A HeuristicLab Evolutionary Algorithm for FINCH

Achiya Elyasaf
Ben-Gurion University of the Negev, Be'er Sheva, Israel
achiya.e@gmail.com

Michael Orlov
Ben-Gurion University of the Negev, Be'er Sheva, Israel
orlovm@cs.bgu.ac.il

Moshe Sipper
Ben-Gurion University of the Negev, Be'er Sheva, Israel
sipper@cs.bgu.ac.il

ABSTRACT

We present a HeuristicLab plugin for FINCH.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.3 [**Programming Languages**]: Language Constructs and Features; I.2.2 [**Artificial Intelligence**]: Automatic Programming-program transformation, program modification

General Terms

Algorithms, Languages

Keywords

Java bytecode, Software Evolution, HeuristicLab

1. FINCH

FINCH (Fertile Darwinian Bytecode Harvester) is a system designed to evolutionarily improve actual *extant* software, which was *not intentionally written* for the purpose of serving as a GP representation in particular, nor for evolution in general. The only requirement is that the software source code be either written in Java or can be compiled to Java bytecode. The following chapter provides an overview of FINCH, ending with a précis of results. Additional information can be found in [6, 7].

Java compilers typically do not produce machine code directly, but instead compile source-code files to platform-independent *bytecode*, to be interpreted in software or, rarely, to be executed in hardware by a *Java Virtual Machine (JVM)* [4]. The *JVM* is free to apply its own optimization techniques, such as Just-in-Time (JIT) on-demand compilation *Java compilation* to native machine code—a process that is transparent to the user. The *JVM* implements a stack-based architecture with high-level language features such as object management and garbage collection, virtual function calls, and strong typing. The bytecode language itself is a well-designed assembly-like language with a limited yet powerful instruction set [3, 4]. Figure 1 shows a recursive Java program for computing the *factorial* of a number, and its corresponding bytecode.

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.
ACM 978-1-4503-1964-5/13/07.


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

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

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

(a)

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

(b)

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

The *JVM* architecture is successful enough that several programming languages compile directly to Java bytecode (e.g., Scala, Groovy, Jython, Kawa, JavaFX Script, and Clojure). Moreover, Java *decompilers* are available, which facilitate restoration of the Java source code from compiled bytecode. Since the design of the *JVM* is closely tied to the design of the Java programming language, such *decompilation* often produces code that is very similar to the original source code [5].

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

Note that we do not wish to invent a language to improve upon some aspect or other of GP (efficiency, terseness, readability, etc.), as has been amply done. Nor do we wish to extend standard GP to become Turing complete, an issue which has also been addressed [9]. 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.

Currently, FINCH uses ASM [1] and ECJ evolutionary framework [2], with ECJ providing the evolutionary engine. The configuration of ECJ as well as FINCH is done by multiple hierarchical parameter files that are used to define the evolutionary algorithm parameters (e.g., number of generations, mutation and crossover probabilities) and the bytecode parameters (e.g., the bytecode seed and the fitness evaluator). To the non-expert user, executing a simple experiment in FINCH or even in ECJ is a non-trivial task. We wish to simplify the usage of FINCH by adding a GUI for FINCH.

Even though ECJ version 2.0 includes a simple GUI, it still requires handling several parameter files. Thus we turn to HeuristicLab.

2. HEURISTICLAB

HeuristicLab [8] is a GUI framework for heuristic and evolutionary algorithms. HeuristicLab provides a feature-rich software environment for heuristic optimization researchers and practitioners. It is based on a generic and flexible model layer and offers a graphical algorithm designer that enables the user to create, apply, and analyze heuristic optimization methods. A powerful experimenter allows HeuristicLab users to design and perform parameter tests even in parallel. The results of these tests can be stored and analyzed easily in several configurable charts. HeuristicLab is available under the GPL license.

We present here a preliminary work on a HeuristicLab evolutionary algorithm for FINCH. Using HeuristicLab along with FINCH will simplify the learning process. The crossover, mutation and fitness evaluation will be done by FINCH, while all of the parameters handling as well as executing the experiments and analyzing the results will be done directly from HeuristicLab, thus excising the use of ECJ.

3. A SUMMARY OF RESULTS

In this section we present some results of FINCH. Due to space limitations we only provide a brief description of our results, with the full account available in [6, 7]. To date, we have successfully tackled several problems:

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

Acknowledgments

Achiya Elyasaf is partially supported by the Lynn and William Frankel Center for Computer Sciences. This research was supported by the Israel Science Foundation (grant no. 123/11).

4. REFERENCES

[1] 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, Oct. 2002.

[2] ECLab Evolutionary Computation Laboratory, George Mason University. ECJ 2.0. `http://cs.gmu.edu/~eclab/projects/ecj/`, 2010.

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

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

[5] 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.

[6] 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.

[7] M. Orlov and M. Sipper. Flight of the finch through the java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, 2011.

[8] S. Wagner. *Heuristic Optimization Software Systems - Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment*. PhD thesis, Johannes Kepler University, Linz, Austria, 2009.

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