Checking Type Safety of Foreign Function Calls

Jeffrey S. Foster
University of Maryland, College Park

Joint work with Michael Furr

Introduction

• Many high-level languages contain a foreign function interface (FFI)
  - OCaml, Java, SML, Haskell, COM, SOM, ...
  - Allows access to functions written in other languages

• Lots of reasons to use them
  - Gives access to system calls
  - Other legacy libraries may be infeasible to port
  - Performance
  - Suitability of language for particular problem

Dangers of FFIs

• In most FFIs, programmers write “glue code”
  - Translates data between host and foreign languages
  - Typically written in one of the languages

• Unfortunately, FFIs are often easy to misuse
  - Little or no checking done at language boundary
  - Mistakes can silently corrupt memory
  - One solution: interface generators
    - But there’s still lots of hand-written code around

This Work

Static type checking for FFI programs

• Targets: OCaml-to-C FFI and the JNI

• Analysis focuses on C glue code
  - Goal: infer what types glue code thinks it’s using
SAFFIRE

- Static Analysis of Foreign Function Interfaces
  - Pair of tools, one for each FFI
  - Detected many errors on a suite of programs

- Key design point: Only as complex as necessary
  - FFI glue code is messy
    - ...but not all that complicated (to avoid mistakes!)
  - We can use fairly simple analysis in surprising places
    - E.g., to track values of integers and strings

The OCaml FFI

- OCaml:
  
  external ml_foo : int -> int list -> unit = "c_foo"

- C:
  
  typedef long value;
  value c_foo(value int_arg, value int_list_arg);

  - All OCaml types conflated to value
    - Can be a primitive (int, unit) or a pointer (int list)
  - No checking that value is used at the right OCaml type

Type Tags

- Unboxed data (e.g., int) has low bit set to 1
- Boxed data (e.g., int list) stored in structured block
  - Is_long() macro to test low-order bit

```
type t =
  A of int
| B
| C of int * int
| D
```

Primitive Types

- Need to bit shift ints to convert to or from C
  - Val_int() and Int_val() macros available
    - Can you guess which is which?
    - Worse: Can apply either to a pointer
      - Since value is a typedef of long
  - Primitives of different types have same rep.
    - 0 : int = B = unit
Structured Blocks

- Pointer arithmetic to access fields and tags
  - Field(x, i) = *((value *) x + i) – read ith field of x
  - Tag_val() – read tag in header (tuple, rec tag is 0)
  - Can be applied to anything! (See cast above)

- Again, different types have same representation
  - Could be int * int * int
  - Could be Foo of type t' = Foo of int * int * int | ...

Representational Types

- Types to model C's view of OCaml data
  - # of nullary constructors
  - arg types of other constructors
  - mt ::= (C, S)
  - S ::= σ | P + S | ε
  - P ::= π | mt × P | ε

Examples:

- int ⇒ (∞, ε)
- int * int ⇒ (0, (∞,0)×(∞,0) + ε)
- type t = A of int | B | C of int * int | D
  - ⇒ (2, (∞,0) + (∞,0)×(∞,0) + ε)

Example: “Pattern Matching”

```c
if (Is_long(x)) {
  if (Int_val(x) == 0) /* B */
    ...
  if (Int_val(x) == 1) /* D */
    ...
} else {
  if (Tag_val(x) == 0) /* A */
    Field(x, 0) = Val_int(0)
  if (Tag_val(x) == 1) /* C */
    Field(x, 1) = Val_int(0)
}
```

Tracking OCaml Types through C

- Extend the C type value to
  - boxed or unboxed

(C, S) value {B,T}

- Representational Type
- value (if int)
- block tag (if ptr)

(C, S) flow-insensitive (a value has one OCaml type)

B,T flow-sensitive (vary by program point)
- These may also be Top if unknown
Inferring Sum Types

Example Type Rules

- Type rules map C expressions to extended types
  - Includes additional information on pointer offsets

Example Type Rules (cont’d)

- Flow-sensitivity with type env on “both sides"
  - $A \vdash s; A’$
    - $A$ is original environment
    - $A’$ is environment after $s$ executes
  - Map $G$ from source labels to environments, for branches

Inferring Sum Types
**Key Features**

- In C, all OCaml data has one type
  - And different OCaml types have same representation
  - Need to track integer and pointer offsets to understand C code
  - Combination of constraints and dataflow analysis

- OCaml also has GC
  - C code needs to register its pointers to OCaml heap
    - Easy to forget to do
  - Our system includes some checks for this

**The JNI**

- Several similarities to OCaml FFI
  - All Java objects conflated to one C type
  - C code has richer view of Java data than Java
    - Writing glue code similar to using Java reflection

- Key differences
  - Can only access Java data via function calls
    - No low-level macros available
  - JNI uses strings to identify fields, classes, methods
  - Polymorphism very important in JNI code

**Example JNI Code**

- Java:
  ```java
  Class Foo {
    int x;
    private native void bar(Foo);
  }
  
  void Java_Foo_bar(jobject obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls,"x","I");
    int y = GetIntField(obj,fid);
    ...
  }
  ```

- C:
  ```c
  void Java_Foo_bar(jobject obj) { 
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls,"x","I");
    int y = GetIntField(obj,fid);
    ...
  }
  ```

- Java:
  ```java
  Class Foo {
    int x;
    private native void bar(Foo);
  }
  
  void Java_Foo_bar(jobject obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls,"x","I");
    int y = GetIntField(obj,fid);
    ...
  }
  ```

- C:
  ```c
  void Java_Foo_bar(jobject obj) { 
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls,"x","I");
    int y = GetIntField(obj,fid);
    ...
  }
  ```
Example JNI Code

- **Java:**
  ```java
  Class Foo {
    int x;
    private native void bar(Foo);
  }
  ```

- **C:**
  ```c
  void Java_Foo_bar(jobject obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, "x", "I");
    int y = GetIntField(obj, fid);
    ...  
  }
  ```

Example JNI Code

- **Java:**
  ```java
  Class Foo {
    int x;
    private native void bar(Foo);
  }
  ```

- **C:**
  ```c
  void Java_Foo_bar(jobject obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, "x", "I");
    int y = GetIntField(obj, fid);
    ...  
  }
  ```

Example JNI Code

- **Java:**
  ```java
  Class Foo {
    int x;
    private native void bar(Foo);
  }
  ```

- **C:**
  ```c
  void Java_Foo_bar(jobject obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, "x", "I");
    int y = GetIntField(obj, fid);
    ...  
  }
  ```

Example JNI Code

- **Java:**
  ```java
  Class Foo {
    int x;
    private native void bar(Foo);
  }
  ```

- **C:**
  ```c
  void Java_Foo_bar(jobject obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, "x", "I");
    int y = GetIntField(obj, fid);
    ...  
  }
  ```
Example JNI Code

- Java:
  ```java
  class Foo {
    int x;
    private native void bar(Foo);
  }
  ```

- C:
  ```c
  void Java_Foo_bar(jobject obj) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, "x", "I");
    int y = GetIntField(obj, fid);
    ... 
  }
  ```

Types must match!

Representational Types for the JNI

- Name of the class
  ```
  s ::= "Str" | v
  ```

- List of fields
  ```
  F ::= Φ | s:jt, F | ε
  ```

- List of methods
  ```
  M ::= μ | s: (jt × ... × jt → jt), M | ε
  ```

- Example
  ```
  foo ⇒ {
    "Foo";
    "x" : int;
    "bar" : ({"Foo"} → void)
  }
  ```

Tracking Java Types through C

- Extend the C type `jobject` to `jt jobject`
  - No need for flow-sensitivity, unlike OCaml FFI

- Also track string values in C
  - Assign `char *`s the type `str{s}`
  - Ex: "foo" : str{"foo"}
  - Ex: `void bar(char *x); x : str{v}
    - String value not yet known

Two Other Java Types

- Instances of `java.lang.Class` are important in JNI
  ```
  jt ::= ... | jt Class
  ```

  - A `Class` instance representing the class of `jt`
    - `GetObjectClass : (v:φ; μ) jobject → (v:φ; μ) Class jobject`

- Sometimes we don’t know a string’s value yet
  - So we don’t know what Java class it corresponds to
  ```
  jt ::= ... | String(s)
  ```

  - An object of class `s`
    - `FindClass : str{v} → String(v) Class jobject`
Wrapper Functions

```c
int my_getIntField(jobject obj, char *field) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, field, "I");
    return GetIntField(obj,fid);
}
```

- Accepts any object `obj` with int `field`
- Polymorphic in type of `obj` and contents of field
  ```c
  my_getIntField(obj1, "x");
  my_getIntField(obj2, "offset");
  ```
  - String types are singletons, hence contents = type
- These come up often in practice
  - And JNI has >200 functions! Need to treat polymorphically

Example

```c
int my_getIntField(jobject obj, char *field) {
    jobject cls = GetObjectClass(obj);
    jfieldID fid = GetFieldID(cls, field, "I");
    return GetIntField(obj,fid);
}
```

```c
∀ v₁,v₃,μ₃ . {v₃; v₁:int, ...;μ₃} jobject × str{v₁} → int
```

- Second arg is some string `v₁`
- First arg is some object with an int field of name `v₁`
- The function returns an `int`

Key Features

- Java object types conflated to single C type
  - Need to track string values through C to decide what calls to FFI methods are doing
  - Polymorphism important for wrapper functions
- Other features
  - Need to also track field, method ids through C
  - GC not as important
    - Java automatically tracks objects it passes to C

Soundness

- We can prove soundness via standard progress and preservation techniques
  - Proof for slightly restricted version of the systems
- Theorem: If a program is well-typed, then it does not get stuck
  - OCaml or Java data is never used at the wrong type
Implementation

OSaffire Results

<table>
<thead>
<tr>
<th>Program</th>
<th>C loc</th>
<th>OCaml loc</th>
<th>Time</th>
<th>Errs</th>
<th>Warnings</th>
<th>F-Pos</th>
<th>Impe</th>
</tr>
</thead>
<tbody>
<tr>
<td>apsn-1.00</td>
<td>124</td>
<td>156</td>
<td>1.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>camlzip-1.01</td>
<td>139</td>
<td>151</td>
<td>1.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ocaml-mad-0.1.0</td>
<td>139</td>
<td>147</td>
<td>1.5</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ocaml-ssl-0.1.0</td>
<td>187</td>
<td>192</td>
<td>2.2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ocaml-vorbis-0.1.1</td>
<td>1183</td>
<td>2.8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Rpllib-0.12</td>
<td>1401</td>
<td>21</td>
<td>1.7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lablgl-1.00</td>
<td>1586</td>
<td>1357</td>
<td>7.5</td>
<td>5</td>
<td>5</td>
<td>140</td>
<td>20</td>
</tr>
<tr>
<td>cryptokit-1.2</td>
<td>2173</td>
<td>2315</td>
<td>5.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lablgtk-2.2.0</td>
<td>5998</td>
<td>14847</td>
<td>61.3</td>
<td>9</td>
<td>11</td>
<td>74</td>
<td>48</td>
</tr>
<tr>
<td>Total</td>
<td>24</td>
<td>22</td>
<td>214</td>
<td>70</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Time includes compilation

OSaffire Errors

- Type mismatches (19 errors)
  - 5 errors due to Val_int instead of Int_val or reverse
  - 1 due to forgetting that an argument was in an option
  - Others similar

- Remainder are GC errors (didn’t discuss)
  - 3 – Forgetting to register C pointer to ML heap
  - 2 – Forgetting to release a registered pointer

Implementation Details

- OSaffire
  - Type extractor build from camlp4
  - Concretizes abstract types, fully resolves any aliases

- JSaffire
  - Uses wrapper script to capture classpath during build
  - Uses class file parser to get type information

- Both: C analysis built using CIL
OSaffire Warnings

- Forgetting to add unit parameter to C fn
  - OCaml: `external f : int -> unit -> unit = "f"
  - C: `value f(value x);

- Polymorphism abuse
  - OCaml: `type input_channel, output_channel
  - OCaml: `external seek : int -> 'a -> unit = "seek"
  - C: `value seek(value pos, value file);

JSaffire Results

<table>
<thead>
<tr>
<th>Program</th>
<th>C LOC</th>
<th>Java LOC</th>
<th>Time</th>
<th>Errors</th>
<th>Warnings</th>
<th>F-Pos</th>
<th>Imps</th>
</tr>
</thead>
<tbody>
<tr>
<td>libconf-java-2.10.1</td>
<td>1119</td>
<td>679</td>
<td>2.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libglade-java-2.10.1</td>
<td>149</td>
<td>1022</td>
<td>6.9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libgnome-java-2.10.1</td>
<td>5606</td>
<td>5135</td>
<td>17.4</td>
<td>45</td>
<td>9</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>libgtk-java-2.6.2</td>
<td>27095</td>
<td>32955</td>
<td>36.3</td>
<td>74</td>
<td>13</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>libgtkhtml-java-2.6.0</td>
<td>455</td>
<td>729</td>
<td>2.9</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libgtkmm-embedded-java-1.7.0</td>
<td>166</td>
<td>498</td>
<td>3.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libxte-java-0.11.11</td>
<td>437</td>
<td>184</td>
<td>2.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jnetfilter</td>
<td>1113</td>
<td>1569</td>
<td>17.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libreadline-java-0.8.0</td>
<td>1459</td>
<td>324</td>
<td>2.2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pgi-java</td>
<td>10136</td>
<td>123</td>
<td>2.7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pinot-0.4</td>
<td>998</td>
<td>293</td>
<td>1.8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>1174</td>
<td>366</td>
<td>26.5</td>
<td>0</td>
<td>36</td>
<td>44</td>
<td>22</td>
</tr>
</tbody>
</table>

JSaffire Errors

- 68 functions declared with the wrong arity
- 56 C pointers passed when object expected
  - Most result of a software rewrite
  - 17 type mismatches:
    - e.g., `String ≠ byte[]`
- 14 functions named incorrectly
  - Functions must follow a strict convention to be called from Java

OSaffire Imprecision and False Pos.

- Tags and offsets are sometimes Top
-Globals and function pointers
- Polymorphic variants
- Pointer arithmetic disguised as `long` arithmetic
  - `(t^a)v + 1 == (t^a)(v + sizeof(t^a))`
    - OSaffire gets confused
JSaffire Warnings

• 1 malformed Java class string
• 2 incorrect type declarations
  - JNI contains several typedef’s for object (e.g., jstring, jarray)
  - Warn when C function was declared with the wrong type, even when the value was of the right type
• 33 dead C functions
  - C function appeared to implement a certain Java native method, but no native method was defined in the Java class file

JSaffire False Pos. and Imprecision

• 44 false positives
  - C code uses subtyping for Java types
  - Our tool is based on unification, so considered these type errors
• 22 imprecision messages
  - 16 due to unresolved overloading
    - JSaffire didn’t have enough info to find a consistent type
  - 6 passing arguments to JNI functions packed in an array

Conclusion

• FFIs are a useful part of a language

• FFI code is messy
  - But not complicated, hence analyzable

• Saffire: Type checking multi-lingual code
  - The first we know of to check glue code
  - Makes FFIs safer to use