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

**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 | ...`

### Garbage Collection

- C FFI functions need to play nice with the GC
  - Pointers from C to the OCaml heap must be registered
    - Otherwise the OCaml GC may corrupt them
  - Easy to forget to do, especially for indirect calls
  - Difficult to find this error with testing

- When can a GC occur?
  - Any time a C function calls the OCaml runtime
    - E.g., to call a function, to allocate memory, etc.

### Example: “Pattern Matching”

```ocaml
type t =
  A of int | B | C of int * int | ...

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)
}
```

### Example

```ocaml
value bar(value list) {
  CAMLparam1(list);
  CAMLlocal1(temp);
  temp = alloc_tuple(2);
  CAMLreturn(Val_unit);
}
value foo(value arg) {
  bar(arg);
  return(arg);
}
```
Representational Types

- Types to model C’s view of OCaml data

\[
mt ::= (C, S) \\
S ::= \sigma \mid P + S \mid \epsilon \\
P ::= \pi \mid mt \times P \mid \epsilon
\]

Examples:

- \(int \Rightarrow (\infty, \epsilon)\)
- \(int \times int \Rightarrow (0, (\infty,0) \times (\infty,0) + \epsilon)\)
- \(type t = A of int | B | C of int \times int | D \Rightarrow (2, (\infty,0) + (\infty,0) \times (\infty,0) + \epsilon)\)

Tracking OCaml Types through C

- Extend the C type value to \((C, S)\) value

\[B, T\] flow-sensitive (vary by program point)

- These may also be Top if unknown

Inferring Sum Types

\[
\text{if (is_long(x)) } \{ \\
\psi \geq 1 \quad \text{if (Int_val(x) == 0)} \quad \text{/* B */} \\
\psi \geq 2 \quad \text{if (Int_val(x) == 1)} \quad \text{/* D */} \\
\} \quad \text{else } \{ \\
\sigma = \pi + \sigma' \\
\pi = \text{int} \times \pi' \\
\sigma' = \pi'' + \sigma''' \\
\pi'' = \alpha \times \text{int} \times \pi'''
\]

Inferring Sum Types

\[
\psi \geq 1 \\
\psi \geq 2 \\
\sigma = \pi + \sigma' \\
\pi = \text{int} \times \pi' \\
\sigma' = \pi'' + \sigma'' \\
\pi'' = \alpha \times \text{int} \times \pi'''
\]

Solution to constraints:

\[x: (\psi, \sigma) \text{ value}\{\text{Top, Top}\} \]

Compatible the OCaml type

\[type t = \]

\[\begin{array}{l}
A \text{ of int} \\
B \\
C \text{ of int} \times \text{int} \\
D
\end{array}\]

\[\Rightarrow (2, (\infty,0) + (\infty,0) \times (\infty,0) + \epsilon)\]
Example Type Rules

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

\[ A \vdash e : mt \text{ value} \{boxed, n, m\} \]
\[ mt = (C, P_0 + \cdots + P_m + S) \]
\[ P_m = mt_0 \times \cdots \times mt_n \times P \]

\[ A \vdash *e : mt_n \text{ value}\{Top, 0, Top\} \]

Checking Type Safety of Foreign Function Calls

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

\[ A \vdash x : mt \text{ value}\{B,0,T\} \]
\[ A' = A[x \rightarrow mt \text{ value}\{boxed, 0,T\}] \]
\[ A' \leq G(L) \]

\[ A \vdash \text{if unboxed}(x) \text{ then } L, A[x \rightarrow mt \text{ value}\{boxed, 0,T\}] \]

Checking GC Safety

• Algorithm
  ▪ Build a call graph of the C code
  ▪ Let \( f_i \) be a call to \( f \) at line \( i \)
  ▪ Let \( P(f_i) = \) unprotected locals and parameters at call
  ▪ Check: If path from \( f \) to function that may call GC, require \( P(f_i) = 0 \)

\[ \text{foo()} \rightarrow \text{bar()} \rightarrow \text{alloc_tuple()} \]

\[ P(\text{foo}) = \{ \text{arg} \} \quad \text{error: non-empty} \]

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 data is never used at the wrong type
More Features of OCaml

- Type system does not include objects
  - But neither do FFI programs we looked at

- No parametric polymorphism for FFI functions
  - Allow annotation to be added by hand
  - Only needed 4 times

- Polymorphic variants not handled
  - Results in some false positives

OSaffire: Phase 1, OCaml

- Tool built from camlp4 preprocessor
- Analyzes OCaml source and extracts types of foreign functions
  - Concretizes any abstract types in modules
  - Fully resolves all aliases
- Incrementally updates central type repository
  - Seeded with types from standard library
- Result: Type environment fed into Phase 2

OSaffire: Phase 2, C

- Second tool built using CIL
  - This is the tool that issues warnings etc.

- `Int_val()`, `Tag_val()`, etc. recognized using syntactic pattern matching
  - Modified OCaml header file so we can track macros through expansion
  - Tests look a bit more complicated in source, but still easy to identify the cases in practice
More Details

- Warnings for global values
  - Need to register them, but we don’t check for this
  - Not common in practice (10 warnings)
- C has address-of operator &
  - If &x taken for local x, treat like global
- Type casts handled with unsound heuristics
  - Goal: Track C data embedded in OCaml
- Function pointers yield warnings
  - Only added 8 warnings to benchmarks

OSaffire Results

<table>
<thead>
<tr>
<th>Program</th>
<th>C-loc</th>
<th>O-loc</th>
<th>Ext</th>
<th>Time</th>
<th>Err</th>
<th>Wrn</th>
<th>FPos</th>
<th>Imp</th>
</tr>
</thead>
<tbody>
<tr>
<td>apm-1.00</td>
<td>124</td>
<td>156</td>
<td>4</td>
<td>0.01s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>camlzip-1.01</td>
<td>139</td>
<td>820</td>
<td>9</td>
<td>0.01s</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>38</td>
<td>3</td>
<td>0.01s</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ocaml-ssl-0.1.0</td>
<td>187</td>
<td>151</td>
<td>14</td>
<td>0.02s</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ocaml-glpk-0.1.1</td>
<td>305</td>
<td>147</td>
<td>30</td>
<td>0.03s</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pgc-0.5.5</td>
<td>572</td>
<td>192</td>
<td>29</td>
<td>0.02s</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>443</td>
<td>7</td>
<td>0.07s</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fplib-0.12</td>
<td>1401</td>
<td>21</td>
<td>17</td>
<td>0.06s</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>324</td>
<td>0.40s</td>
<td>4</td>
<td>5</td>
<td>140</td>
<td>20</td>
</tr>
<tr>
<td>cryptokit-1.2</td>
<td>2173</td>
<td>2315</td>
<td>24</td>
<td>0.03s</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>1307</td>
<td>3.83s</td>
<td>9</td>
<td>11</td>
<td>74</td>
<td>48</td>
</tr>
</tbody>
</table>

Total 24 22 214 75

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
  - 3 – Forgetting to register C pointer to ML heap
  - 2 – Forgetting to release a registered pointer

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);
OSaffire Imprecision and False Pos.

• Tags and offsets are sometimes Top

• Globals and function pointers

• Polymorphic variants

• Pointer arithmetic disguised as long arithmetic
  - \[(t^v_0)v + 1 = (t^v_0)(v + \text{sizeof}(t^v_0))\]
  - OSaffire gets confused

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);
    ...
}
```
Example JNI Code

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

- C:
  ```c
  void JavaFoo_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 JavaFoo_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 JavaFoo_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 JavaFoo_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

- 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
  ...
  ...
  ```

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

• Accepts any object \texttt{obj} with int field \texttt{field}
  ■ Polymorphic in type of \texttt{obj} and contents of \texttt{field}
    - String types are singletons, hence contents = type
  ■ These come up often in practice
    - And JNI has >200 functions! Need to treat polymorphically

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

Polymorphism via Semiunification

• Generate \textit{instantiation constraints} when function types instantiated
• Solve instantiation constraints using semi-unification (Henglein 1993, Fähndrich et al 2000)
• Undecidable in theory
• Worked well for analyzing C glue code
  ■ Did not encounter non-termination
• In-order traversal allows for fast, straight-forward implementation

Example

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

\( \forall v_1,v_2,v_3. \{v_3; v_1\text{int}, \ldots; v_3\} \text{ jobject } \times \text{ str}\{v_1\} \rightarrow \text{ int} \)

■ Second arg is some string \( v_1 \)
■ First arg is some object with an int field of name \( v_1 \)
■ The function returns an \( \text{ int} \)

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
More Details: JSaffire

- Soundness also provable for JSaffire
  - Well-typed C code does not access Java data at the wrong type
- Same architecture as OSaffire
- Wrapper script captures classpath during build
- Uses class file parser to get type information

JSaffire Results

<table>
<thead>
<tr>
<th>Program</th>
<th>C-loc</th>
<th>J-loc</th>
<th>Ext</th>
<th>Time</th>
<th>Err</th>
<th>Wrn</th>
<th>FP</th>
<th>Imp</th>
</tr>
</thead>
<tbody>
<tr>
<td>libgconf-java-2.10.1</td>
<td>1119</td>
<td>670</td>
<td>93</td>
<td>1.32s</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>libglade-java-2.10.1</td>
<td>149</td>
<td>1022</td>
<td>6</td>
<td>0.64s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>libgnome-2.10.1</td>
<td>5606</td>
<td>5135</td>
<td>599</td>
<td>6.53s</td>
<td>45</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>libgtk-java-2.6.2</td>
<td>27095</td>
<td>32395</td>
<td>3201</td>
<td>1.04s</td>
<td>74</td>
<td>8</td>
<td>36</td>
<td>18</td>
</tr>
<tr>
<td>libgtkhtml-java-2.6.0</td>
<td>455</td>
<td>729</td>
<td>72</td>
<td>0.65s</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libgtkmozembed-java-1.7.0</td>
<td>166</td>
<td>498</td>
<td>23</td>
<td>0.66s</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>36</td>
<td>0.67s</td>
<td>0</td>
<td>26</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jnetfilter</td>
<td>1113</td>
<td>1599</td>
<td>105</td>
<td>5.38s</td>
<td>9</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>17</td>
<td>0.63s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>pgpjava</td>
<td>10136</td>
<td>123</td>
<td>12</td>
<td>1.11s</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>posix1.0</td>
<td>978</td>
<td>293</td>
<td>26</td>
<td>0.70s</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Java Mustang compiler</td>
<td>532k</td>
<td>1974k</td>
<td>2495</td>
<td>630s</td>
<td>1</td>
<td>88</td>
<td>96</td>
<td>2620</td>
</tr>
<tr>
<td>Total</td>
<td>156</td>
<td>124</td>
<td>142</td>
<td>2642</td>
<td></td>
<td></td>
<td></td>
<td></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
- 18 type mismatches:
  - e.g., String ≠ byte[]
- 14 functions named incorrectly
  - Functions must follow a strict convention to be called from Java

JSaffire Warnings

- 1 malformed Java class string
- 13 incorrect type declarations
  - JNI contains several typedef's for jobject (e.g., jstring, jarray)
  - Warn when C function was declared with the wrong type, even when the value was of the right type
- 110 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 Positives

- 140 false positives
  - C code uses subtyping for Java types
  - Our tool is based on unification, so considered these type errors
  - Also due to unifying a Class with a class object
    - Safe, but those are different types in JSaffire

JSaffire Imprecision

- 2642 imprecision messages
  - Vast majority from Mustang
    - The Java compiler does everything possible with the JNI!
  - 36 due to unresolved overloading
    - JSaffire didn’t have enough info to find a consistent type
  - 707 due to using parts of JNI we don’t model
    - E.g., passing arguments to JNI functions in array
  - 115 due to directly manipulating jobject type
  - 1784 due to function pointers

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