Embedding GI inside the JVM

Benoit Baudry

Kwaku Yeboah-Antwi
Specialize to environment .. or not
Open-ended evolution

RED QUEEN HYPOTHESIS

IN REFERENCE TO AN EVOLUTIONARY SYSTEM, CONTINUING ADAPTATION IS NEEDED IN ORDER FOR A SPECIES TO MAINTAIN ITS RELATIVE FITNESS AMONGST THE SYSTEMS IT IS CO-EVOLVING WITH
Continuously evolve
In software

• Code specialization
  • API are large and generic
  • exact usage (e.g., surface really needed) is known only at runtime

• Spontaneous diversification
  • Instantiation creates large quantities of clones
  • diversification at instantiation can increase resilience
Let’s do it in Java

• JVM hardly supports runtime modifications of the bytecode
  • E.g., add, delete or rename fields, methods

• No built-in support
  • to edit the bytecode
  • to steer the search for new code
Our Contribution

ECSEL R

• ECologically Inspired Software EvoLution @ Runtime

• A patch in the JVM + evolutionnary capacities for Java programs
The evo loop

Initialization — EvoAgent
- Action FN: getAllObjects()

Selection
- EvoDaemon
- Evo Strategy: selectObjects()
- Object B
- Object C

Variation
- EvoDaemon
- Genetic Oper: mutateObject()
- Object C
- Object B

Evaluation
- EvoAgent
- Action FN: testObjects()
- Object B
- Object C

Installation
- EvoAgent
- Action FN: redefObjects()
- Object B
- Object C

The records of Object B are tested for fitness. If the fitness is not suitable, the EvoDaemon discards the Object B. Otherwise, Object B is redefined.
The **Initialization** and **Selection** steps

- Analyze all live objects on the heap
- Select a subset according to some function
  - random
  - by size
  - by frequency of usage
- Copy the bytecode of selected objects
The step

• ECSELRI embeds default evo operations
  • Method Addition
  • Merge Methods
  • Method Deletion
  • Field Addition Operator
  • Transplantation Operator
  • Random Instruction Operator
  • Passthrough Operator Return
The **Evaluation** step

- **Static check**
  - ensure syntactic validity
  - check consistency after evolution: ECSLER embeds checks about the well-formedness of bytecode operations

- **Dynamic check**
  - parallel execution of original and evolved
  - fitness evaluation
public int greaterThan(int intOne, intTwo) {
    if (intOne > intTwo) {
        return 0;
    } else {
        return 1;
    }
}
greaterThan(10,20);
public int greaterThan(int intOne, intTwo) {
    if (intOne > intTwo) {
        return 0;
    } else {
        return 1;
    }
}

greaterThan(10,20);
Dynamic checks

• Run both versions in parallel
  • for a given period
• This requires our patched JVM

![Diagram]

- inputs
  - original
  - mutated

- eval fitness
  - if ok
    - keep mutated
  - if not ok
    - discard mutated
Diversification example

```java
int hash(final Object key) {
    int h = key.hashCode();
    h += ~(h << 9);
    h ^= h >>> 14;
    h += h << 4;
    h ^= h >>> 10;
    return h;
}
```
Diversification example

```java
public long hash() {
    long h = this.hashCode();
    h = this.initCapacity * h;
    h += ~(h << 9);
    h ^= h >>> 14;
    h += h << 4;
    h ^= h >>> 10;
    return h;
}
```

```java
public long hash() {
    long h = this.hashCode();
    h = this.initCapacity * h;
    h += ~(h << 9);
    h ^= h >>> 14;
    h += h << (4+6);
    h ^= h >>> 10;
    return h;
}
```

```java
public long hash() {
    long h = this.hashCode();
    h = this.initCapacity * h;
    h += ~(h << 9);
    h ^= h >>> 14;
    h += h << (4+6);
    h ^= h >>> 10;
    h = h * 100;
    return h;
}
```
Conclusion

• We now have a machine to evolve Java programs at runtime
• Next step: use it for runtime GI

• https://bitbucket.org/Kwaku/agentd/src