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. Background
2. Why Evolve Instruction-Level Code
3. FINCH:Evolving Programs
4. Using Instruction-level Code to Automate Bug Repair
5. Conclusions

Outline

1 Background
- EC
- Java Bytecode and the JVM

2 Why Evolve Instruction-Level Code

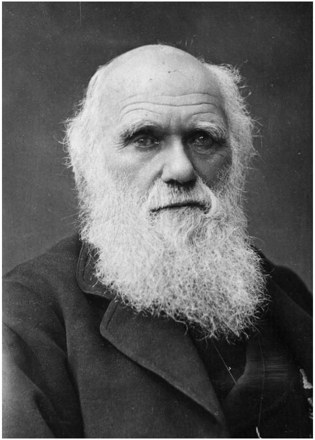
3 FINCH:Evolving Programs

4 Using Instruction-level Code to Automate Bug Repair

5 Conclusions

What is Evolutionary Computation?

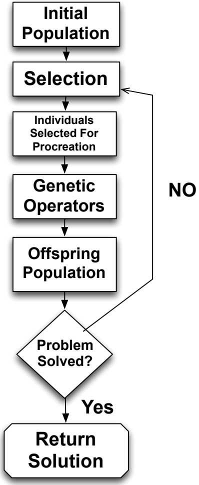
- EC is a a technique that is used to automate computer problem solving.
- Loosely emulates evolutionary biology.



Charles Darwin
`http://tinyurl.com/lqwj3wt`

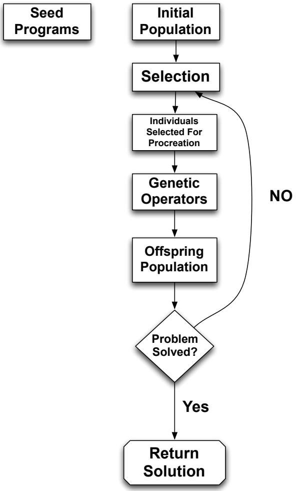
How does it work

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



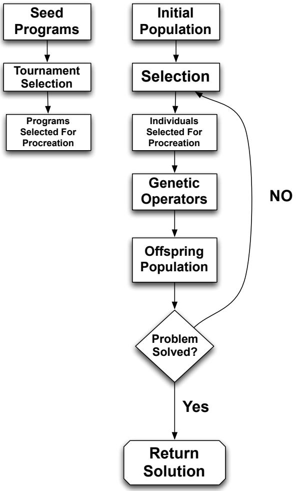
Genetic Programming

- Uses the EC technique to evolve programs
- The population is programs



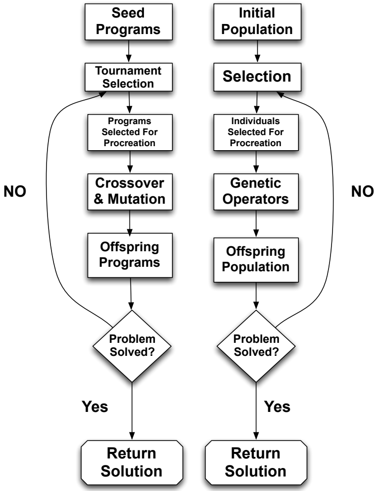
Genetic Programming

- Tournament Selection

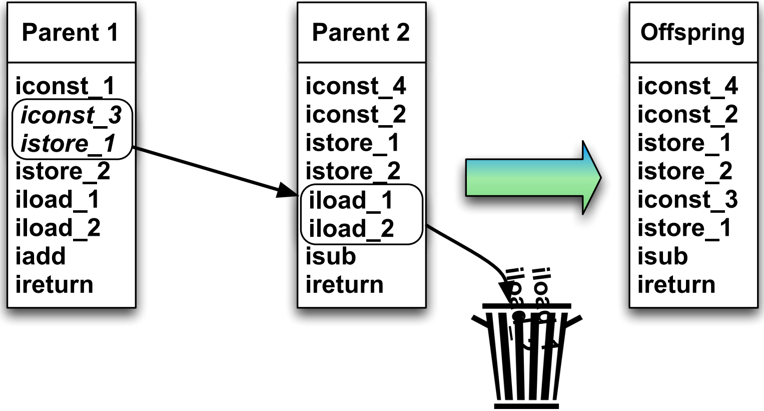


Genetic Programming

- Crossover
- Mutation



Crossover



Parent 1

iconst_1
iconst_3
istore_1
istore_2
iload_1
iload_2
iadd
ireturn

Parent 2

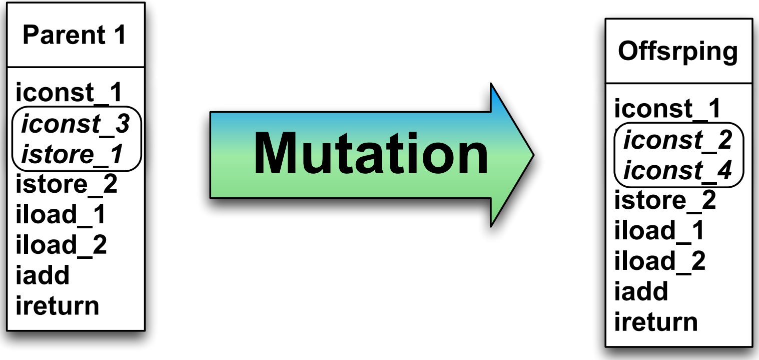
iconst_4
iconst_2
istore_1
istore_2
iload_1
iload_2
isub
ireturn

Offspring

iconst_4
iconst_2
istore_1
istore_2
iconst_3
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

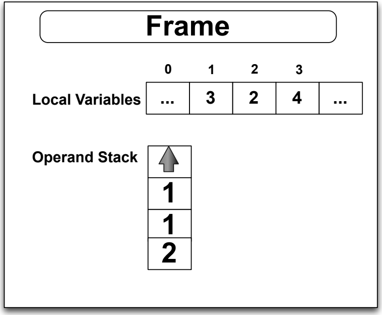
Offsrping

iconst_1
iconst_2
iconst_4
istore_2
iload_1
iload_2
iadd
ireturn

Crossover with Java Bytecode

Java Virtual Machine

- Frames
- Array of local variables
- Operand Stack



Frame

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

Operand Stack

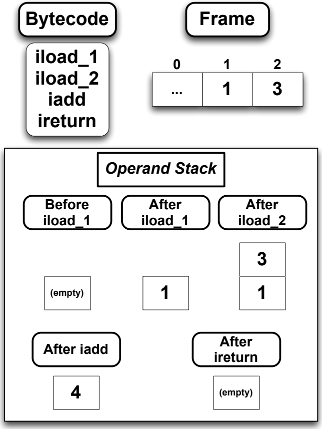
1

1

2

Java Bytcode and Frames

- Opcodes
- Prefix indicates type



Outline

1 Background

2 Why Evolve Instruction-Level Code

3 FINCH:Evolving Programs

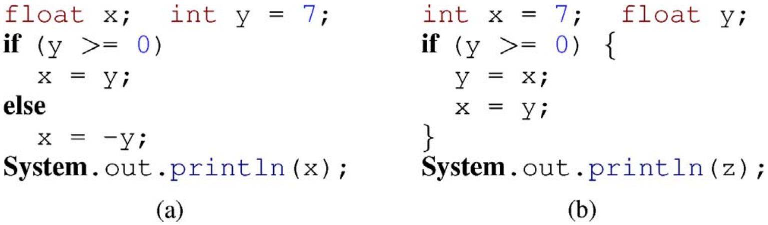
4 Using Instruction-level Code to Automate Bug Repair

5 Conclusions

Source Code Constraints

- While it would be useful, 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.

Source Code Constraints



```
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.

Source Code Constraints

- EAs 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 Background

2 Why Evolve Instruction-Level Code

3 FINCH:Evolving Programs
- How it Works
- Results

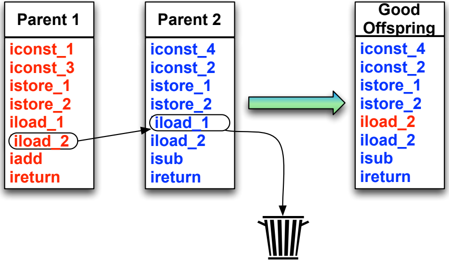
4 Using Instruction-level Code to Automate Bug Repair

5 Conclusions

Selecting Offspring

- There is still a chance to produce non-compilable code
- Solution: add restrictions to code selection
  - Stack and Frame Depth
  - Variable Types
  - Control Flow

Good Crossover



Parent 1

```
iconst_1
iconst_3
istore_1
istore_2
iload_1
iload_2
iadd
ireturn
```

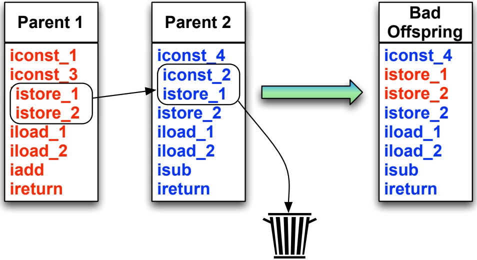
Parent 2

```
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
```

Bad Crossover

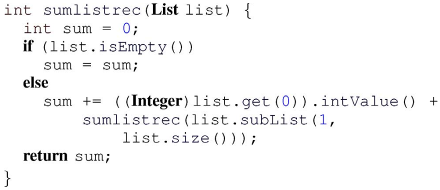


Results

- 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;
}
```

Results



```
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 Background

2 Why Evolve Instruction-Level Code

3 FINCH:Evolving Programs

4 Using Instruction-level Code to Automate Bug Repair
- How it Works
- Results

5 Conclusions

Selecting Offspring

- Debugging programs that consist of thousands of lines of code
- Uses a weighted path due to size of programs
- No checks if executable instruction-level code is produced

Selecting Offspring

- Each instruction is given a weight
- The weight is determined by what tests execute that instruction
- Each experiment started with one negative test case and multiple positive test cases

Non-Halting Offspring

- 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

Outline

1 Background

2 Why Evolve Instruction-Level Code

3 FINCH:Evolving Programs

4 Using Instruction-level Code to Automate Bug Repair

5 Conclusions

Conclusions

References