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

The big picture

- Evolving whole programs is hard to do with source code.
- Evolving whole programs with bytecode and assembly is not as hard.

Outline

1 Evolutionary Computation

2 Why Byetocde 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 Byetocde and Assembly?

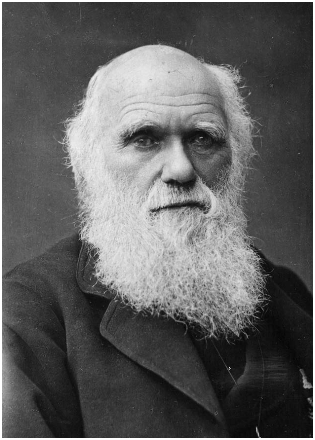
3 Java Bytecode and the JVM

4 FINCH:Evolving Java Bytecode

5 Using Instruction-level Code to Automate Bug Repair

What is Evolutionary Computation?

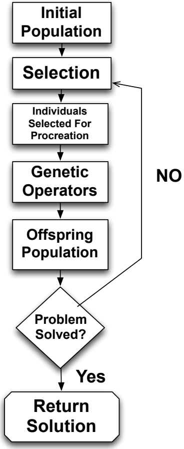
- Evolutionary Computation (EC) is a a technique that is used to automate computer problem solving.
- Loosely emulates evolutionary biology.



Charles Darwin
http://tinyurl.com/lqwj3wt

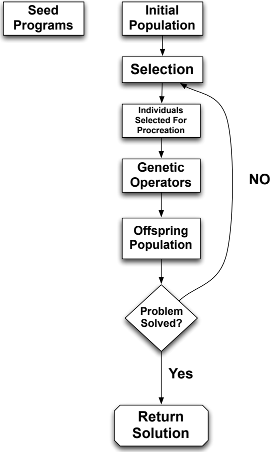
How does it work?

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



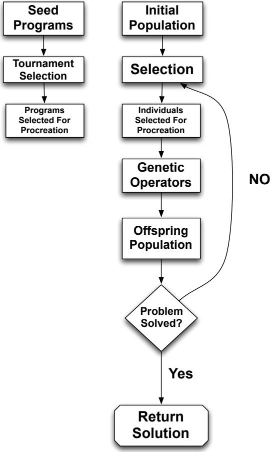
Genetic Programming

- Uses the EC process to evolve programs
- This done by using Evolutionary Algorithm (EA)
- The population consists of programs



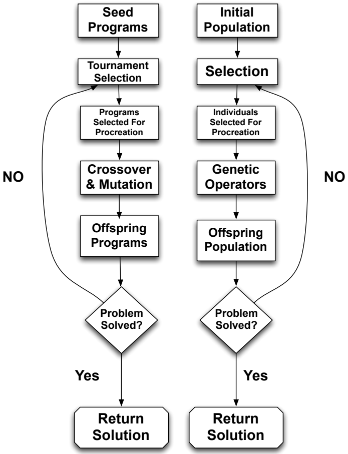
Genetic Programming

- Tournament Selection
  - Randomly select a specified number of programs
  - Pick the program with the highest fitness
  - That program then is selected for reproduction

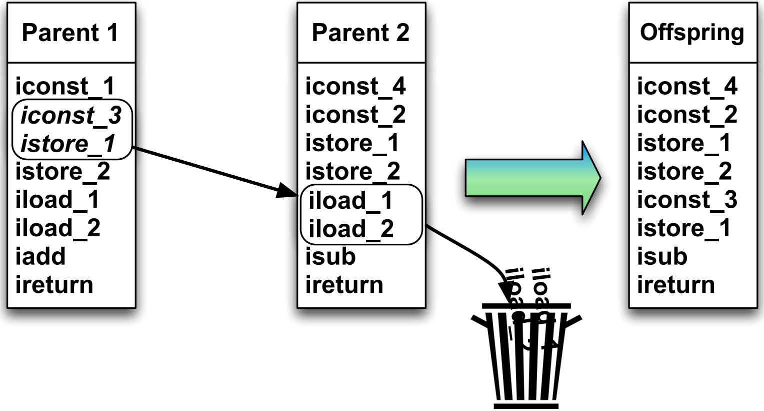


Genetic Programming

- Crossover
  - Sexual reproduction
- Mutation
  - Asexual reproduction
  - Can be used along with crossover



Crossover



Parent 1

iconst_1
iconst_3
istore_1
istore_2
iload_1
iload_2
iadd
ireturn

Parent 2

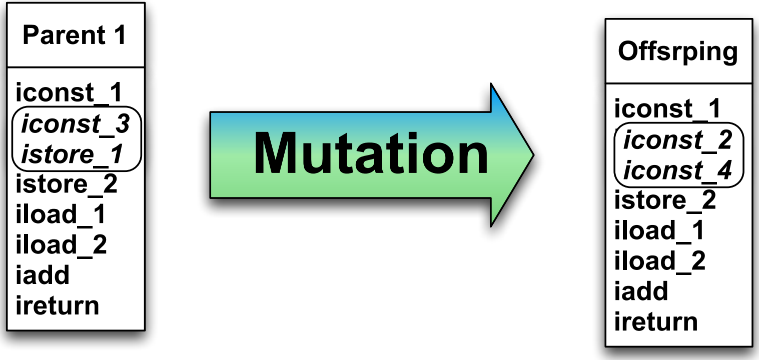
iconst_4
iconst_2
istore_1
istore_2
iload_1
iload_2
isub
ireturn

Offspring

iconst_4
iconst_2
istore_1
istore_2
iconst_3
istore_1
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_2
iconst_4
istore_2
iload_1
iload_2
iadd
ireturn

Crossover with Java Bytecode

Outline

1 Evolutionary Computation

2 Why Byetocde and Assembly?

- Difficulties in Source Code

3 Java Bytecode and the JVM

4 FINCH:Evolving Java Bytecode

5 Using Instruction-level Code to Automate Bug Repair

6 Conclusions

Source Code Semantic Constraints

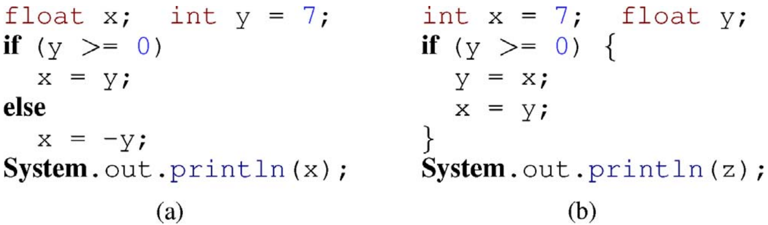
- It is difficult to apply evolution to an entire program in source code
  - Source code is made to simplify reading and writing programs
  - Source code does not represent the semantic constraints of the program.

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



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

(a)

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

(b)

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

EAs and Source Code

- EAs that evolve source code are usually designed to avoid dealing with semantic constraints

```
class Robot{
...
  double robotSpeed(){
      double evolvedVariable = valueFromEA;
      return (robot.location + evolvedVariable)/2;
  }
...
}
```

Instruction-Level Code Constraints

- Consists of a smaller alphabets
- Simpler syntactically
- Less semantic constraints to violate

Instruction-level Code Benefits

- Do not need to understand the structure of the program being evolved
- Can evolve a lot from a little
- If there is a compiler for it we can evolve it

Outline

1 Evolutionary Computation

2 Why Byetocde and Assembly?

3 Java Bytecode and the JVM

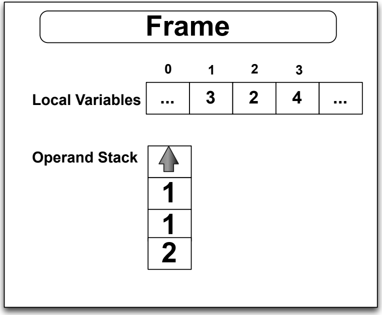
4 FINCH:Evolving Java Bytecode

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



Frame

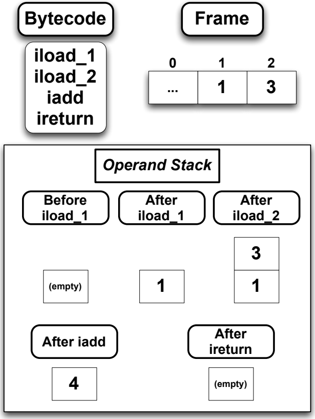
	0	1	2	3	
Local Variables	...	3	2	4	...

Operand Stack

1
1
2

Java Bytcode and Frames

- Opcodes
- The prefix indicates type



Outline

1 Evolutionary Computation

2 Why Byetocde 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 M. Orlov and M. Sipper
- It evolves Java bytecode
- It deals with semantic constraints

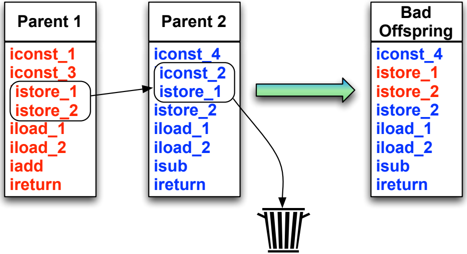
Dealing With Semantic Constraints

- There is still a chance to produce non-executable code
- Solution: check if offspring follows semantic constraints
  - Stack and Frame Depth
  - Variable Types
  - Control Flow

Dealing With Semantic Constraints

1. Apply crossover to two parents
2. Check if they comply to semantic constraints
3. If the program passes the constraint test then it proceeds to offspring generation
4. If it fails the constrain 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

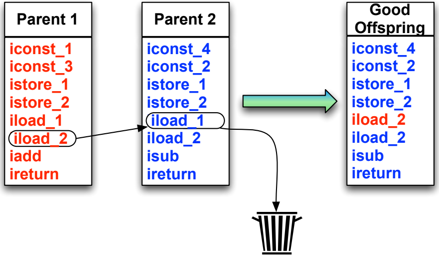
Parent 2

iconst_4
iconst_2
istore_1
istore_2
iload_1
iload_2
isub
ireturn

Bad Offspring

iconst_4
istore_1
istore_2
istore_2
iload_1
iload_2
isub
ireturn

Good 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

Good Offspring

iconst_4
iconst_2
istore_1
istore_2
iload_2
iload_2
isub
ireturn

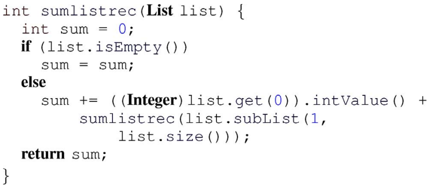
Array Sum

- The array sum problem
    - Started with a zero 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 Byetocde 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., designed EAs that fixes bugs in x86 assembly and Java bytecode
- Evolves Java bytecode and x86 Assembly
- Does not check for semantic constraints

Tests and Fitness

- Fitness was determined by tests
- Test consisted of one *negative* test and multiple *positive* tests
- The negative test was used to check if the bug was fixed

- Programs at times consist of thousands of lines of code
- Uses a weighted path due to size of programs
- The weighted path was determined by what tests execute that instruction

Instruction Weight

- Only executed by failing test: weight = 1.0
- Executed by negative test and one positive: weight = 0.1
- Not executed by negative test case: weight = 0

Bugs Fixed

- Buffer overflow
- Infinite loops

Outline

1 Evolutionary Computation

2 Why Byetocde 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

[1] [2]

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.