Improving the Static Resolution of Dynamic Java Features

Jason Sawin
Ohio State University
Explaining My Research To Dad
Code is Like a Recipe

Chef Justin’s
Prawn Provencale
at Park Plaza Garden’s
(Yield: 1 serving)

4 jumbo prawns
1 tablespoon minced garlic
1 tablespoon olive oil
2 tablespoons minced red onion
3 tablespoons chopped fresh tomatoes
Juice of 1/2 fresh lemon
3 ounces chicken stock

1 teaspoon julienned basil, or more to taste
1 teaspoon chopped Italian parsley
Pinch of crushed pepper
2 tablespoons butter
Salt and pepper to taste
3 large crostinis

1. Sauté prawns with garlic, olive oil, onion and tomatoes, until shrimp is golden.
2. Add lemon juice, chicken stock, basil, parsley, crushed red pepper, butter and salt and pepper to taste. Let simmer to reduce to sauce consistency.
3. To serve, position three large crostinis on a plate. Place prawns in the center. Pour sauce mixture on top.

Nutritional Information per serving
Calories ________ 411
Fat ________ 28g (15 sat.)
Carbohydrate ________ 8g
Cholesterol ________ 99mg
Sodium ________ 807mg
Protein ________ 9g

Savoring Secrets
Dynamic Recipe?
Dynamic Features of Java

• Dynamic Class Loading
  – Ability to install classes at run time

• Reflection
  – Ability to examine or modify the run-time behavior of a running applications

• JVM
  – Implicitly calls certain code elements

• Native Method
  – Ability to interface with libraries written in non-Java languages
Common Uses of Dynamic Features

• Many large applications support plug-ins
  – Eclipse, columba, Apache Tomcat

• On startup they read specification files or look for class files in specific directories

• Dynamic class loading loads the found plug-ins

• Reflection can be used to access the members of the loaded classes
Dynamic Features In Action

Class c;  String className; Method m; Object h;
.....
Class c = Class.forName(className);
m = c.getMethod("handle", ...);
h = c.newInstance();
m.invoke(h,...)
.....
Static Analysis: Not just Counting Eggs

• Analyzes static representations of programs
  – No execution
• Key component of tools for: program understanding, program transformation, testing, debugging, performance optimizations ...
• Dynamic features pose a significant challenge to static analyses
Traditional Approaches

• Ignore it
  – Do not model the implicit actions of dynamic features
  – Results will be unsound for any application that uses dynamic features

• Overly conservative
  – Assume that every applicable entity can be a potential target of dynamic feature constructs
  – All relevant information is obfuscated by infeasible interactions

• Query the user
Modern Approaches

• String analysis
  – `Class.forName(<string>)`

• Utilize casting information
  – `z = (Foo)clazz.newInstance()`

• Aid analysis user through specification points that identify where string values flow from

• Livshits *APLAS 2005*
Problem 1

• **Problem**: string analysis only considers purely static string values
  – Many uses of dynamic features will rely on string values which are not hard coded into application

• **Insight**: hybrid or “semi-static” analysis can increase information being considered
  – Increases precision
  – Loss of soundness
Problem 2

- **Problem**: state-of-the-art string analysis is not powerful enough to precisely model all language features

- **Insight**: extensions can be made that improve the precision of string analysis
Problem 3

• **Problem**: precise string analysis is costly in both time and memory

• **Insight 1**: parallelizing string analysis allows it to leverage modern multi-core architectures

• **Insight 2**: extensions can be made that can increase the efficiency of string analysis
  – Identify and remove irrelevant information
Problem 4

• **Problem**: CHA call graph construction algorithm must provide treatment for dynamic features
  • Essential component of *many* static analyses

• **Insight**: Treatments require consideration of the assumptions
  • Incorporate string analysis to aid in precise resolution of dynamic features
  • Less precise treatments for unresolved instances
Outline

• Background
• Extensions to String Analysis
• Increasing the Scalability of String Analysis
• CHA and Dynamic Features
Dynamic Class Loading In Java Libraries

• Extended state-of-the-art string analysis for Java: Java String Analyzer (JSA)
  – Publicly available implementation
  – Semi-static extension
  – Precision improving extensions

• Investigated dynamic class loading sites in Java 1.4 standard libraries
  – Used by all Java applications
Semi-Static Extension

• At analysis time dynamically gather information about the values of system **environment variables**
  – Typically they inform applications about the environment in which they are executing
  – Inject this information at environment variable entry points
    – `System.getProperty(<String>)`
    – Use JSA to resolve `<String>`

• Challenges: default values and multiple key values
  – `System.getProperty(<String>,<String>)`
private static final String handlerPropName = "sun.awt.exception.handler";
private static String handlerClassName = null;

private boolean handleException(Throwable thrown) {
    handlerClassName = ((String) AccessController.doPrivileged(
        new GetPropertyAction(handlerPropName)));
    ....
    Class c = Class.forName(handlerClassName);
    ....
}
Increasing the Precision of JSA

• Field Extension: JSA provides no treatment for fields
  – We provide a context-insensitive treatment for fields
    • Currently only considering private fields of type String and specific instances of array fields of type String

• Type Extension: String is a subclass of Object
  – Use type inferencing analysis to determine objects of static type Object which are actually objects of type String
  – Needed for calls to doPrivileged
Overview of Experiment Evaluation

• Part 1: manual investigation
  – Establish a “perfect” baseline
  – Gain insights to the validity of our approach

• Part 2: evaluation of semi-static JSA
  – Compare to perfect baseline
  – Compare to current state-of-the-art
Part 1: Manual Investigation Results

• Dynamic class loading sites were categorized into 3 groups
  – **Static dependent (SD)** – The target string depended on only static string values
  – **Environment dependent (EVD)** – The target string depended on values returned from environment variable entry point
  – **Other dependent (OD)** – The target string flowed from sources such as files or directory structures

<table>
<thead>
<tr>
<th></th>
<th>SD</th>
<th>EVD</th>
<th>OD</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual Inv</td>
<td>33</td>
<td>35</td>
<td>12</td>
<td>80</td>
</tr>
</tbody>
</table>
Part 2: Experimental Results

![Bar chart showing resolved sites for JSA versions JSA1, JSA2, JSA3, and JSA4. The chart compares SD (light yellow) and EVD (dark blue) versions.]

- JSA1: 22 SD, 0 EVD
- JSA2: 22 SD, 16 EVD
- JSA3: 28 SD, 22 EVD
- JSA4: 27 SD, 32 EVD
Summary of Initial Study

• Hybrid and modeling extensions greatly increase JSA ability to resolve instances of dynamic class loading
  – Very practical relaxation of assumptions
  – Java Library: 40% of client-independent dynamic class loading sites depend upon the values of environment variables
  – Resolved 2.6 times more sites than unenhanced JSA

• JSA scaled well to package sized inputs but performance degraded for larger inputs
Outline

• Background
• Extensions to String Analysis
• Increasing the Scalability of String Analysis
• CHA and Dynamic Features
JSA Design

• **Front-end**: creates a graph that abstractly represents the flow of string values through the input classes
  – Nodes: string operations
  – Edges: def/use relationship

• **Back-end**: Builds a context-free grammar from the flow graph and ultimately generates a finite state automaton for each hotspot
Baseline Testing

- Executed JSA on 25 benchmark applications
- For 18 benchmarks the front-end required more resources than the back-end
- For 3 benchmarks JSA exhausted a 6GB heap and failed to complete
Front-end Design

Diagram:

1. App. Classes
2. Make Intraprocedural CFGs
3. Liveness Analysis
4. Alias Analysis
5. Reaching Definitions Analysis
6. Interprocedural Flow Graph
7. Simplify Flow Graph
8. BACKEND
Parallel Front-end Design
Parallel Front-end Time Results

Benchmarks

- Buoy
- Compress
- DB
- Fractal
- GattMath
- Jack
- Javac
- JavaCup
- Jb61
- JFlex
- JLex
- Jwps
- Jtar
- Mindterm
- MpegAudio
- Muffin
- Rabbit
- Sablecc
- Sockecho
- Sockproxy
- VietPad
- Violet

Time (in ms)

- JSA
- 1 Thread
- 2 Threads
- 3 Threads
- 4 Threads
Concat Simplification

Pass

Jason

Pass Jason
Remove Superfluous Information

Anger is brief madness

hotspot
Propagating “anystring” Value

Don’t think just anystring do
Total Time With New Simplifications

![Bar Chart]

- **Time (in ms)**
- **Benchmarks**: Buoy, Compress, DB, Fractal, GattMath, Jack, Javac, JavaCup, Jb61, JFlex, JLex, Jtar, Mindterm, MpegAudio, Rabbit, Sockecho, Sockproxy, VietPad, Violet

**X.XX Speedup**
- JSA
- New Simps

Values:
- Buoy: 1.18, 1.04, 1.25, 1.00, 1.37
- Compress: 1.18, 1.04, 1.25, 1.00, 1.37
- DB: 1.18, 1.04, 1.25, 1.00, 1.37
- Fractal: 1.18, 1.04, 1.25, 1.00, 1.37
- GattMath: 1.18, 1.04, 1.25, 1.00, 1.37
- Jack: 1.18, 1.04, 1.25, 1.00, 1.37
- Javac: 1.18, 1.04, 1.25, 1.00, 1.37
- JavaCup: 1.18, 1.04, 1.25, 1.00, 1.37
- Jb61: 1.18, 1.04, 1.25, 1.00, 1.37
- JFlex: 1.18, 1.04, 1.25, 1.00, 1.37
- JLex: 1.18, 1.04, 1.25, 1.00, 1.37
- Jtar: 1.18, 1.04, 1.25, 1.00, 1.37
- Mindterm: 1.18, 1.04, 1.25, 1.00, 1.37
- MpegAudio: 1.18, 1.04, 1.25, 1.00, 1.37
- Rabbit: 1.18, 1.04, 1.25, 1.00, 1.37
- Sockecho: 1.18, 1.04, 1.25, 1.00, 1.37
- Sockproxy: 1.18, 1.04, 1.25, 1.00, 1.37
Time Results Cont

Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Jpws</th>
<th>Muffin</th>
<th>Sablecc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (in ms)</td>
<td>230.56</td>
<td>184.98</td>
<td>1.08</td>
</tr>
</tbody>
</table>

X.XX Speedup
- JSA
- New Simps
Missing 3 Benchmarks

- New simplifications allowed JSA to complete an analysis of the previously unanalyzed benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Front-end (4 threads)</th>
<th>Back-end</th>
<th>Total Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEdit</td>
<td>6053</td>
<td>1021</td>
<td>7074</td>
</tr>
<tr>
<td>Jess</td>
<td>1350</td>
<td>457</td>
<td>1807</td>
</tr>
<tr>
<td>JGap</td>
<td>1873</td>
<td>556</td>
<td>2423</td>
</tr>
</tbody>
</table>
Summary

- For 22 benchmarks the parallel version of JSA’s front-end achieved an average speedup of 1.54 and reduced the average memory used by 43%
- New graph simplifications achieved a speedup of over 180 for 2 benchmarks and allow JSA to complete an analysis of 3 benchmarks that previously exhausted the heap memory
- These improvements make it more practical to incorporate JSA into analysis frameworks
Outline

• Background
• Extensions to String Analysis
• Increasing the Scalability of String Analysis
• CHA and Dynamic Features
CHA Call Graph Construction Algorithm

• Class Hierarchy Analysis (CHA)
  – For every virtual call site $e.m(...)$ where $T$ is the static type of $e$, it examines all subtypes of $T$ for methods which override $m(...)$
  – The set of overriding methods are considered possible targets of the call

• Used by a wide range of interprocedural static analyses

• Problems with dynamic features
CHA and Dynamic Features

• Every implementation of CHA makes assumptions about dynamic features
  – Wide range of possible assumptions
    • Very conservative to unsound
• Different assumptions allow for different resolution techniques
  – String analysis
  – Cast information
• First investigation of different sources of unsoundness
Assumption Hierarchy Overview

- Semi-static Strings
  - Correct Strings
    - Correct Casting
      - Encapsulation Safe
        - Fully Conservative
Conservative Treatment

• Fully Conservative
  – Every class could be loaded at *dynamic class loading* sites
  – Every concrete class could be instantiated at calls of the
    from `Class.newInstance` and `Constructor.newInstance`
  – All methods could be implicit targets of `Method.invoke`
    calls
  – All methods could be called by *native methods*
Encapsulation Safe

• Assume that dynamic features will respect encapsulation
  – Every class could be loaded at dynamic class loading sites
  – Calls `Method.invoke` and `newInstance` will respect encapsulation
  – All public methods could be called by `native methods`
Correct Cast Information

• **New assumption**: casts will not generate run-time exceptions
• Can use cast information to resolve instances of dynamic class loading, reflective invocation and instantiation
  – Requires a post-dominance analysis and a reaching definitions analysis
• Unresolved dynamic features are treated in an encapsulation safe manner
Correct String Information

- **New assumption**: dynamic features will not affect the value of *private String* fields and formal *String* type parameters of *private* and *package-private* methods whose values flow to dynamic class loading sites
  - Allows JSA to be used to resolve instances of dynamic class loading
  - The type information from resolved instances is propagated to reflective calls
Semi-static String Values

- **New assumption**: values of environment variables which can affect dynamic class loading sites will be the same at analysis time and run time
  - Allows JSA to consider semi-static string values
Experimental Evaluation

• Implemented a version of the CHA call graph construction algorithm for each level of the assumption hierarchy
• Applied each implementation to 10 benchmark applications
• Compared the number of nodes and edges
Experimental Results: Nodes

- On average *Semi* contains 10% fewer nodes than *Cons*
Experimental Results: Edges

- On average *Semi* contains 54% fewer edges than *Cons*
Summary of Results

• The semi-static version of CHA created a graphs that contained, on average, 10% fewer nodes and 54% fewer edges than the fully conservative version

• Semi-static version was able to resolve an average of 6% of the reflective invocation calls, 50% of the dynamic class loading sites, and 61% of the reflective instantiation sites

• Under very reasonable assumptions a much more precise call graph can be created
Summary of Results Cont.

• We also compared Semi to the CHA in Soot
  – Soot provides “conservative” treatment for calls to
    `Class.forName` and `Class.newInstance`
• Soot’s graphs contained an average of 37% fewer
  nodes and 62% fewer edges than those created by
  Semi
• Significant portions of a call graph could be missing if
  dynamic features are treated unsoundly
Conclusion

• Increased the modeling capabilities of string analysis
  – Semi-static and modeling extensions
• Decreased the cost of precise string analysis
  – Parallel design and removing irrelevant information
• Increased a CHA call graph construction algorithm’s ability to precisely treat instances of dynamic features.
• This work is a step toward making static analysis tools better equipped to handle dynamic features of Java
Future work

• Explore other semi-static sources of string values
  – Configuration files

• Further increase JSA scalability
  – Library summaries
  – Object pools

• Refine resolution techniques
  – String analysis to resolve `Class.getMethod` calls

• Apply resolution techniques to other analyses
  – RTA