ILEA: Inter-Language Analysis across Java and C

Gang Tan, Boston College
Greg Morrisett, Harvard University
Multi-Lingual Programming

- Software components developed in different programming languages
  - e.g., Java GUI + C back end
  - e.g., Perl web scripts + Java middleware
- Reuse legacy code; mix-and-match benefits of different languages
Pure Java?

- `java.io.FileInputStream`
  - A Java wrapper for C code that invokes system calls
  - Java and C interacts through the Java Native Interface (JNI)

- `java.util.zip.*`
  - Java wrappers that invoke the Zlib C compression/decompression library
Pure Java? Not Really …

Java

C/C++

700,000 loc in JDK API 1.5
Static Analysis on Multi-Lingual Applications

- Static analysis: optimization; bug finding
- Most existing source-code analyses are limited a priori to code written in a single language
- Extending the horizon of analysis:

<table>
<thead>
<tr>
<th></th>
<th>Intra-procedural analysis</th>
<th>Inter-procedural analysis</th>
<th>Inter-language analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Within a same procedure</td>
<td>Across the procedure boundaries</td>
<td>Across the language boundaries</td>
</tr>
</tbody>
</table>
Bug Finding Across Languages

- Example: Null dereferences

```java
String s = c_fun();
int len = s.length();
```

```
jstring c_fun ( ...) {
    if (…) {
        …
        return NULL;
    }
}
```

$s$ might be null

In security-critical code, high precision is needed
Our Goal

- How can we extend the horizon of existing Java analyses to cover C code?

- Basic approach
  - Build a specification of the C code
  - Extend the Java analysis to understand the specification

- Questions:
  - What specification language would be the best for all possible analyses?
  - How to generate specifications?
Some annotation language for a particular analysis

- Nonnull annotations:
  \[
  \text{jstring c\_fun ( object obj)}
  \]

- Immutability annotations [Pechtchanski and Sarkar 02]
- Annotation language by [Guyer and Lin 99]

+ Very useful for a particular analysis
- Any simple set of annotations will be incomplete
What Specification Language?

- Pre- and post-conditions expressed in some logic
  - ESC, Spec#, Bandera, ...
    - \{obj \neq null\}
    - jstring c_fun (jobject obj)
      - \{ret \neq null\}

+ Can provide a range of specifications
- Teaching existing Java analysis to understand the specification requires a significant effort
Our Specification Language

- A slightly extended Java as the specification language for C
  - Easy to teach existing Java analyses to understand it
    - They already understand Java
  - Possible to build completely accurate model for some C code
    - If it can be faithfully compiled to Java

- A few primitives for approximation
Return a random value of type $\tau$

Example

- A C function that invokes gettimeofday() and allocates a Java object of type
  
  ```
  Time {int sec; int msec;}
  ```

- To model the C function
  
  ```
  new Time (choose(int), choose(int));
  ```
**mutate(x:object)**

- **Perform some (type-preserving) mutation to the object x**

- **Example**
  
  A C function that mutates existing objects of any type but does not allocate
  
  ```c
  for (int i = 0; i < choose(int); i++) {
      Object x = choose(Object);
      mutate(x);
  }
  ```
Extended JVML

- Formalized the semantics of the new instructions based on the JVML$_f$ model [Freund and Mitchell 03]
  - choose $\tau$, mutate
  - top: may have any type-preserving effect on the JVM heap
- The new instructions are non-deterministic
  - They relate one state to possibly multiple states
Revisiting Our Questions

- What specification language would be the best for all possible analyses?
  - JVML + a few primitives for approximation
    - Expressive
    - Easy to migrate existing Java analysis

- How to generate specifications?
  - Automatic specification extractors
Specification Extractors

Our spec. language allows a range of specification extractors

Our spec. extractor

- Implemented on top of the CIL (Necula et al.)
  - Use a pointer analysis provided by the CIL
  - Output is in Jasmin syntax
Our Specification Extractor

- **Strategy:** make common and easy cases precise, while leaving rare or difficult cases for approximation.

- **Support:** loops; pointers; structs; unions; most of the JNI API functions; …
  - Thanks to the approximation instructions

- **Assumptions**
  - Faithfully capture the JVM-heap effect that the C code might have
  - Single-threaded code

- **Formalization**
  - The paper formalized a subset and presented theorems
Specification Extraction: Example I

```c
int c-fun (int i) {
    int j;
    int *p = &i;
    if (*p > 0) j=i;
    else j = 2*i;
    return j;
}
```

Type-based variable mapping

```c
int c-fun (int i) {
    int j;
    if (choose boolean)
        j=i;
    else j=2*i;
    return j;
}
Spec.
```
void addOne (jintArray arr) {
    int len = GetArrayLength(arr);
    int * buf = malloc(len * sizeof(int));
    int i;
    GetIntArrayRegion(arr, 0, len, buf);
    for (i = 0; i < len; i++) {
        buf[i] += 1;
    }
    SetIntArrayRegion(arr, 0, len, buf);
    free(buf);
}
void addOne(jintArray arr) {
    int len = GetArrayLength(arr);
    int * buf = malloc(len * sizeof(int));
    int i;
    GetIntArrayRegion(arr, 0, len, buf);
    for (i = 0; i < len; i++) {
        buf[i] += 1;
    }
    SetIntArrayRegion(arr, 0, len, buf);
    free(buf);
}

void sumArray (int[ ] arr) {
    int len=arr.length();
    int i;
    for (i=0; i<len; i++) {
        mutate(arr);
    }
    SetIntArrayRegion(arr, 0, len, buf);
}

C code

Spec.
Utilizing Specifications

- **Thesis**
  - With small changes, existing Java analysis can understand the behavior of C code

- **Experiment**
  - Jlint: a Java bug finder
    - perform inter-procedural dataflow analysis
  - Finding null dereferences
  - JNI-based C code is prone to the error of null dereferences
Setup of Our Experiment

1. JNI-based C Code
2. Specification Extractor
3. Extended JVML Spec.
4. Java Code
5. Augmented Jlint
6. Warnings
## Preliminary Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>C loc</th>
<th>Java loc</th>
<th>Time</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.StrictMath</td>
<td>8658</td>
<td>1128</td>
<td>0.69s</td>
<td>0</td>
</tr>
<tr>
<td>java.util.zip</td>
<td>9195</td>
<td>824</td>
<td>0.79s</td>
<td>0</td>
</tr>
<tr>
<td>posix 1.0</td>
<td>1874</td>
<td>860</td>
<td>0.24s</td>
<td>15</td>
</tr>
<tr>
<td>libreadline-java-0.8.0</td>
<td>1810</td>
<td>1196</td>
<td>0.13s</td>
<td>9</td>
</tr>
</tbody>
</table>

* Errors = Warnings - False positives
Related Work

- Java bug finders and optimizers
  - FindBugs, Jlint, ESC/Java, Bandera, SOOT, Jikes, …

- Analysis of multilingual software
  - Saffire [Furr and Foster 05, 06]
  - SafeJNI [Tan et al. 06]
  - Interaction between statically and dynamically typed languages
    - [Gray et al. 05] [Matthews and Findler 07]
  - Better interfaces between Java and C
    - Jeannie [Hirzel and Grimm 07]
    - Janet [Budak et al. 00]

- .NET, Fortify
  - Heavy-weight
The End