Dynamic Purity Analysis for Java Programs

Haiying XuChris PickettClark Verbrugge
{hxu31, cpicke, clump}@sable.mcgill.ca

Sable Research Group, McGill University
Montréal, Québec, Canada H3A 2A7

PASTE 2007
June 14, 2007
Outline

1. Introduction and Motivation
2. Static Purity Analysis
3. Dynamic Purity Analysis
4. Experimental Results
5. Memoization
6. Conclusion and Future Work
Roughly, a pure method has *no externally visible side effects*.

Different variations on purity are possible:

- **Sălcianu and Rinard:**
  - *can create, modify and return new objects*

- **Rountev:**
  - *similar, but cannot return a new object*
Why is Method Purity Important?

Artzi, Kiezun, Glasser, Ernst:
- program comprehension
- modelling
- formal verification
- compiler optimization
- memoization
- thread level speculation
- stack allocation
- refactoring
- test input generation
- regression oracle creation
- invariant detection
- specification mining
- program slicing
Why is Method Purity Important?

Artzi, Kiezun, Glasser, Ernst:

- program comprehension
- modelling
- formal verification
- compiler optimization
- **memoization**
- thread level speculation
- stack allocation
- refactoring
- test input generation
- regression oracle creation
- invariant detection
- specification mining
- program slicing
In this work, we:

- Design and implement *dynamic purity analysis*.
- Investigate several different purity definitions.
- Introduce three different dynamic purity metrics.
- Implement memoization as a purity consumer.
1. Introduction and Motivation

2. Static Purity Analysis

3. Dynamic Purity Analysis

4. Experimental Results

5. Memoization

6. Conclusion and Future Work
Consider the classic functional form of purity:

**A method is strongly pure iff it**

- Does not r/w the heap or static data
- Does not perform any synchronization
- Does not invoke any native method
- Does not invoke any impure method
Static Purity Analysis Framework

Soot

Class Files

Jimple

Static Analysis

Add Tags

Attribute Generation

SableVM

Class Files + Attributes

Attribute Parser

Dynamic Metrics

Output
### Static Analysis Results

<table>
<thead>
<tr>
<th>metric</th>
<th>comp</th>
<th>db</th>
<th>jack</th>
<th>javac</th>
<th>jess</th>
<th>mpeg</th>
<th>rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>static method purity</td>
<td>14%</td>
<td>13%</td>
<td>13%</td>
<td>12%</td>
<td>13%</td>
<td>13%</td>
<td>13%</td>
</tr>
<tr>
<td>dynamic method purity</td>
<td>6%</td>
<td>6%</td>
<td>6%</td>
<td>5%</td>
<td>5%</td>
<td>6%</td>
<td>5%</td>
</tr>
<tr>
<td>dynamic invocation purity</td>
<td>≈0%</td>
<td>2%</td>
<td>10%</td>
<td>10%</td>
<td>6%</td>
<td>16%</td>
<td>3%</td>
</tr>
<tr>
<td>dynamic bytecode purity</td>
<td>≈0%</td>
<td>2%</td>
<td>1%</td>
<td>≈0%</td>
<td>≈0%</td>
<td>2%</td>
<td>≈0%</td>
</tr>
</tbody>
</table>

- **Static/dynamic method purity**
  % of reachable/reached methods that are pure

- **Dynamic invocation purity**
  % of all invocations that are pure

- **Dynamic bytecode purity**
  % of bytecode instruction stream contained in a pure method
1 Introduction and Motivation

2 Static Purity Analysis

3 Dynamic Purity Analysis

4 Experimental Results

5 Memoization

6 Conclusion and Future Work
Static purity analysis is hard:
- Implementation is complex
- Whole-program analysis is expensive

Dynamic evaluation tells a different story:
- Static vs. dynamic call graph
- Choice of metrics
- Input dependence
Purity can also depend on method input:

```java
int x;
void foo (boolean b) {
    if (b)
        x = 10;
}
```

If we only ever execute `foo (false)`, `foo` is pure!
Four different kinds of dynamic purity:

- **Strong**: the same as strong static purity
  - no heap or static r/w
  - no calls to impure methods

- **Moderate**:
  - allow object allocation, if the object does not escape
  - allow heap r/w to non-escaping objects
  - allow calls to certain impure methods

- **Weak**:
  - moderate, but no limitations on heap reads

- **Once-Impure**:
  - weak, but no restrictions on the first invocation
```java
class Obj {
    int f;
    public Obj() {
        f = 10;
    }
    Obj bar() {
        Obj o = new Obj();
        return o;
    }
    int foo() { // moderately pure
        Obj o = bar();
        return o.f;
    }
    ...
```
... static int x;

int baz (Obj o) { // weakly pure
    return o.f;
}

int baf (boolean b) { // once-impure for TF+
    if (b) {
        Obj.x = 9 * 6; // write to static field
    }
    return 42;
}
Fairly uniform across benchmarks.
Moderate purity does not improve much—cannot dereference input.
Unpredictable from method purity.
Two different groups appear.
Somewhat predictable from invocation purity. Three different groups appear.
## Sources of Impurity

<table>
<thead>
<tr>
<th>source</th>
<th>comp</th>
<th>db</th>
<th>jack</th>
<th>javac</th>
<th>jess</th>
<th>mpeg</th>
<th>rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUTFIELD</td>
<td>27%</td>
<td>29%</td>
<td>21%</td>
<td>21%</td>
<td>23%</td>
<td>24%</td>
<td>28%</td>
</tr>
<tr>
<td>PUTFIELD+</td>
<td>52%</td>
<td>52%</td>
<td>58%</td>
<td>66%</td>
<td>61%</td>
<td>60%</td>
<td>53%</td>
</tr>
</tbody>
</table>

**method impurity**

<table>
<thead>
<tr>
<th>source</th>
<th>comp</th>
<th>db</th>
<th>jack</th>
<th>javac</th>
<th>jess</th>
<th>mpeg</th>
<th>rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUTFIELD</td>
<td>81%</td>
<td>82%</td>
<td>45%</td>
<td>25%</td>
<td>24%</td>
<td>40%</td>
<td>71%</td>
</tr>
<tr>
<td>PUTFIELD+</td>
<td>19%</td>
<td>17%</td>
<td>37%</td>
<td>58%</td>
<td>19%</td>
<td>60%</td>
<td>28%</td>
</tr>
</tbody>
</table>

**invocation impurity**

<table>
<thead>
<tr>
<th>source</th>
<th>comp</th>
<th>db</th>
<th>jack</th>
<th>javac</th>
<th>jess</th>
<th>mpeg</th>
<th>rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUTFIELD</td>
<td>21%</td>
<td>85%</td>
<td>38%</td>
<td>25%</td>
<td>8%</td>
<td>11%</td>
<td>33%</td>
</tr>
<tr>
<td>PUTFIELD+</td>
<td>79%</td>
<td>13%</td>
<td>48%</td>
<td>66%</td>
<td>45%</td>
<td>89%</td>
<td>66%</td>
</tr>
</tbody>
</table>

**bytecode impurity**

PUTFIELD is the main reason for impurity.
1. Introduction and Motivation
2. Static Purity Analysis
3. Dynamic Purity Analysis
4. Experimental Results
5. Memoization
6. Conclusion and Future Work
Overview of memoization:

- Maps method input to output
- Allows repeat invocations of pure methods to be skipped
- Once-impure purity is a natural fit

How can we use memoization?

- Candidate for optimization
- Good functional sanity test
Memoization Framework

SableVM

Class Files

Purity Analysis

Escape Analysis

offline

online

Class Files + Purity Data

Clients

Memoization

Input:Output Mapping

Argument Lookup

Dynamic Metrics

Output
Factors influencing memoization decisions:

- Method size (50 instructions)
- Input size (100 KB—otherwise potentially the whole heap!)
- Hashtable warm up period (1000 cold start misses)
- Hit ratio (better than 1 in 10)
- Global memory consumption (1 GB)

These are fairly generous limits...
Execution Times

![Bar chart showing execution times for various tasks with different optimization techniques.](chart.png)
Memoization Improvements

Why doesn’t memoization achieve speedup?
- Small number of memoized methods
- Most memoized methods are short
- Usually, less than 1% of bytecode is skipped (best case 9%)
- Implementation limitations

Potential improvements:
- Consider purity on a per-input basis
- Track only those fields read by the method
- Adaptively turn off memoization if no benefit
- Allow for cycles in input data structures
1. Introduction and Motivation
2. Static Purity Analysis
3. Dynamic Purity Analysis
4. Experimental Results
5. Memoization
6. Conclusion and Future Work
Conclusions

Static results correlate weakly with dynamic behaviour

We considered three different metrics:

- method purity varies only slightly
- invocation purity separates benchmarks into two groups
- bytecode purity separates benchmarks into three groups

Consumer applications can impose strong constraints
Future Work

Future work:

- Consider purity at different granularities
- Visualize purity evolution over time
- Support arbitrary kinds of dynamic purity
- Memoization improvements
- Other applications besides memoization (lots!)
  - e.g., speculate past nearly pure methods