A dynamic analysis to explore scopes of programmer reasoning

Presenter: Devin Coughlin
Evan Chang, Amer Diwan, Jeremy Siek
University of Colorado at Boulder
MVD 2010
Explaining the Results of Bug-Finding Tools

“A misunderstood explanation means the error is ignored, or, worse, transmuted into a false positive.”

-- Engler et al. (Coverity)

If programmers don’t understand a warning about bug, they will assume it is a false positive, and they will stop using your tool.
Engler’s “Law”

It is better to *suppress* a warning about a real bug than to give an explanation that the *programmer* won’t understand.
When to suppress?

- When the **chain of causality** is too long for the programmer to **check in their head**.
- How long is too long?
- And how do programmers’ precision and correctness change as “**length**” increases?
When to suppress?

- When the **chain of causality** is too long for the programmer to **check in their head**.
- How long is too long?
- And how do programmers’ precision and correctness change as **“length”** increases?

**Key Challenge:**

Quantifying programmer’s scope of reasoning
Scopes of Reasoning
Scopes of Reasoning

Points-To Analysis
Scopes of Reasoning

Dataflow Analysis
Scopes of Reasoning

Shape Analysis
Scopes of Reasoning

Programmer

The New Yorker
Mar. 29, 1976
Price 75 cents
Investigate

How to investigate programmers’ scope of reasoning?

- Observe their artifacts
  - every program is an opportunity for investigation
  - and we have lots of programs

- Observe them directly
  - but: human subjects experiments are very expensive, so need to know what to look for
Investigate

How to investigate programmers’ scope of reasoning?

- Observe their artifacts
  - every program is an opportunity for investigation
  - and we have lots of programs

- Observe them directly
  - but: human subjects experiments are very expensive, so need to know what to look for
Experimental Methodology

High-level approach:

• Identify operations that could potentially cause runtime errors, but do not.
• Find the “closest” guards against those errors.
• Measure the “distance” between the operation and guard.
Experimental Methodology

High-level approach:

• Identify operations that could potentially cause runtime errors, but do not.
• Find the “closest” guards against those errors.
• Measure the “distance” between the operation and guard.

Premise:

Error-guard distance is a proxy for the programmer’s scope of reasoning.
# Experimental Methodology

Example concrete approach:

<table>
<thead>
<tr>
<th>Runtime Error</th>
<th>Java null pointer exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guards</td>
<td>if and new</td>
</tr>
<tr>
<td>Distance Metric</td>
<td>callstring length</td>
</tr>
</tbody>
</table>
Experimental Methodology

Example concrete approach:

<table>
<thead>
<tr>
<th>Runtime Error</th>
<th>Java null pointer exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guards</td>
<td>if and new</td>
</tr>
<tr>
<td>Distance Metric</td>
<td>callstring length</td>
</tr>
</tbody>
</table>

`callstring length`: roughly, the **level of inlining** required to bring value flow from the guard to the potential runtime error all into the **same function**.
Proposed Protocol

1. **Original Program**
2. **Instrument**
   - Instrumented Program
3. **Run**
   - Trace
4. **Interpret**
   - Event Stream
5. **Interpret**
   - Error-Guard Distances
6. **Summarize**
   - Scope of Reasoning
Instrument and Emit Trace

Instrument bytecode (java.lang.Instrument, ASM-3.2, JVMTI) to emit an error-guard specific trace as bytecode runs.

**Java Bytecode**
- static
- messy
- stack-based
- includes control flow
- includes ints, floats, etc. as values

**Reference Only Trace**
- dynamic
- clean(er)
- still stack-based
- no control flow
- references are the only values
Interpret Trace and Events

Trace Interpreter delegates to Event Interpreter

- **tedious** mechanics
- operand, method stacks
- parameter passing
- instrumentation handoff
- abstract value tracking

- value creation, copying
- value transformation, based on event
- only events of interest
- changes from experiment to experiment
# Interpret Trace and Events

<table>
<thead>
<tr>
<th>Trace Interpreter</th>
<th>delegates to</th>
<th>Event Interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>• <strong>tedious</strong> mechanics</td>
<td></td>
<td>• value creation, copying</td>
</tr>
<tr>
<td>• operand, method stacks</td>
<td></td>
<td>• value transformation, based on event</td>
</tr>
<tr>
<td>• parameter passing</td>
<td></td>
<td>• <strong>only</strong> events of interest</td>
</tr>
<tr>
<td>• instrumentation handoff</td>
<td></td>
<td>• changes from experiment to experiment</td>
</tr>
<tr>
<td>• abstract value tracking</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We can supply different event interpreters to experiment with notions of **distance** and **scope**.
Abstract values are pairs \((h_{\text{min}}, h_{\text{max}})\) of stack heights.

<table>
<thead>
<tr>
<th>Event</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>null check</td>
<td>(h_{\text{min}} = h_{\text{current}})</td>
</tr>
<tr>
<td></td>
<td>(h_{\text{max}} = h_{\text{current}})</td>
</tr>
<tr>
<td>instantiation</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>(h_{\text{max}} = \max(h_{\text{current}}, h_{\text{max}}))</td>
</tr>
<tr>
<td>call return</td>
<td>(h_{\text{min}} = \min(h_{\text{current}}, h_{\text{min}}))</td>
</tr>
<tr>
<td>dereference</td>
<td>(\text{callstring length} = h_{\text{max}} - h_{\text{min}})</td>
</tr>
</tbody>
</table>
Callstring Behavior
Callstring Behavior

92% of dereference sites at $k = 6$ or less
A Modular Framework for Investigation

<table>
<thead>
<tr>
<th>Dynamic Instrumentation</th>
<th>Completely Reusable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace Interpreter</td>
<td>Error/Guard Specific</td>
</tr>
<tr>
<td>Event Interpreter</td>
<td>Distance Metric Specific</td>
</tr>
</tbody>
</table>
Ongoing Work

- More **programs**
  - large vs. small
  - single vs. multiple programmers
- More **errors** and **guards**
- Array bounds checking
- More distance **metrics**
  - Look at package/class/method?
  - Look at public/protected/private?
  - Borrow from heap abstraction community?
  - Extend Ko’s notion of **familiarity**[^1]?


16
Open Questions

• Does observing a program really tell us about the programmer’s scope of reasoning?
  • How can we validate this approach?
• We observe bytecode rather than source. What is lost in that lowering?
• Is the dynamic analysis approach appropriate?
  • What could we do with static analysis?
Open Questions

• Does observing a program *really* tell us about the programmer’s scope of reasoning?
  • How can we **validate** this approach?

• We observe **bytecode** rather than **source**. What is lost in that lowering?

• Is the **dynamic** analysis approach appropriate?

• What could we do with **static** analysis?

Other Questions?