“Ants, Mutants and Beyond”
Combining formal and stochastic techniques to improve software

jerry.swan@york.ac.uk
douglas.carson@keysight.com
zoltan.kocsis@cs.stir.ac.uk

October 12, 2015
Overview

2. Search Based Software Engineering (SBSE).
3. Genetic Improvement (GI).
4. Automatic Improvement (AIP).
Some classes of problem are well-understood and have efficient solution methods (e.g. the simplex algorithm for problems with linear objectives/constraints).

However, in many cases we only have an operational description of the problem — calculating derivatives to drive a ‘conventional’ optimization method such as Newton’s method may not be easy (or even possible).

In such cases, it is common to turn to a variety of ‘nature-inspired’ metaheuristic search methods: Genetic Algorithms, Ant-Colony Optimization etc.
General problem-solving with Metaheuristics

Metaheuristics are **stochastic** (a form of ‘generate-and-test’) and require only a few **problem-specific ingredients**:

- **Solution representation**: e.g. list of cities for the TSP.
- **Objective function**: measures the quality of a solution (e.g. tour length).
- **Search operators**: some means of *mutating* or *(re)*combining solutions to produce a new solution (e.g. swapping two cities).
Genetic Algorithms/Genetic Programming - I

Idea: generate/breed solutions and apply ‘survival of the fittest’

John H. Holland (1929-2015)  
John Koza
Genetic Algorithms/Genetic Programming - II

Genetic Algorithms Pipeline

1. **Population of solutions**
2. **Fitness evaluation**
3. **Select according to fitness**
4. **Mutate individuals**
5. **Recombine pairs to produce children**
6. **Population of solutions**

The cycle continues through these steps, iterating to find the best solutions.
There is a recent trend to tackle problems in **software engineering** using metaheuristics:

- **Requirements Selection:**
  - Objective function: cost, value, . . .
  - Representation: bitvector of requirements.

- **Test case prioritization**
  - Objective function: rate of coverage, time, faults, . . .
  - Representation: permutation of the test suite.

- **Program Synthesis**
  - Objective function: correctness, speed, power, memory . . .
  - Representation: proof trees; expression trees; source code.
Pragmatics of Program Synthesis

**Scalability** remains an issue for program synthesis:

- We don’t yet know how to generate sizeable algorithms from scratch.
- **Generative** approaches such as *GP* still work best at the scale of expressions . . .
- . . . but **human ingenuity** already **provides a vast repertoire** of (abstract) *algorithms* and (concrete) *programs* . . .
The ‘Template Method’ Design Pattern\(^1\) divides an algorithm into a fixed skeleton and some variants.

The fixed parts orchestrate the behaviour of the variants.

Example: Quicksort performance depends on the pivot function, so we can treat it as a variant:

```java
DoubleArray qsort(DoubleArray arr) {
    double pivot = pivotFn(arr);
    // Implementation of pivotFn can be varied generatively
    return qsort(arr.filter(< pivot))
        ++ arr.filter(== pivot)
        ++ qsort(arr.filter(> pivot));
}
```

Expressing algorithms as templates allows us to learn good implementations for the variant parts.

\(^1\) [Gamma, Helm, et al. 1995].
‘Template Method Hyper-heuristics’\textsuperscript{3}

\textsc{templar}\textsuperscript{2} is a Java\textsuperscript{TM} framework designed to make the generation of customized algorithms as simple as possible. Ingredients:

- A list of \textit{variation points} describing the parts of the algorithm to be automatically generated.
- An \textit{algorithm template} expressing the algorithm skeleton. The template produces a \textit{customized version of the algorithm} from \textit{automatically-generated implementations} of the variation points.
- An \textit{objective function} to evaluate the customized algorithm.
- An \textit{algorithm factory} that \textit{searches the space of variation points} to produce an \textit{optimized version of the algorithm}.

\textsuperscript{2}[Swan and Burles 2015].
\textsuperscript{3}[Woodward and Swan 2014].
Hyper-quicksort: Optimizing for energy consumption

![Graph showing the energy consumption of different sorting algorithms (Mid, Sedgewick, Random, Hyper-quicksort) as the input array size increases. The x-axis represents the input array size in logarithmic scale, and the y-axis represents the energy consumption in Joules in logarithmic scale. The graph shows that Hyper-quicksort generally consumes less energy compared to the other algorithms.]
## Hyper-Quicksort - Results

<table>
<thead>
<tr>
<th>Array size</th>
<th>Middle index</th>
<th>Sedgewick</th>
<th>Random Index</th>
<th>Hyper-quick</th>
<th>sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.191</td>
<td>0.163</td>
<td>0.446</td>
<td>0.094</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.296</td>
<td>0.345</td>
<td>0.410</td>
<td>0.173</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0.651</td>
<td>0.757</td>
<td>0.967</td>
<td>0.410</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>1.366</td>
<td>1.145</td>
<td>1.708</td>
<td>0.976</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>3.505</td>
<td>4.034</td>
<td>5.221</td>
<td>2.341</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>8.175</td>
<td>7.646</td>
<td>9.269</td>
<td>6.387</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>19.777</td>
<td>21.391</td>
<td>27.685</td>
<td>15.268</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>62.961</td>
<td>42.508</td>
<td>41.245</td>
<td>33.012</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>198.438</td>
<td>132.663</td>
<td>111.894</td>
<td>70.234</td>
<td></td>
</tr>
</tbody>
</table>

Energy consumption (Joules) against input array size
Genetic Improvement (GI)

Addresses scalability by applying (a variant of) GP to *pre-existing programs*. It can be used to:

- Fix bugs\(^4\) (maintanance dominates software lifecycle cost).
- Obtain a multi-objective trade-off between Non-Functional Properties\(^5\) (NFPs).
- Optimize/improve functional properties\(^6\).

\(^4\)[Le Goues, Forrest, et al. 2013].
\(^6\)[Burles, Swan, et al. 2015].
GEN-O-FIX - Improving programs

A Scala framework for self-improving software systems\(^7\), i.e. it can improve a system as it runs.

- Performs both GIP and GP, rather than ‘plastic surgery’.
- Tight integration of compiler and improvement mechanism (via reflection) is more efficient and less brittle than existing approaches\(^8\).
- Callbacks to newly-generated functionality can also be injected into legacy Java code.
- Uses an actor-based approach for executing program variants.

---

\(^7\) [Swan, Epitropakis, et al. 2014].
Why Scala?

- Supports expressive webservice frameworks (‘Hello, Web’ in 6 lines of code).
- Increasingly popular for concurrency support (Twitter core rewritten in Scala).
- Extensively used in industry:
Gen-O-Fix System Diagram

Client Application

Dynamic adaptation for embedded systems

Gen-O-Fix

Generates

Input

Source + Binary

Optimises

Performance

Energy

Optimises

Memory

Optimises

Integrity
**Gen-O-Fix** example: Hotswapping webservers code

- A stock-price predictor for shares\(^9\) in David Bowie\(^{10}\).
- Achieved via univariate symbolic regression . . .
- of a function extracted from web-application source code.

\(^9\)*Not actual shares.
\(^{10}\)*Not the actual David Bowie.
\textbf{Automatic Improvement}

\[ \text{project : } \text{OZSpec} \rightarrow \text{UMLDiagram} \]

\[ \forall(oz, uml) : \text{project} \]
\[ \{ c : oz \cap \text{Classdef} \land c.\text{name} \} = \{ c : uml.\text{classes} \]
\[ \land c.\text{name} \} \land \forall c_1, c_2 : oz \cap \text{Classdef} \land \exists_1 c' : \]
\[ uml.\text{classes} \land c'.\text{name} = c_1.\text{name} \]
\[ c'.\text{attrs} = \{ cls : \text{Classdef} \mid cls \in oz \land cls.\text{name} \}
\[ \triangleleft c_1.\text{state.decpart} \]
\[ c'.\text{ops} = \{ o : \text{Opdef} \mid o \in c_1.\text{ops} \land o.\text{name} \} \]
\[ c_2.\text{name} \in \{ t : \text{ran} c_1.\text{state.decpart} \land t.\text{name} \} \Rightarrow \]
\[ \exists_1(c_1', c_2') : uml.\text{agg} \land c_1'.\text{name} = c_1.\text{name} \]
\[ \land c_2'.\text{name} = c_2.\text{name} \]
\[ c_2.\text{name} \in \{ \text{inh} : \text{dom} c_1.\text{inherit} \land \text{inh}.\text{name} \} \Rightarrow \]
\[ \exists_1(c_1', c_2') : uml.\text{inh} \land c_1'.\text{name} = c_1.\text{name} \]
\[ \land c_2'.\text{name} = c_2.\text{name} \]
AIP versus Formal Methods

Automatic Improvement Programming (AIP) provides (some of) the benefits of formal methods without the difficulty of writing formal specs.

- **Formal Methods:**
  - Require highly specialized developers.
  - Hard to write large programs.

- **AIP:**
  - Doesn’t need formal specs or tools with steep learning curve.
  - Has no scalability issues since we search for transformable patterns in the source code.
AIP versus GI

- Metaheuristic approaches to program improvement (e.g. GI):
  - Rely heavily on **random** perturbation / recombination.
  - Can **degrade** program structure/correctness/explanatory power.

- AIP:
  - Can use **semantics-preserving** transformations.
  - Can come with an **asymptotic** guarantee of superiority.
Well-known that implementing `hashCode` in Java is (often fatally) error-prone.

Our analysis revealed 487 incorrect implementations in Apache HADOOP\(^{11}\).

We **repaired** HADOOP by correcting `hashCode` implementations (semantics-preserving), whilst simultaneously improving on the efficiency of the incorrect version (generative).

\(^{11}\)Kocsis, Neumann, et al. 2014.
**PolyFunic** uses Category Theory to replace stochastic **Gen-O-Fix** mutation operators with *catamorphisms*:

- This guarantees that the mutation is semantics-preserving.
- Trial study obtained an asymptotic improvement in efficiency\(^{12}\) \((O(n) \text{ to } O(1))\).

Performance comparison of na"ıve and AIP-optimized code

\(^{12}\)[Kocsis and Swan 2014].
£80K research income from:

- Dataductus: Funded the application of *search combinators* to hybrid search.
- BT: Funding research studentship in adaptive scheduling.
- Keysight: funded development of **HYLAS** (one of only 30 such grants worldwide).
References I


References II


References III

