USA, Dec. 2006. ISBN 978-0-387-31029-9. 19

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 *Adaptable and Extensible Component Systems (Systèmes à Composants Adaptables et Extensibles), October 17–18, 2002, Grenoble, France*, pages 184–195. ACM SIGOPS France, Oct. 2002. URL `http://asm.objectweb.org/current/asm-eng.pdf`. 21, 27

C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, London, 1859. 14

A. Elyasaf, M. Orlov, and M. Sipper. A heuristiclab evolutionary algorithm for FINCH. In C. Blum, E. Alba, T. Bartz-Beielstein, D. Loiacono, F. Luna, J. Mehnen, G. Ochoa, M. Preuss, E. Tantar, and L. Vanneschi, editors, *GECCO '13 Companion: Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, pages 1727–1728, Amsterdam, The Netherlands, 6-10 July 2013. ACM. doi:doi:10.1145/2464576.2480786. 55

J. Engel. *Programming for the Java™ Virtual Machine*. Addison-Wesley, Reading, MA, USA, July 1999. ISBN 0-201-30972-6. 4, 19

S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues. A genetic programming approach to automated software repair. In G. Raidl et al., editors, *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, July 8–12, 2009, Montréal Québec, Canada*, pages 947–954, New York, NY, USA, July 2009. ACM Press. ISBN 978-1-60558-325-9. doi:10.1145/1569901.1570031. 9, 53

J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification*. The Java™ Series. Addison-Wesley, Boston, MA, USA, third edition, May 2005. ISBN 0-321-24678-0. URL `http://java.sun.com/docs/books/jls`. 16, 44

B. Harvey, J. Foster, and D. Frincke. Towards byte code genetic programming. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, Orlando, Florida, USA, July 13–17, 1999*, volume 2, page 1234, San Francisco, CA, USA, Oct. 1999. Morgan Kaufmann. ISBN 1-55860-611-4. URL `http://www.csds.uidaho.edu/deb/ByteCode.pdf`. 8

L. Huelsbergen. Fast evolution of custom machine representations. In D. Corne, Z. Michalewicz, and B. McKay, editors, *The 2005 IEEE Congress on Evolutionary Computation, 2–5 September 2005, Edinburgh, Scotland, UK*, volume 1, pages 97–104. IEEE Press, Sept. 2005. ISBN 0-780-39363-5. doi:10.1109/CEC.2005.1554672. 9

S. Klahold, S. Frank, R. E. Keller, and W. Banzhaf. Exploring the possibilites and restrictions of genetic programming in Java bytecode. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference, Madison, Wisconsin, USA, July 22–25, 1998*, pages 120–124, Madison, WI, USA, July 1998. Omni Press. 8

T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–32, May 2008. doi:10.1145/1369396.1370017. 19

J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, MA, USA, Dec. 1992a. ISBN 0-262-11170-5. xi, 3, 28, 29, 30, 33, 34, 35, 37, 38, 39, 40, 41, 42, 57

J. R. Koza. A genetic approach to the truck backer upper problem and the intertwined spiral problem. In *IJCNN, International Joint Conference on Neural Networks, Baltimore, Maryland, USA, 7–11 June 1992*, volume 4, pages 310–318. IEEE Press, July 1992b. ISBN 0-7803-0559-0. doi:10.1109/IJCNN.1992.227324. 41

F. Kühling, K. Wolff, and P. Nordin. A brute-force approach to automatic induction of machine code on CISC architectures. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Genetic Programming: 5th European*

Conference, EuroGP 2002, Kinsale, Ireland, April 3–5, 2002, volume 2278 of *Lecture Notes in Computer Science*, pages 288–297, Berlin / Heidelberg, Apr. 2002. Springer-Verlag. ISBN 978-3-540-43378-1. doi:10.1007/3-540-45984-7_28. 8

W. B. Langdon and P. Nordin. Seeding genetic programming populations. In R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming: European Conference, EuroGP 2000, Edinburgh, Scotland, UK, April 15–16, 2000*, volume 1802 of *Lecture Notes in Computer Science*, pages 304–315, Berlin / Heidelberg, Apr. 2000. Springer-Verlag. ISBN 978-3-540-67339-2. doi:10.1007/b75085. 28

W. B. Langdon and R. Poli. The halting probability in von Neumann architectures. In P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, editors, *Genetic Programming: 9th European Conference, EuroGP 2006, Budapest, Hungary, April 10–12, 2006*, volume 3905 of *Lecture Notes in Computer Science*, pages 225–237, Berlin / Heidelberg, Apr. 2006. Springer. ISBN 978-3-540-33143-8. doi:10.1007/11729976_20. 17

C. Le Goues. *Automatic Program Repair Using Genetic Programming*. PhD thesis, University of Virginia, Charlottesville, VA, USA, May 2013. URL `https://www.cs.cmu.edu/~clegoues/docs/claire-dissertation.pdf`. 53

T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. The Java™ Series. Addison-Wesley, Boston, MA, USA, second edition, Apr. 1999. ISBN 0-201-43294-3. URL `http://java.sun.com/docs/books/jvms`. 4, 21, 23

S. Luke and L. Panait. Lexicographic parsimony pressure. In W. B. Langdon, E. Cantú-Paz, K. E. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. G. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, July 9–13, 2002*, pages 829–836, San Francisco, CA, USA, July 2002. Morgan Kaufmann. ISBN 1-55860-878-8. 32

S. Luke and L. Panait. A Java-based evolutionary computation research system, Mar. 2004. URL `http://cs.gmu.edu/~eclab/projects/ecj`. 9, 27

E. Lukschandl, M. Holmlund, E. Modén, M. Nordahl, and P. Nordin. Induction of Java bytecode with genetic programming. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference, Madison, Wisconsin, USA, July 22–25, 1998*, pages 135–142, Madison, WI, USA, July 1998. Omni Press. 7

E. Lukschandl, H. Borgvall, L. Nohle, M. Nordahl, and P. Nordin. Distributed Java bytecode genetic programming with telecom applications. In R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming: European Conference, EuroGP 2000, Edinburgh, Scotland, UK, April 15–16, 2000*, volume 1802 of *Lecture Notes in Computer Science*, pages 316–325, Berlin / Heidelberg, Apr. 2000. Springer-Verlag. ISBN 978-3-540-67339-2. doi:10.1007/b75085. 8

J. Miecznikowski and L. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In R. N. Horspool, editor, *Compiler Construction: 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8–12, 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127, Berlin / Heidelberg, Apr. 2002. Springer-Verlag. ISBN 978-3-540-43369-9. doi:10.1007/3-540-45937-5_10. 4

J. Mizoguchi, H. Hemmi, and K. Shimohara. Production genetic algorithms for automated hardware design through an evolutionary process. In Z. Michalewicz, J. D. Schaffer, H.-P. Schwefel, D. B. Fogel, and H. Kitano, editors, *Proceedings of the First IEEE Conference on Evolutionary Computation, ICEC '94, IEEE World Congress on Computational Intelligence, June 27–29, 1994, Orlando, Florida, USA*, volume 2, pages 661–664. IEEE Neural Networks, June 1994. ISBN 0-7803-1899-4. doi:10.1109/ICEC.1994.349980. 16, 17

D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, Summer 1995. doi:10.1162/evco.1995.3.2.199. 7

Y. Nakamura, K. Oguri, and A. Nagoya. Synthesis from pure behavioral descriptions. In R. Camposano and W. H. Wolf, editors, *High-Level VLSI Synthesis*, pages 205–229. Kluwer, Norwell, MA, USA, May 1991. ISBN 0-792-39159-4. URL http://www-lab09.kuee.kyoto-u.ac.jp/parthenon/NTT. 17

P. Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications.* Krehl Verlag, Münster, Germany, 1997. ISBN 3-931-54607-1. 3, 8

P. Nordin, W. Banzhaf, and F. D. Francone. Efficient evolution of machine code for CISC architectures using blocks and homologous crossover. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming*, volume 3, chapter 12, pages 275–299. The MIT Press, Cambridge, MA, USA, July 1999. ISBN 0-262-19423-6. 8

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.* Carnegie Mellon University, Pittsburgh, PA, USA, July 2006. ISBN 0-9786956-0-7. URL http://www.sei.cmu.edu/uls. 54

M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*, volume 4 of *Genetic Programming*. Kluwer, Norwell, MA, USA, May 2003. ISBN 1-4020-7444-1. 15, 42

M. Orlov and M. Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In G. Raidl et al., editors, *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, July 8–12, 2009, Montréal Québec, Canada*, pages 1043–1050, New York, NY, USA, July 2009. ACM Press. ISBN 978-1-60558-325-9. doi:10.1145/1569901.1570042. 2, 4, 13, 14, 27, H:1

M. Orlov and M. Sipper. FINCH: A system for evolving Java (bytecode). In R. Riolo, T. McConaghy, and E. Vladislavleva, editors, *Genetic Programming Theory and Practice VIII, GPTP-2010, May 20–22, Ann Arbor, Michigan, USA*, volume 8 of *Genetic and Evolutionary Computation*, chapter 1, pages 1–16. Springer, New York, Nov. 2010. ISBN 978-1-4419-7746-5. doi:10.1007/978-1-4419-7747-2_1. 2, H:1

M. Orlov and M. Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, Apr. 2011. doi:10.1109/TEVC.2010.2052622. 2, H:1

M. Orlov, C. Bregman, and M. Sipper. Automatic evolution of Java-written game heuristics. In M. B. Cohen and M. O. Cinnéide, editors, *Search Based Software Engineering: Proceedings of the Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10–12, 2011*, volume 6956 of *Lecture Notes in Computer Science*, page 277, Berlin Heidelberg, sep 2011. Springer-Verlag. ISBN 978-3-642-23715-7. doi:10.1007/978-3-642-23716-4_30. 2, 55, H:1

T. Perkis. Stack-based genetic programming. In Z. Michalewicz, J. D. Schaffer, H.-P. Schwefel, D. B. Fogel, and H. Kitano, editors, *Proceedings of the First IEEE Conference on Evolutionary Computation, ICEC '94, IEEE World Congress on Computational Intelligence, June 27–29, 1994, Orlando, Florida, USA*, volume 1, pages 148–153. IEEE Neural Networks, June 1994. ISBN 0-7803-1899-4. doi:10.1109/ICEC.1994.350025. 7, 9

R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, London, UK, Mar. 2008. ISBN 978-1-4092-0073-4. URL `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza). 3, 16, 28

M. D. Schmidt and H. Lipson. Incorporating expert knowledge in evolutionary search: A study of seeding methods. In G. Raidl et al., editors, *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, July 8–12, 2009, Montréal Québec, Canada*, pages 1091–1098, New York, NY, USA, July 2009. ACM Press. ISBN 978-1-60558-325-9. doi:10.1145/1569901.1570048. 28

E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS 2013, pages 317–328, Houston, Texas, USA, Mar. 16-20 2013. ACM. doi:doi:10.1145/2451116.2451151. URL `http://www.cs.virginia.edu/~weimer/p/schulte2013embedded.pdf`. 10

F. Servant, D. Robilliard, and C. Fonlupt. JEB: Java evolutionary byte-code — implementation and tests. In *Artificial Evolution, 7th International Conference, Evolution Artificielle, EA 2005, Lille, France, October 26–28, 2005*, Oct. 2005. URL `http://www-lil.univ-littoral.fr/~robillia/Publis/05_jeb.ps.gz`. 9

M. Sipper, Y. Azaria, A. Hauptman, and Y. Shichel. Designing an evolutionary strategizing machine for game playing and beyond. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 37(4):583–593, July 2007. 28

L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the Push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002. doi:10.1023/A:1014538503543. 9, 11

E. B. Tchernev. Forth crossover is not a macromutation? In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference, July 22–25, 1998, Madison, Wisconsin, USA*, pages 381–386, San Francisco, CA, USA, Aug. 1998. Morgan Kaufmann. ISBN 1-55860-548-7. URL `http://userpages.umbc.edu/~etcher1/gppaper/forcro.htm`. 8

E. B. Tchernev and D. S. Phatak. Control structures in linear and stack-based genetic programming. In M. Keijzer, editor, *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference, June 26–30, 2004, Seattle, Washington, USA*. Distributed on CD-ROM at GECCO-2004, June 2004. URL `http://www.cs.bham.ac.uk/~wbl/biblio/gecco2004/LBP041.pdf`. 8

H. Tuan-Hao, R. I. B. McKay, D. Essam, and N. X. Hoai. Solving symbolic regression problems using incremental evaluation in genetic programming. In *IEEE Congress on Evolutionary Computation, CEC 2006, Vancouver, British Columbia, Canada, July 16–21, 2006*, pages 2134–2141. IEEE Press, July 2006. ISBN 0-7803-9487-9. doi:10.1109/CEC.2006.1688570. 30, 31, 32

D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, Aug. 2011. ISSN 1089-778X. doi:doi:10.1109/TEVC.2010.2083669. 10

M. S. Withall, C. J. Hinde, and R. G. Stone. An improved representation for evolving programs. *Genetic Programming and Evolvable Machines*, 10(1):37–70, Mar. 2009. doi:10.1007/s10710-008-9069-7. 42, 43, 44

M. L. Wong and K. S. Leung. *Data Mining Using Grammar Based Genetic Programming and Applications*, volume 3 of *Genetic Programming*. Kluwer, Norwell, MA, USA, Feb. 2000. ISBN 978-0-7923-7746-7. doi:10.1007/b116131. 16

J. R. Woodward. Evolving Turing complete representations. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *The 2003 Congress on Evolutionary Computation, CEC 2003, Canberra, Australia, 8–12 December, 2003*, volume 2, pages 830–837. IEEE Press, Dec. 2003. ISBN 0-7803-7804-0. doi:10.1109/CEC.2003.1299753. 4

Index

A
AIM-GP, 8
array sum, 41
artificial ant, 33
    Santa Fe trail, 33

B
BNF, 16
bug fixing, *see* software repair
bytecode, 4
    correctness, 11
    decompilation, 30
    evolution, 11
    instrumentation, 18
    optimization, 18
    requirements, 21

C
compiler optimization, 18

D
DEVTAG, 30
DJBGP, 8

E
ECJ, 9, 27
extant software, 3

F
factorial, 21
FINCH, 1, 4
    compatible crossover, 14, 22
    forward control flow, 33, 37
    looping constructs, 41
    recursion, 45
Forth, 8

G
genetic programming, 3
    linear GP, 3
    Stack GP, 7
    tree GP, 3
grammatical evolution, 3, 10, 15

I
incremental evaluation, 30
intertwined spirals, 37

J
JAFF, 17
JBGP, 7
JEB, 9
just-in-time compilation, 4, 19
JVM, 11, 19
    languages, 4
    software lifecycle, 11
    verification, 11

M
machine code evolution, 3, 8
mutation, 15

P
Push, 9

R
representation, 3, 7

S
software improvement, 46
software repair, 9, 46
symbolic regression, 28
    complex symbolic regression, 30

T
termination, 17
tic-tac-toe, 46

Y
yield, 28

Glossary

BNF Backus-Naur Form. 15

CFG Context-Free Grammar. 16

GP Genetic Programming. 3

JVM Java Virtual Machine. 1, 4

הנדסת תוכנה דארוויניסטית

מחקר לשם מילוי חלקי של הדרישות
לקבלת תואר
דוקטור לפילוסופיה

מאת

מיכאל אורלוב

הוגש לסינאט

אוניברסיטת
בן-גוריון בנגב

ספטמבר 2013 תשרי תשע״ד

באר־שבע

הנדסת תוכנה דארוויניסטית

מחקר לשם מילוי חלקי של הדרישות
לקבלת תואר
דוקטור לפילוסופיה

מאת

מיכאל אורלוב

הוגש לסינאט

אוניברסיטת
בן-גוריון בנגב

אישור המנחה ______________________________

אישור דיקן בית הספר ללימודי
מחקר מתקדמים ע״ש קרייטמן ______________________________

ספטמבר 2013 תשרי תשע״ד

באר־שבע

העבודה נעשתה בהדרכת **פרופ׳ משה זיפר**
במחלקה למדעי המחשב
בפקולטה למדעי הטבע

תקציר

תזה זאת מציגה את FINCH, מתודולוגיה לאבולוציה של בייטקוד של Java, המאפשרת אבולוציה של תוכניות Java *קיימות ולא מוגבלות*, או תוכניות בשפות אחרות אשר מקומפלות לבייטקוד של Java. שם התזה, "הנדסת תוכנה דארוויניסטית", משקף את הראיה האופטימיסטית שלנו כיצד מתודולוגיה זאת יכולה להשפיע על יישום של מחשוב אבולוציוני בתחום הנדסת תוכנה.

מערכת מחשוב אבולוציוני דורשת את תכונות המפתח של בחירה, הרכבה והערכה של פריטים על גבי דורות. התכונה המורכבת ביותר, ברגע שאנו מנסים ליישם תכונות אלו על תוכניות הכתובות בשפה קיימת וכללית, היא הרכבה (רקומבינציה). הגישה שלנו מבוססת על המושג של *הכלאה תואמת*, המייצרת תוכניות תקינות ע"י ביצוע בדיקות אופרנדים מבוססות מחסנית, משתנים מקומיים, וזרימת בקרה על קטעי בייטקוד מקור ויעד.

עד כמה שידוע לנו, הגישה הזאת לאבולוציה של תוכנה קיימת הינה ייחודית. בתחילת העבודה, אנו משווים את גישתנו מול עבודות קיימות המשתמשות בחלק מצומצם מאוסף פקודות הבייטקוד של Java בתור שפת ייצוג עבור פריטים בתכנות גנטי.

לאחר מכן, אנחנו מתארים מימוש של FINCH, עם התמקדות באלגוריתמים המשיגים הכלאה תואמת עבור יצירת פריטים תקינים. אנו דנים בהתכנות של מימושים חילופיים, ומתארים כיצד מטופלים מכשולים כגון אי-עצירה.

אחרי כן, אנו ממחישים את ההצלחה המסחררת של FINCH בפתרון אוסף של בעיות, כולל רגרסיה פשוטה ומורכבת, ניווט מסלול, סיווג תמונה, סכום מערך, ואיקס מיקס דריקס. FINCH מנצלת את עושר ארכיטקטורת JVM ומערכת הטיפוסים שלה על מנת לייצר פתרונות בצורה של תוכניות Java המובנות לבני אדם.

אנו מקווים שהיכולת לפתח תוכניות Java בצורה אבולוציונית תוביל לכלי חדש ורב ערך באוסף הכלים של מהנדסי תוכנה.

העבודה המתוארת כאן פורסמה בעבר ב-
.[Orlov and Sipper, 2009, 2010, 2011, Orlov et al., 2011]

קטגוריות ומתארי נושא

I.2.2 [**בינה מלאכותית**]: תכנות אוטומטי—טרנספורמציה של תוכנית, שינוי תוכנית;
D.3.3 [**שפות תכנות**]: מבנים ותכונות של שפה; D.2.2 [**הנדסת תוכנה**]: כלים ושיטות עיצוב.[1]

מילות מפתח

אבולוציה של תוכנה, תכנות גנטי, חישוב אבולוציוני, בייטקוד של ג׳אווה.

[1] לפי מערכת הסיווג של ACM Computing: http://www.acm.org/about/class/1998