Applying Genetic Programming to Bytecode and Assembly

Eric Collom

Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

29 April '14,
UMM Senior Seminar

Outline

1 Evolutionary Computation

2 Why Evolve Bytecode and Assembly?

3 Java bytecode and the JVM

4 FINCH:Evolving Java Bytecode

5 Using Instruction-level Code to Automate Bug Repair

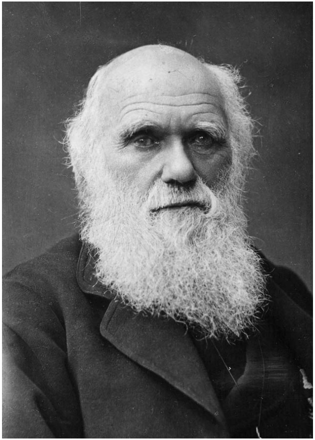
6 Conclusions

Outline

1. Evolutionary Computation
   - What is it?
   - How does it work?
   - Genetic Programming
2. Why Evolve Bytecode and Assembly?
3. Java bytecode and the JVM
4. FINCH:Evolving Java Bytecode
5. Using Instruction-level Code to Automate Bug Repair

What is Evolutionary Computation?

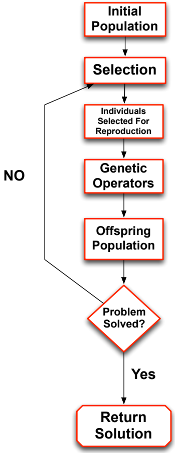
- Evolutionary Computation (EC) is a technique that is used to automate computer problem solving.
- Loosely emulates evolutionary biology



Charles Darwin
http://tinyurl.com/lqwj3wt

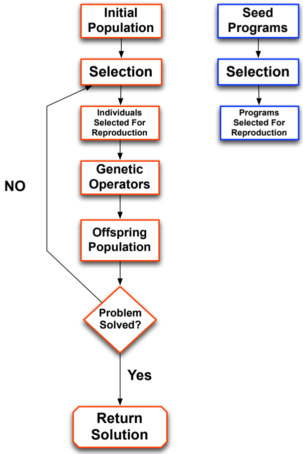
How does it work?

- Continuous optimization
- Selection is driven by the *fitness* of individuals
- Genetic operators mimic sexual reproduction and mutation



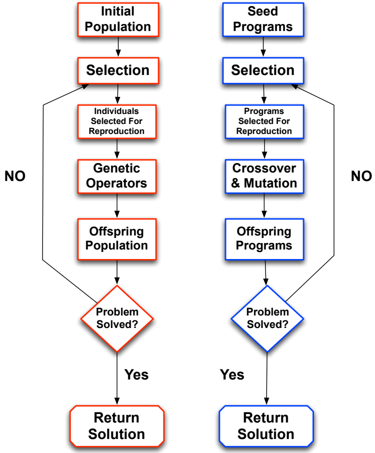
Genetic Programming

- Genetic programming (GP) uses the EC process to evolve **programs**
- This done by using an Evolutionary Algorithm (EA)

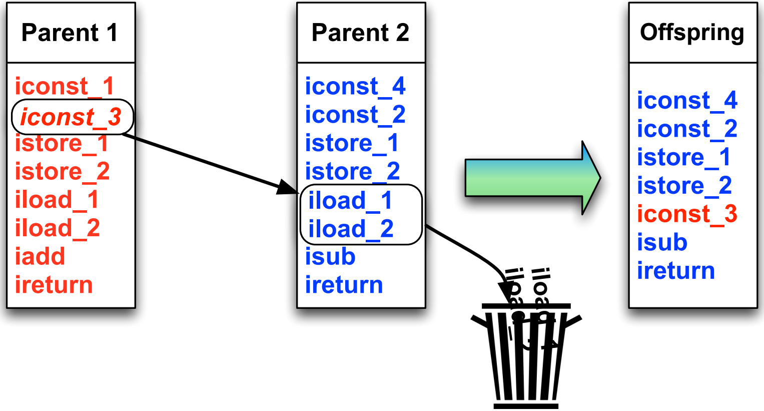


Genetic Programming

Two genetic operators used in GP are *crossover* and *mutation*



Crossover



Parent 1

iconst_1
iconst_3
istore_1
istore_2
iload_1
iload_2
iadd
ireturn

Parent 2

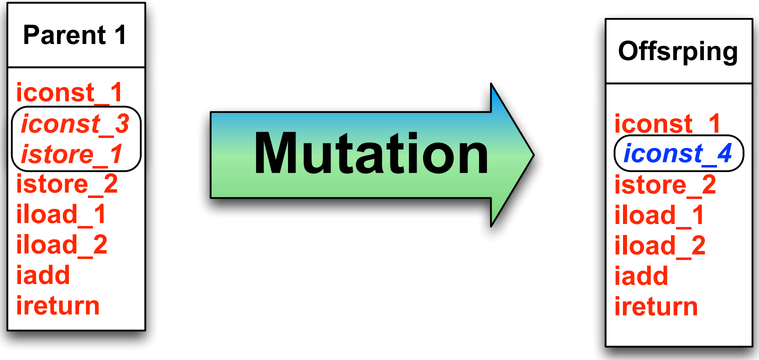
iconst_4
iconst_2
istore_1
istore_2
iload_1
iload_2
isub
ireturn

Offspring

iconst_4
iconst_2
istore_1
istore_2
iconst_3
isub
ireturn

Crossover with Java Bytecode

Mutation



Parent 1

iconst_1
iconst_3
istore_1
istore_2
iload_1
iload_2
iadd
ireturn

Mutation

Offsrping

iconst_1
iconst_4
istore_2
iload_1
iload_2
iadd
ireturn

Mutation with Java Bytecode

Outline

1. Evolutionary Computation

2. Why Evolve Bytecode and Assembly?
   - Difficulties With Source Code
   - Instruction-Level Code

3. Java bytecode and the JVM

4. FINCH:Evolving Java Bytecode

5. Using Instruction-level Code to Automate Bug Repair

Source Code Semantic Constraints

- It is difficult to apply evolution to an entire program in source code
  - Source code is made to simplify reading and writing programs
  - Source code does not represent the semantic constraints of the program.

Syntax vs Semantics

- Syntax represents structure
- Semantics represent meaning

Semantically Wrong: The sun rises in the West.
Semantically Correct: The sun rises in the East.

Syntax vs Semantics

Both (a) and (b) are valid syntactically. However, (b) is invalid semantically.

```
float x; int y = 7;
if(y>= 0){
    x=y;
}else{
    x= -y;
}
System.out.println(x);
```

(a)

```
float y; int x = 7;
if(y>= 0){
    y=x;
    x=y;
}
System.out.println(z);
```

(b)

Instruction-Level Code Constraints

- Consists of smaller alphabets
- Simpler syntactically
- Fewer semantic constraints to violate

Outline

1 Evolutionary Computation

2 Why Evolve Bytecode and Assembly?

3 Java bytecode and the JVM

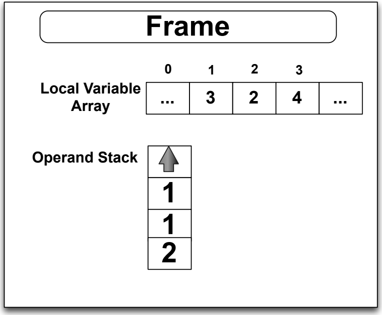
4 FINCH:Evolving Java Bytecode

5 Using Instruction-level Code to Automate Bug Repair

6 Conclusions

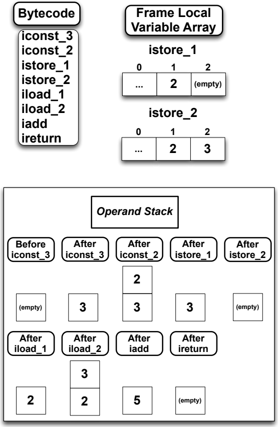
Java Virtual Machine

- A frame stores data and partial results as well as return values for methods
- Each method call has a frame



Java bytecode and Frames

- Opcodes
- The prefix indicates type



Bytecode

```
iconst_3
iconst_2
istore_1
istore_2
iload_1
iload_2
iadd
ireturn
```

Frame Local Variable Array

istore_1

0	1	2
...	2	(empty)

istore_2

0	1	2
...	2	3

Operand Stack

Before iconst_3: (empty)

After iconst_3: 3

After iconst_2: 2, 3

After istore_1: 3

After istore_2: (empty)

After iload_1: 2

After iload_2: 3, 2

After iadd: 5

After ireturn: (empty)

Outline

1 Evolutionary Computation

2 Why Evolve Bytecode and Assembly?

3 Java bytecode and the JVM

4 FINCH:Evolving Java Bytecode
- How it Works
- The Array Sum Problem

5 Using Instruction-level Code to Automate Bug Repair

What is FINCH?

- FINCH is an EA developed by Orlov and Sipper
- It evolves Java bytecode
- It deals with semantic constraints

Dealing With Semantic Constraints

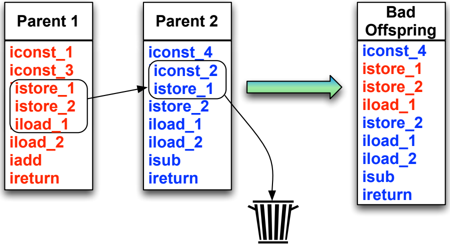
The semantic constraints that are checked for are

- Stack and Frame Depth
- Variable Types
- Control Flow

Dealing With Semantic Constraints

1. Apply crossover to two parents
2. Check if the offspring complies to semantic constraints
3. If the program passes the constraint test then it proceeds to offspring generation
4. If it fails the constraint check then another attempt is made with the same parents

Bad Crossover



Parent 1

iconst_1
iconst_3
istore_1
istore_2
iload_1
iload_2
iadd
ireturn

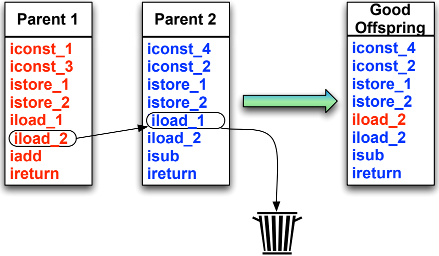
Parent 2

iconst_4
iconst_2
istore_1
istore_2
iload_1
iload_2
isub
ireturn

Bad Offspring

iconst_4
istore_1
istore_2
iload_1
istore_2
iload_1
iload_2
isub
ireturn

Good Crossover



Array Sum

- The array sum problem
    - Started with a worst case fitness seed program
    - Counted function calls to check for a non-halting state

```
int sumlistrec(List list) {
   int sum = 0;
  if(list.isEmpty())
     sum *= sumlistrec(list);
  else
     sum += list.get(0)/2 + sumlistrec(
          list.subList(1, list.size()));

  return sum;
}
```

Array Sum

Decompiled Solution

```
int sumlistrec(List list) {
     int sum = 0;
    if(list.isEmpty())
        sum = sum;
    else
        sum += ((Integer) list.get(0)).intValue() +
                sumlistrec(list.subList(1,list.size()));

    return sum;
}
```

Outline

1 Evolutionary Computation

2 Why Evolve Bytecode and Assembly?

3 Java bytecode and the JVM

4 FINCH:Evolving Java Bytecode

5 Using Instruction-level Code to Automate Bug Repair

- How it Works
- Results

Automating Bug Repair

- Schulte, et al., automated bug repair by evolving Java bytecode and x86 assembly
- Fixed bugs in real code
- Did not check for semantic constraints

Weighted Path

- Programs at times consist of thousands of lines of code
- Uses a weighted path due to size of programs
- The weight of a path was determined by the instructions that were executed by tests

Weighted Path

- Test were provided that consisted of one *negative* test and multiple *positive* tests
- The negative test was used to represent the bug and check if individuals found a solution
- The positive tests were used to retain functionality

Instruction Weight

- Each instruction executed only by the negative test was given a weight of 1.0
- An instruction executed by the negative test and atleast one positive was given a weight of 0.1
- If an instruction was not executed by the negative test case a weight of 0 was assigned

What was debugged?

Schulte et al., were able to debug:

- Infinite loops
- Buffer overflows
- Incorrect type declarations

Outline

1 Evolutionary Computation

2 Why Evolve Bytecode and Assembly?

3 Java bytecode and the JVM

4 FINCH:Evolving Java Bytecode

5 Using Instruction-level Code to Automate Bug Repair

6 Conclusions

Conclusions

- It is difficult to evolve entire programs in source code due to semantic constraints
- It is easier to deal with semantic constraints with instruction-level code
- It is feasible to not deal with semantic constraints in some situations
- It is possible to evolve small programs and fix simple bugs using instruction level code

coll0474@morris.umn.edu

Questions?

References

M. Orlov and M. Sipper.
Flight of the FINCH Through the Java Wilderness.
Evolutionary Computation, IEEE Transactions on, 15(2):166–182, April 2011.

E. Schulte, S. Forrest, and W. Weimer.
Automated Program Repair Through the Evolution of Assembly Code.
In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 313–316, New York, NY, USA, 2010. ACM.