Evolving Software Building Blocks with FINCH

Michael Orlov, SCE, Israel

GECCO / Genetic Improvement 2017, July 16

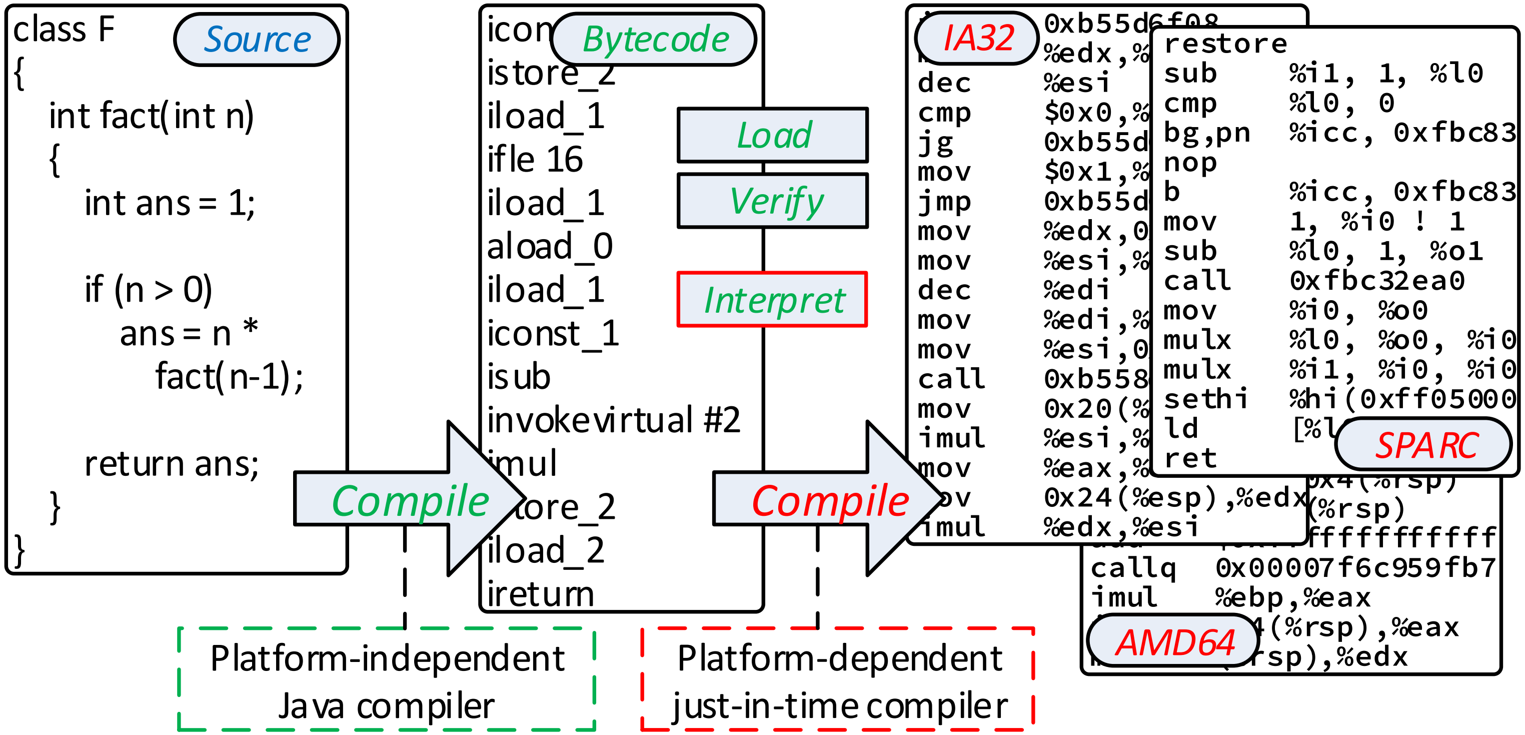
Proposed Exploration Subject

Can software evolution systems that evolve **linear representations** originating from a higher-level structural language, take advantage of **building blocks** inherent to that original language?

Why Linear GP?

- Why use linear representation in the first place?
- **Preference** of linear vs. tree GP is irrelevant
- Our methodology produces **search space** of correct bytecode sequences resulting from crossover-based evolution

FINCH Background



```
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
ireturn
```

```
        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
```

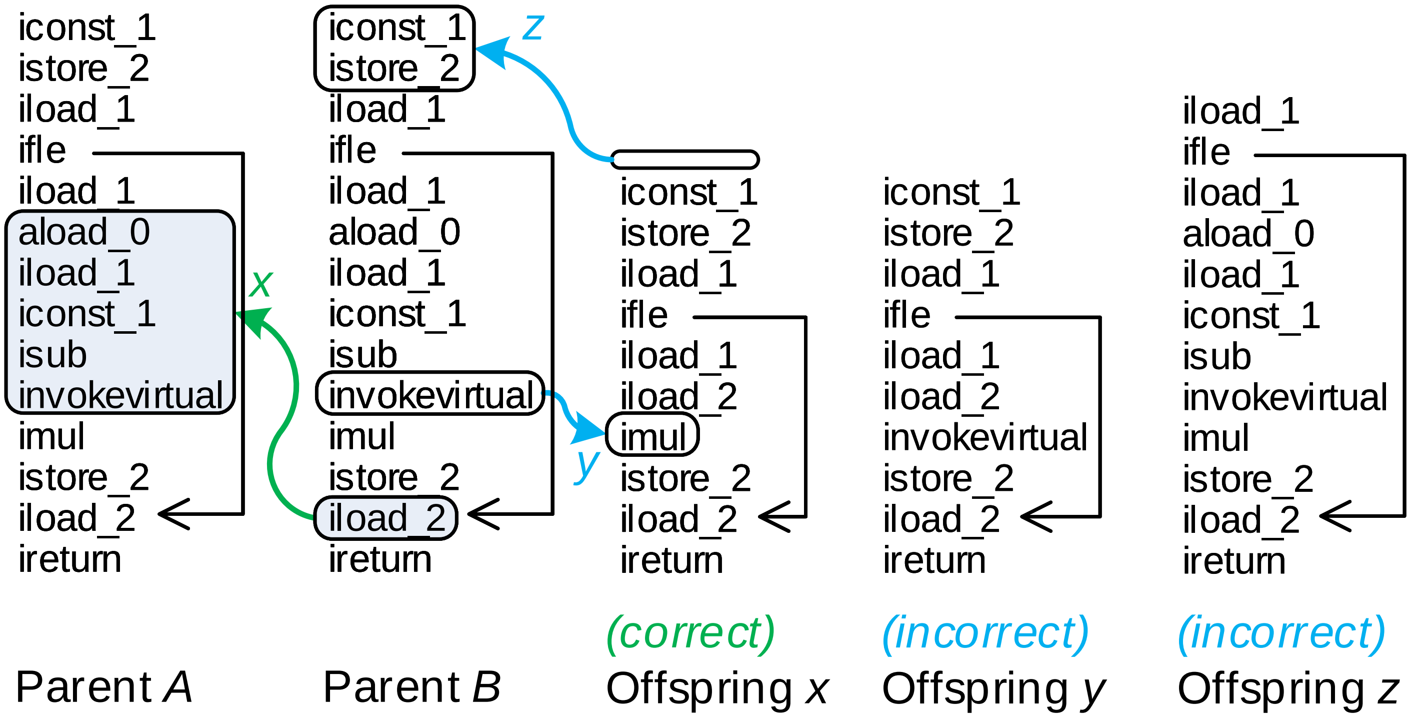
```
restore
sub     %i1, 1, %l0
cmp     %l0, 0
bg,pn   %icc, 0xfbc83
nop
b       %icc, 0xfbc83
mov     1, %i0 ! 1
sub     %l0, 1, %o1
call    0xfbc32ea0
mov     %i0, %o0
mulx    %l0, %o0, %i0
mulx    %i1, %i0, %i0
sethi   %hi(0xff05000
ld      [%l
ret
```

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

Platform-independent
Java compiler

Platform-dependent
just-in-time compiler

FINCH Background (contd.)



```
iconst_1
istore_2
iload_1
ifle
iload_1
aload_0
iload_1
iconst_1
isub
invokevirtual
imul
istore_2
iload_2
ireturn
```

Parent *A*

```
iconst_1
istore_2
iload_1
ifle
iload_1
aload_0
iload_1
iconst_1
isub
invokevirtual
imul
istore_2
iload_2
ireturn
```

Parent *B*

```
iconst_1
istore_2
iload_1
ifle
iload_1
iload_2
imul
istore_2
iload_2
ireturn
```

(correct)
Offspring *x*

```
iconst_1
istore_2
iload_1
ifle
iload_1
iload_2
invokevirtual
istore_2
iload_2
ireturn
```

(incorrect)
Offspring *y*

```
iload_1
ifle
iload_1
aload_0
iload_1
iconst_1
isub
invokevirtual
imul
istore_2
iload_2
ireturn
```

(incorrect)
Offspring *z*

Why NOT Linear GP?

- Why do we use GP with **real-world programming languages** to begin with?
  - Structural building blocks are inherent to programming languages
- Naïve linear GP has no concept of **high-level building blocks**
- Random correct crossovers are strongly biased towards small bytecode sections

Static Bytecode Analysis

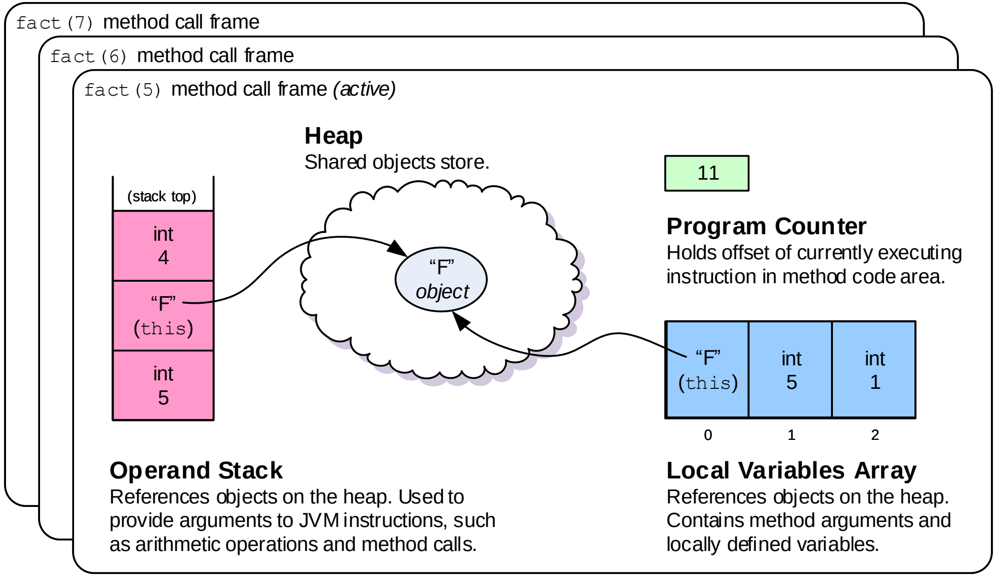
- Currently employed for detecting crossover correctness
- Is generalizable to all kinds of **static information** available to JVM verifier:
  - Deduced value types inside operand stack
  - Deduced value types inside local variables array
  - Aggregated operations on stack and local variables by **sections of bytecode**

Which Building Blocks to Detect?

- **Expressions**
  - ans=n*fact(n-1) → also: n, n, 1, n-1, fact(n-1)
  - x=(y>0)?a:(b-3) → also: y, 0, y>0, a, b, 3, b-3
- **Statements**
  - ans=n*fact(n-1)
  - if (x>0) then S.o.p(x); else return -1; → also: S.o.p(x)
- **Control flow exits**
  - return -1, throw new RuntimeException()
  - break et al. probably shouldn’t be handled (violate assumptions)

Building Blocks Recovery – How?

- Local variables?
  - Cross building blocks scope
- Class fields?
  - Same as above…
- Stack?
  - Closely corresponds to program's control flow!



fact (7) method call frame
fact (6) method call frame
fact (5) method call frame *(active)*
Heap
Shared objects store.
(stack top)
int
4
"F"
(this)
int
5
"F"
object
11
Program Counter
Holds offset of currently executing instruction in method code area.
"F"
(this)
int
5
int
1
0
1
2
Operand Stack
References objects on the heap. Used to provide arguments to JVM instructions, such as arithmetic operations and method calls.
Local Variables Array
References objects on the heap. Contains method arguments and locally defined variables.

Recovery via bytecode: **Statements**

- Exhibit neutrality wrt. stack state
- Consider the previously mentioned statement:
  - ans = n * fact(n-1)
- Stack state below top position is untouched
- After assignment is completed, all extra stack values are gone

Recovery via bytecode: **Expressions**

- Add exactly one value to the stack
- Consider the following expression:

```
n * fact(n-1)
```

- Stack state below top position is untouched
- After value is computed, it is placed above previous stack top

Post-recovery – What's next?

- Full **tree GP ecosystem** is now at our disposal
- Can bias variation operators towards subtree features (height, type)
- Important: ultimately, tree GP variation operators still produce linear **bytecode sections**, which are passed on to FINCH

Problems to Watch For

- High-level building blocks are not organic to the evolving individuals
- Building blocks are reconstructed from linear representation for each individual
- Unorthodox behavior during evolution?
  - Requires experimental examination