Optimizing the Evaluation of Patterns in Pointcunts

Remko Bijker, rbijker@student.utwente.nl
Christoph Bockisch, c.m.bockisch@cs.utwente.nl
Andreas Sewe, sewe@st.informatik.tu-darmstadt.de
Aspect-Oriented Programming

- Localizing implementation of behavior common to multiple modules
- Common core concepts
  - Definition of functionality (*advice* in AspectJ)
  - Definition when functionality is applicable (*pointcut* in AspectJ)
- Pointcuts *evaluate* to set of *join points*
  - Generally regions in time (e.g., beginning of method invocation till returning from method)
- Join points described in terms of
  - *Static properties*
  - *Dynamic properties*
Aspect-Oriented Execution Environments

- Aspect deployment
  - Aspect must become active
  - Partially evaluate pointcuts to join-point shadows (code locations)
- Insert instructions
  - Evaluating dynamic properties
  - Executing advice

Support for deployment at
- Compile-time
- Class-loading-time
- Run-time

Partial pointcut evaluation affects overall application performance.
Partial Pointcut Evaluation

- Static part of pointcut includes
  - Kind of join point (e.g., method call, field write)
  - **Pattern** over **signature** of associated member

- (Naïve) partial evaluation
  - **Iterate** over list of potential join-point shadows
  - Match pattern against each

- Hypotheses
  - Naïve partial evaluation is **slow**
  - Patterns only match few join-point shadows
  - Data (signatures and patterns) is **structured**
Optimization Strategies for Pattern Evaluation

- **Observation**
  - *Signatures* and *patterns* made up of *parts*
  - Match when all sub-patterns match

<table>
<thead>
<tr>
<th>Method</th>
<th>Declaring Class</th>
<th>Modifiers</th>
<th>Result Type</th>
<th>Name</th>
<th>Parameter Types</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>Declaring Class</td>
<td>Modifiers</td>
<td></td>
<td></td>
<td>Parameter Types</td>
<td>Exceptions</td>
</tr>
<tr>
<td>Static Initializer</td>
<td>Declaring Class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field (Read/Write)</td>
<td>Declaring Class</td>
<td>Modifiers</td>
<td>Type</td>
<td>Name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- List of join-point shadows as *database table*
  - Signature parts as columns, patterns as queries
  - *Index* on signature part speeds up sub-pattern
  - *Search-plan optimization* orders sub-pattern evaluation
Research Questions

- Which evaluation order is (heuristically) most efficient?
  - What is the selectivity of each sub-pattern?

- Which is the best data structure for an index?

- Is the additional effort of maintaining an index paying off?
Experimental Setup

- Survey **real-world programs**
  - OO programs for understanding signatures
    - ANTLR, FreeCol, LIAM, TightVNC
    - Total: 2432 classes, 28065 signatures, 150432 join-point shadows
  - AO programs for understanding patterns
    - ajlib-incubator, Contract4J5, Glassbox, Nversion, Sable Benchmarks
    - Total: 242 aspects, 170 different patterns
  - Pattern evaluation against Java tools and Java Runtime library
- Based on ALIA4J language-implementation approach
  (shown to support AspectJ, Compose*, ConSpec, etc.)
  - Used parts of ALIA4J to extract signatures
  - **Prototyped optimizations** in ALIA4J execution environment
  - Measured evaluation times
Survey Results (Method Patterns)

- Most patterns match methods (149/170)
- Usage of sub-pattern kinds

<table>
<thead>
<tr>
<th>Sub-patter</th>
<th>Any</th>
<th>Wildcard pattern</th>
<th>Exact pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaring class</td>
<td>24%</td>
<td>1%</td>
<td>75%</td>
</tr>
<tr>
<td>Name</td>
<td>12%</td>
<td>16%</td>
<td>72%</td>
</tr>
<tr>
<td>Parameter Types</td>
<td>60%</td>
<td>4%</td>
<td>36%</td>
</tr>
<tr>
<td>Return type</td>
<td>82%</td>
<td>1%</td>
<td>17%</td>
</tr>
<tr>
<td>Modifiers</td>
<td>91%</td>
<td>—</td>
<td>9%</td>
</tr>
<tr>
<td>Exceptions</td>
<td>96%</td>
<td>2%</td>
<td>2%</td>
</tr>
</tbody>
</table>
Survey Results (Method Patterns)

- Determine **selectivity** of sub-patterns
  - Evaluate all patterns against
  - All signatures in surveyed OO applications

- Optimal **evaluation order**

<table>
<thead>
<tr>
<th>Declaring class</th>
<th>Name</th>
<th>Return type</th>
<th>Modifiers</th>
<th>Parameters</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Application of Survey Results

- ALIA4J implements pattern evaluation algorithm
- Modified to consider selectivity
- Compared evaluation time to original algorithm

<table>
<thead>
<tr>
<th>TighVNC</th>
<th>LIAM</th>
<th>ANTLR</th>
<th>FreeCol</th>
<th>Tools</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>107%</td>
<td>108%</td>
<td>105%</td>
<td>99%</td>
<td>119%</td>
<td>112%</td>
</tr>
</tbody>
</table>

- Must weight selectivity with evaluation performance!

<table>
<thead>
<tr>
<th>Name</th>
<th>Declaring Class</th>
<th>Modifiers</th>
<th>Return type</th>
<th>Exceptions</th>
<th>Parameters</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>TighVNC</th>
<th>LIAM</th>
<th>ANTLR</th>
<th>FreeCol</th>
<th>Tools</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>29%</td>
<td>38%</td>
<td>28%</td>
<td>27%</td>
<td>33%</td>
<td>36%</td>
</tr>
</tbody>
</table>

Oct. 17, 2010
Additionally Using Index

- Pattern evaluation less than 10% of original algorithm
- Index must be maintained
- Additional effort pays off with fifth pattern evaluation
Conclusions & Future Work

- Optimizing pattern evaluation has potential
  - Pattern evaluation is inherent to AO execution environments
  - Performance significant in dynamic AOP
  - Survey shows opportunities
  - Indexes and search-plan optimization improve performance
- Extend survey to confirm results
- Confirm performance gain in other execution environments
- Make search-plan optimization context-aware (e.g., return type pattern together with “get*” name pattern is very selective)