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.

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.

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.

Copyright is held by the author/owner(s).
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 exciting 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 $x^3 + x^3 + x^2 + x$ 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