Genetic Programming in the Wild: Evolving Unrestricted Bytecode

Michael Orlov and Moshe Sipper
orlovm, sipper@cs.bgu.ac.il

Department of Computer Science
Ben-Gurion University, Israel

GECCO 2009, July 8–12
Montréal, Québec, Canada

GP: Programs or Representations?

> "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."

A Field Guide to Genetic Programming
[Poli, Langdon, and McPhee, 2008]

GP: Programs or Representations?

> "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**."

A Field Guide to Genetic Programming
[Poli, Langdon, and McPhee, 2008]

Our Goals

From programs. . .

Evolve actual programs

written in Java

. . . to software!

Improve (existing) software

written in unrestricted Java

Our Goals

From programs. . .

Evolve actual programs

written in Java

. . . to software!

Improve (existing) software

written in unrestricted Java

Extending prior work

Existing work uses **restricted subsets** of Java bytecode as **representation language** for GP individuals

We evolve **unrestricted bytecode**

Let's Evolve Java Source Code

- Rely on the building blocks in the initial population
- Defining **genetic operators** is problematic
- How to define **good** source code crossover?

Factorial *(recursive)*

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

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

    return ans;
  }
}
```

⇐

Factorial *(iterative)*

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

    for (;  n > 0;  n--)
      ans = ans * n;


    return ans;
  }
}
```

“Stupid” Example

- **Source-level crossover** typically produces garbage

Factorial *(recursive $\overset{\leftarrow}{\times}$ iterative)*

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

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

    return ans;
  }
}
```

“Stupid” Example

- **Source-level crossover** typically produces garbage

Factorial *(recursive* $\overleftarrow{\times}$ *iterative)*

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

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

    return ans;
  }
}
```

Parse Trees

- Maybe we can design **better** genetic operators?

Parse Trees

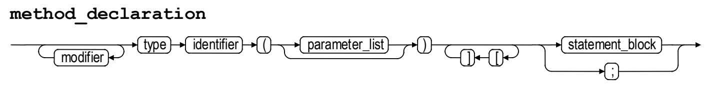
- Maybe we can design **better** genetic operators?
- Maybe. . . but too much harsh **syntax**
  Possibly use **parse tree**?

Parse Trees

- Maybe we can design **better** genetic operators?
- Maybe. . . but too much harsh **syntax**
  Possibly use **parse tree**?

Just one BNF rule *(of many)*

`method_declaration` $::\Longrightarrow$
  `modifier`* `type identifier`
  "(" `parameter_list`$^?$ ")" "[ ]"*
  $\langle$ `statement_block` $|$ ";" $\rangle$



Bytecode

Better than parse trees:

Let's use **bytecode**!

Java Virtual Machine *(JVM)*

- Source code is compiled to **platform-neutral bytecode**
- Bytecode is executed with **fast** just-in-time compiler
- High-order, **simple** yet powerful architecture
- **Stack-based**, supports hierarchical object **types**
- Not limited to Java!

  (Scala, Groovy, Jython, Kawa, Clojure, . . .)

Bytecode (cont'd)

Some basic bytecode instructions

Stack ↔ Local variables

iconst 1	pushes `int` 1 onto operand stack
aload 5	pushes **object** in local variable 5 onto stack *(object **type** is deduced when class is loaded)*
dstore 6	pops two-word `double` to local variables 6–7

Bytecode (cont'd)
Some basic bytecode instructions

Stack ↔ Local variables

iconst 1	pushes `int` 1 onto operand stack
aload 5	pushes **object** in local variable 5 onto stack *(object **type** is deduced when class is loaded)*
dstore 6	pops two-word `double` to local variables 6–7

Arithmetic instructions *(affect operand stack)*

imul	pops two `int`s from stack, pushes multiplication result

Bytecode (cont'd)

Some basic bytecode instructions

Stack ↔ Local variables

iconst 1	pushes `int` 1 onto operand stack
aload 5	pushes **object** in local variable 5 onto stack *(object **type** is deduced when class is loaded)*
dstore 6	pops two-word `double` to local variables 6–7

Arithmetic instructions *(affect operand stack)*

imul	pops two `int`s from stack, pushes multiplication result

Control flow *(uses operand stack)*

ifle +13	pops `int`, jumps +13 bytes if value $\leqslant 0$
lreturn	pops two-word `long`, returns to caller's stack

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

Correct bytecode requirements

Stack use is **type-consistent**

(e.g., can't multiply an `int` *by an* **Object***)*

Local variables use is **type-consistent**

(e.g., can't read an `int` *after storing an* **Object***)*

No stack underflow

No reading from uninitialized variables

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

Correct bytecode requirements

Stack use is **type-consistent**

(e.g., can't multiply an `int` by an **Object***)*

Local variables use is **type-consistent**

(e.g., can't read an `int` after storing an **Object***)*

No stack underflow

No reading from uninitialized variables

- So, genetic operators are still delicate

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

Correct bytecode requirements

Stack use is **type-consistent**
(e.g., can't multiply an `int` *by an* **Object***)*
Local variables use is **type-consistent**
(e.g., can't read an `int` *after storing an* **Object***)*
No stack underflow
No reading from uninitialized variables

- So, genetic operators are still delicate
- Need **good** genetic operators to produce **correct** offspring

Bytecode (cont'd)
Evolutionary operators

- Java bytecode is **less fragile** than source code
- But, bytecode must be **correct** in order to run **correctly**

Correct bytecode requirements

Stack use is **type-consistent**
(e.g., can't multiply an `int` *by an* **Object***)*
Local variables use is **type-consistent**
(e.g., can't read an `int` *after storing an* **Object***)*
No stack underflow
No reading from uninitialized variables

- So, genetic operators are still delicate
- Need **good** genetic operators to produce **correct** offspring
- Conclusion: Avoid **bad** crossover and mutation

Evolutionary Operators

Unidirectional bytecode crossover:

Bytecode A

...

...

...

Section α

...

...

...

Bytecode B

...

...

...

Section β

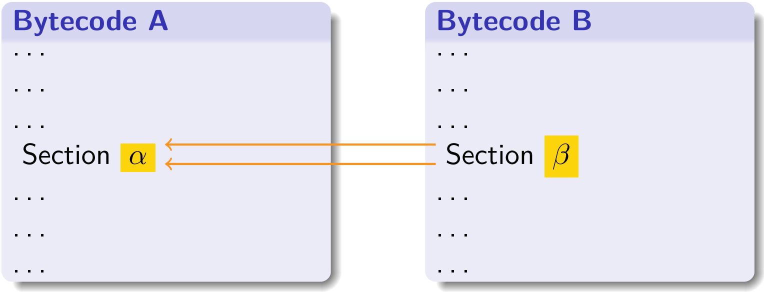
...

...

...

Evolutionary Operators

Unidirectional bytecode crossover:



Bytecode A

. . .

. . .

. . .

Section α

. . .

. . .

. . .

Bytecode B

. . .

. . .

. . .

Section β

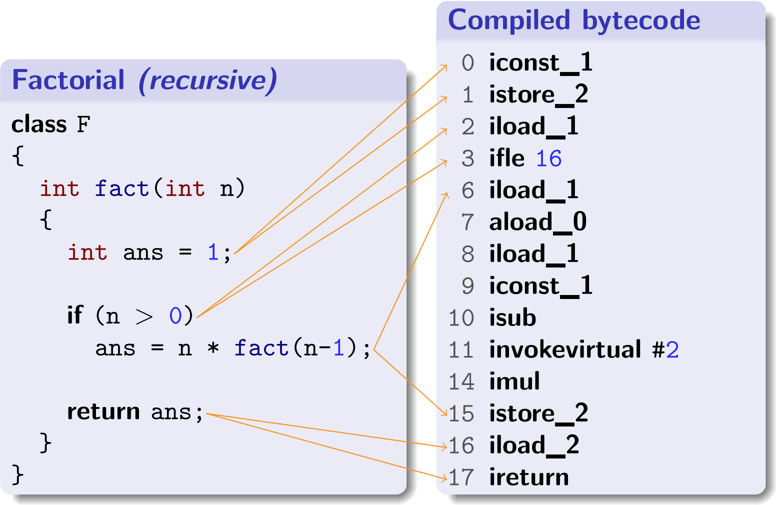
. . .

. . .

. . .

Evolutionary Operators
Good and bad crossovers

Parent **A**:



Factorial *(recursive)*

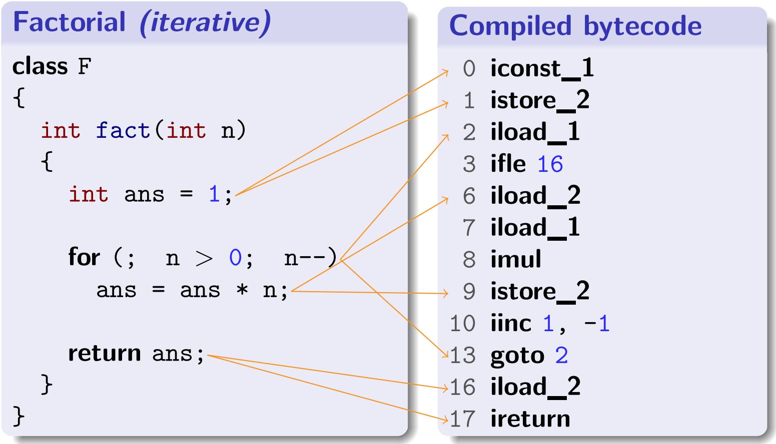
```
class F
{
  int fact(int n)
  {
    int ans = 1;

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

    return ans;
  }
}
```

Compiled bytecode

```
 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
```

Evolutionary Operators
Good *and* *bad* *crossovers*

Parent **B**:



Factorial *(iterative)*

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

    for (;  n > 0;  n--)
      ans = ans * n;

    return ans;
  }
}
```

Compiled bytecode

```
 0 iconst_1
 1 istore_2
 2 iload_1
 3 ifle 16
 6 iload_2
 7 iload_1
 8 imul
 9 istore_2
10 iinc 1, -1
13 goto 2
16 iload_2
17 ireturn
```

Evolutionary Operators

Good and bad crossovers

Replace a section in **A** with section from **B**

Bytecode 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
```

⇐

Bytecode B

```
 0 iconst_1
 1 istore_2
 2 iload_1
 3 ifle 16
 6 iload_2
 7 iload_1
 8 imul
 9 istore_2
10 iinc 1, -1
13 goto 2
16 iload_2
17 ireturn
```

Evolutionary Operators

Good crossover example

Stack use is depth- and type-consistent, variables are initialized.

Bytecode 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
```

⇐

Bytecode B

```
 0 iconst_1
 1 istore_2
 2 iload_1
 3 ifle 16
 6 iload_2
 7 iload_1
 8 imul
 9 istore_2
10 iinc 1, -1
13 goto 2
16 iload_2
17 ireturn
```

Evolutionary Operators
Good crossover example

Stack use is depth- and type-consistent, variables are initialized.

Bytecode ($A \overleftarrow{\times} B$)

```
 0 iconst_1
 1 istore_2
 2 iload_1
 3 ifle 12
 6 iload_1
 7 iload_2
 8 iload_1
 9 imul
10 imul
11 istore_2
12 iload_2
13 ireturn
```

Decompiled source

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

    if (n > 0)
      ans = n * (ans * n);

    return ans;
  }
}
```

Evolutionary Operators
Bad crossover example

Stack use is depth- and type-inconsistent.

Bytecode 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
```

⇐

Bytecode B

```
 0 iconst_1
 1 istore_2
 2 iload_1
 3 ifle 16
 6 iload_2
 7 iload_1
 8 imul
 9 istore_2
10 iinc 1, -1
13 goto 2
16 iload_2
17 ireturn
```

Evolutionary Operators
Bad *crossover example*

Stack use is depth- and type-inconsistent.

Bytecode ($A \overleftarrow{\times} B$)

```
 0 iconst_1
 1 istore_2
 2 iload_1
 3 ifle 13
 6 iload_2
 7 iload_1
 8 invokevirtual #2
11 imul
12 istore_2
13 iload_2
14 ireturn
```

“Decompiled” source

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

    if (n > 0)
      ans = ans.fact(n) * ?;

    return ans;
  }
}
```

Compatible Crossover

Constraints of unidirectional crossover $\mathbf{A} \overleftarrow{\times} \mathbf{B}$

Good crossover is achieved by respecting bytecode constraints:

(α *is target section in* ***A***, β *is source section in* ***B***)

Operand stack

e.g., β doesn't pop values with types incompatible to those popped by α

Compatible Crossover
Constraints of unidirectional crossover $\mathbf{A} \overleftarrow{\times} \mathbf{B}$

Good crossover is achieved by respecting bytecode constraints:
(*α is target section in **A**, β is source section in **B***)

Operand stack

e.g., β doesn't pop values with types incompatible to those popped by α

Local variables

e.g., variables read by β in **B** must be written before α in **A** with compatible types

Compatible Crossover

Constraints of unidirectional crossover $\mathbf{A} \overset{\leftarrow}{\times} \mathbf{B}$

Good crossover is achieved by respecting bytecode constraints:
(α *is target section in* ***A***, β *is source section in* ***B***)

Operand stack

e.g., β doesn't pop values with types incompatible to those popped by α

Local variables

e.g., variables read by β in **B** must be written before α in **A** with compatible types

Control flow

e.g., branch instructions in β have no "outside" destinations

Formal Definition
(Example of operand stack requirement)

α and β have compatible stack frames up to stack depth of β: pops of α have identical or narrower types as pops of β; pushes of β have identical or narrower types as pushes of α

Good crossover

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

Stack pops “`AB`” *(2 stop tack frames)* are narrower than “`AA`”, whereas stack push “`C`” is narrower than “`B`”

Types hierarchy: `C` → `B` → `A`

(see [Orlov and Sipper, 2009] for full formal definitions)

Formal Definition
(Example of operand stack requirement)

α and β have compatible stack frames up to stack depth of β:
pops of α have identical or narrower types as pops of β;
pushes of β have identical or narrower types as pushes of α

Bad crossover

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

Stack pops “`AB`” are not narrower than “`Af`”
(B and f are incompatible);
stack push “`A`” is not narrower than “`B`”

Types hierarchy: `B` ⟶ `A`; `f` is a `float`

(see [Orlov and Sipper, 2009] for full formal definitions)

Symbolic Regression
As an evolutionary example. . .

Parameters

- Objective: symbolic regression, $x^4 + x^3 + x^2 + x$
- Fitness: sum of errors on 20 random data points in $[-1, 1]$
- Input: **Number** `num` *(a Java type)*

Symbolic Regression
As an evolutionary example...

Parameters

- Objective: symbolic regression, $x^4 + x^3 + x^2 + x$
- Fitness: sum of errors on 20 random data points in $[-1, 1]$
- Input: **Number** `num` *(a Java type)*

Seeding

- Population initialized using seeding

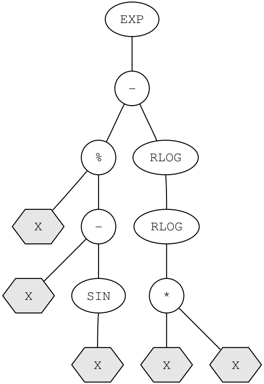
  [Langdon and Nordin, 2000]
- Seed population with clones of Koza's original worst-of-generation-0

  [Koza, 1992]

Symbolic Regression
Seeding with Koza's worst-of-generation-0

Original **Lisp** individual and its **tree** representation:

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



Symbolic Regression
Seeding with Koza's worst-of-generation-0

Translation to **unrestricted Java**

```
class Gecco {
  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 */
}
```

We added a couple of building blocks in the last line

Symbolic Regression
Setup and Statistics

Setup *(similar to Koza's)*

- Population: 500 individuals
- Generations: 51 (or less)
- Probabilities: $p_{\text{cross}} = 0.9$
  (α and β segments are uniform over segment sizes)
- Selection: binary tournament

Symbolic Regression

Setup and Statistics

Setup *(similar to Koza's)*

- Population: 500 individuals
- Generations: 51 (or less)
- Probabilities: $p_{\text{cross}} = 0.9$
  (α and β segments are uniform over segment sizes)
- Selection: binary tournament

Statistics

- Yield: 99% of runs successful (out of 100)
- Runtime: 30–60 s on dual-core 2.6 GHz Opteron
- Memory limits: insignificant w.r.t. runtime

Symbolic Regression
Evolved perfect individuals

A perfect solution easily evolves:

(beware of decompiler quirks!)

```
class Gecco_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 */
}
```

Computes $(x + x \cdot x \cdot x) + (x + x \cdot x \cdot x) \cdot x \ = \ x(1 + x)(1 + x^2)$

Symbolic Regression
Evolved perfect individuals

Another solution:

```
class Gecco_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 */
}
```

Computes $x \cdot (x \cdot (x \cdot x + x) + x) + x \;=\; x(1 + x(1 + x(1 + x)))$

Conclusions

Completely **unrestricted** Java programs can be **evolved**
(via bytecode)

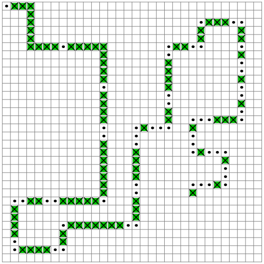
Extant (bad) Java programs can be **improved**
(e.g., initial regression seed)

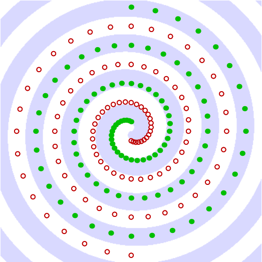
Future Work

Exhibit **viability** on other problems

We currentlty have results for:

complex regression, artificial ant, intertwined spirals, . . .



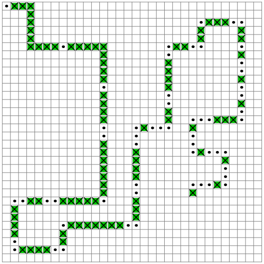


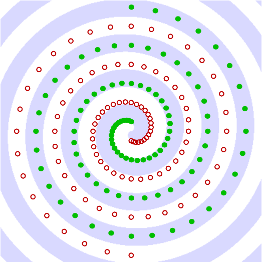
Future Work

Exhibit **viability** on other problems

We currentlty have results for:

complex regression, artificial ant, intertwined spirals, . . .





Loops and recursion are not a problem!

References

J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, MA, USA, Dec. 1992. ISBN 0-262-11170-5.

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 1802/2004 of *Lecture Notes in Computer Science*, pages 304–315, Berlin / Heidelberg, 2000. Springer-Verlag. ISBN 978-3-540-67339-2. doi:10.1007/b75085.

M. Orlov and M. Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, July 8–12, 2009, Montréal Québec, Canada*. ACM Press, July 2009. To appear.

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