Evolving Software Building Blocks with FINCH

Michael Orlov
Shamoon College of Engineering
Beer Sheva, Israel
orlovm@noexec.org

ABSTRACT

This paper proposes to explore the following question: can software evolution systems like FINCH, that evolve linear representations originating from a higher-level structural language, take advantage of building blocks inherent to that original language?

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**; **Genetic programming**; Control structures;

KEYWORDS

Genetic improvement, Java bytecode, genetic programming

ACM Reference format:
Michael Orlov. 2017. Evolving Software Building Blocks with FINCH. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15–19, 2017,* 2 pages.
https://doi.org/10.1145/3067695.3082521

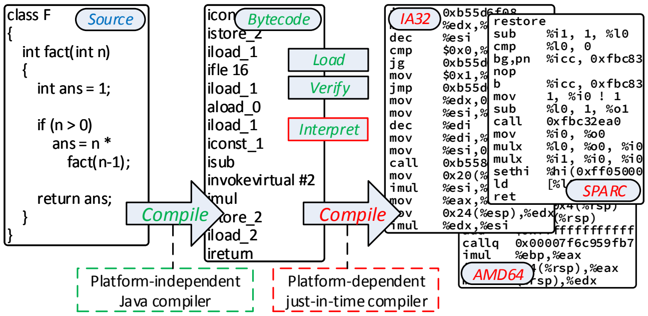
1 POSITION

FINCH [4, 5] is a system for unrestricted Darwinian software evolution that is based on Java bytecode. Any Java source code can be automatically evolved after compiling it to Java bytecode, when provided with a fitness function. The reason for working at bytecode level instead of source code level is the conceptual simplicity of detecting correct crossovers in Java bytecode methods. The main benefit is efficiency: both costly source code (mostly failed) compilation attempts, and Java virtual machine verification of (mostly incorrect) bytecode modifications are altogether avoided. Correct bytecode crossover thus drives unrestricted evolution of Java software, and also software written in one of the many other languages that can use the JVM backend (Scala, Python, Ruby and others).

We therefore opt to use linear genetic programming [1] on Java bytecode not due to a preference of linear vs. tree-based GP, but mainly because the methodology produces a search space that roughly corresponds to all correct bytecode sequences that might result from crossover-based evolution of initially seeded population of programs. Bytecode formal semantics are simple enough to efficiently construct such a search space.

GECCO '17 Companion, July 15–19, 2017, Berlin, Germany

ACM ISBN 978-1-4503-4939-0/17/07…$15.00
https://doi.org/10.1145/3067695.3082521



```
class F
{
   int fact(int n)
   {
      int ans = 1;

      if (n > 0)
         ans = n *
            fact(n-1);

      return ans;
   }
}
```

```
icon
istore_2
iload_1
ifle 16
iload_1
aload_0
iload_1
iconst_1
isub
invokevirtual #2
imul
tore_2
iload_2
iretum
```

```
        0xb55d6f08
        %edx,%
dec     %esi
cmp     $0x0,%
jg      0xb55d
mov     $0x1,%
jmp     0xb55d
mov     %edx,0
mov     %esi,%
dec     %edi
mov     %edi,%
mov     %esi,0
call    0xb558
mov     0x20(%
imul    %esi,%
mov     %eax,%
ov      0x24(%esp),%edx
imul    %edx,%esi
```

```
restore
sub     %i1, 1, %l0
cmp     %l0, 0
bg,pn   %icc, 0xfbc83
nop
b       %icc, 0xfbc83
mov     1, %i0 ! 1
sub     %l0, 1, %o1
call    0xfbc32ea0
mov     %i0, %o0
mulx    %l0, %o0, %i0
mulx    %i1, %i0, %i0
sethi   %hi(0xff05000
ld      [%l
ret
```

```
        x4(%rsp)
        (%rsp)
        ffffffffff
callq   0x00007f6c959fb7
imul    %ebp,%eax
        4(%rsp),%eax
        rsp),%edx
```

Source
Bytecode
Load
Verify
Interpret
Compile
Compile
IA32
SPARC
AMD64
Platform-independent Java compiler
Platform-dependent just-in-time compiler

Figure 1: Java source code is first compiled to bytecode, and subsequently interpreted or executed as native code. Heavy optimizations are reserved for the JIT-compilation phase.

However, bytecode is still not the original source code language, and the completeness of bytecode search space is vast. When our motivation for improving extant software comes from domain knowledge that is already present in the code, we must find a way to use preexisting building blocks — as they are seen in the eyes of the original software designer. Even when our purpose is to evolve software from scratch, we still work with programming languages that were developed for humans, and that assume certain structure at high-level source code constructs. Researchers are certainly interested in evolving software at source code level [2]. Perhaps we could provide similar capacity while still taking advantage of bytecode-level improvements?

The fundamental question posited is thus: *How is it possible to reintroduce the structural benefits of tree-based GP into its compiled and serialized linear version?* More specifically, how to achieve that when the compiled result is Java bytecode? We attempt to consider cost-effective automatic solutions that are easy to integrate, as opposed to a more involved guidance of genetic improvement process, such as proposed by White [6].

2 BYTECODE EVOLUTION BACKGROUND

Java bytecode results from compilation of source code in Java or in another JVM-compatible language. It has a simple yet complete structure, targeted at static correctness verification and later interpretation with optimized just-in-time compilation to native code by the JVM. Figure 1 illustrates this code execution pipeline.

Java bytecode operates over a single stack and a local variables array, which differ for each method invocation frame. Both structures contain statically typed values, allowing for static bytecode analysis by the JVM. Bytecode evolution in FINCH also performs similar static analysis, and locates bytecode sequences that are fit for compatible crossover, consistent with primitive and object value type hierarchies in both the stack and in the variables array.

Static bytecode analysis is not limited to detecting crossover correctness. Below, we discuss the possibility of employing static analysis in order to detect building blocks that are comparable to those naturally found in the original source code, and use them to construct a tree-based GP data structure.

Exploring the question presented in Section 1 does not require knowledge of Java bytecode intricacies, although the interested reader may refer to literature on the subject [3]. The bottom line is that bytecode is a sequence of assembly-like instructions that operate on an abstract model of a typed stack machine that also supports typed registers — an array containing arguments and local variables. So, for instance, the bytecode sequence *iload_1* / *iconst_1* / *isub* in Figure 1 pushes variable number 1 of type *int* onto the stack (the variable *n*, as variable number 0 is the object containing the method), pushes *int* constant equal to 1 onto the stack, and then subtracts the two values, popping them from the stack and pushing back the result $n - 1$.

3 HIGH-LEVEL BUILDING BLOCKS

The following discussion's scope is limited to building blocks that can be recovered from a single method or function, since methods and classes are fundamental entities in Java code. Let us consider a single unrestricted Java method that FINCH is able to handle: it contains expressions, object creation, conditionals, loops, method calls, exceptions throwing and handling blocks, returns from a method, etc. What kind of high-level building blocks, then, can we hope to detect at bytecode level? How is it possible to algorithmically characterize the linear sequences of bytecode that correspond to these blocks? Hopefully, resorting to context-free grammar approximations of high-level language syntax can be avoided.

Once we look at the building blocks mentioned above, it becomes apparent that they may be roughly partitioned into the following three categories: expressions, statements, and control flow exits.

Expressions are all rvalue expressions and sub-expressions, including single variable references, arithmetic and Boolean expressions, non-*void* method and function invocations, conditions in *if-then-else* statements and in loops, *if-then-else* expressions (those that use *?:* syntax), and so on. In the statement $ans = \underline{\underline{n} * \underline{fact(\underline{\underline{n}-\underline{1}})}}$, all underlined parts are expressions. We expect expressions to represent the most commonly encountered type of building blocks, structured recursively. Indeed, many tree-based GP systems use expressions exclusively. Note that although expression's internal structure may be ambiguous to a human, it is deterministic in source code. E.g., *a+b+c* always represents *(a+b)+c* and not *a+(b+c)*.

Statements are building blocks that do not produce a value: assignments, *void* method and function invocations, *then* and *else* parts of conditionals, complete *if-then-else* statements, *while* and *do-while* loops and their bodies, arbitrary scoped blocks, and, importantly, sequences of statements. Partitioning of sequence of statements into sub-sequences is ambiguous — however, treating this issue may remain at the discretion of tree-based GP variation operators designer. One possibility is to maximize the length of higher-level sequences, avoiding sub-sequences altogether.

Finally, control flow exits are the *return* and *throw* statements that abandon control flow, and are otherwise similar to regular statements discussed above. We also omit discussion of special control flow statements like *break* and *continue*, which violate assumptions about high-level building blocks in structured programming.

4 RECOVERING BUILDING BLOCKS

Which bytecode principles discussed in Section 2 may be useful for recovering the original source code building blocks, without having to support the full high-level language syntax? Local variables array and class fields are unlikely to prove useful, since these features cross the scopes of building blocks discussed in Section 3. What seems most fit for the purpose is the stack, the state of which closely corresponds to program's control flow during method or function invocation. In order to unmask the building blocks, we might be able to employ static analysis of evolving method's stack, which is already performed for the purpose of compatible crossover in FINCH.

Statement building blocks are the easiest to detect, as they exhibit neutrality with regards to stack state. For instance, the previously discussed statement *ans = n * fact(n-1)* manipulates stack state above the top position before the statement is executed. However, after the assignment to local variable *ans* is completed, all the extra stack values are gone.

Expressions are quite similar to statements, except that they add exactly one value to the stack after temporarily manipulating its state above the previous top position. This occurs, for instance, with the *n-1* example from Section 2.

Once linear Java bytecode representation is converted to a tree-based one, we may take advantage of our favorite tree GP methods, such as biasing variation operators toward subtree features like height and type, biasing evolution toward preferred tree size and height, and so forth.

Since the high-level building blocks are not organic to the evolving individuals, but are instead reconstructed again and again from linear representation, we might encounter unorthodox behaviors during the evolutionary process. For instance, building blocks might deteriorate or disappear altogether, despite the related bytecode instructions still remaining intact, except for some control flow changes. After all, evolutionary operators still work at linear GP level and not on tree structures. Certainly, this is a subject that requires separate examination.

REFERENCES

[1] Markus F. Brameier and Wolfgang Banzhaf. 2007. *Linear Genetic Programming.* Springer, New York, NY, USA. https://doi.org/10.1007/978-0-387-31030-5

[2] Brendan Cody-Kenny, Edgar Galván López, and Stephen Barrett. 2015. locoGP: Improving Performance by Genetic Programming Java Source Code. In *Genetic Improvement 2015 Workshop,* William B. Langdon, Justyna Petke, and David R. White (Eds.). ACM, Madrid, Spain, 811–818. https://doi.org/10.1145/2739482.2768419

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

[4] Michael Orlov and Moshe Sipper. 2009. Genetic Programming in the Wild: Evolving Unrestricted Bytecode. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation,* Günther Raidl and others (Eds.). ACM, Montréal Québec, Canada, 1043–1050. https://doi.org/10.1145/1569901.1570042

[5] Michael Orlov and Moshe Sipper. 2011. Flight of the FINCH through the Java Wilderness. *IEEE Transactions on Evolutionary Computation* 15, 2 (April 2011), 166–182. https://doi.org/10.1109/TEVC.2010.2052622

[6] David R. White. 2016. Guiding Unconstrained Genetic Improvement. In *Genetic Improvement 2016 Workshop,* Justyna Petke, Westley Weimer, and David R. White (Eds.). ACM, Denver, CO, USA, 1133–1134. https://doi.org/10.1145/2908961.2931688