Checking Type Safety of Foreign Function Calls

Mike Furr

FFIs

- High level languages have become very popular
  - Most include a foreign function interface (FFI)
- Still rely on C for certain operations:
  - OS system calls / low level libraries usually impossible to call directly
  - Large legacy libraries may be infeasible to reprogram in a new language
  - Performance critical code may require C

Glue Code

- To use an FFI, programmer must write “glue code”
  - Convert data structures and semantics between host and foreign languages
  - Typically only written in one of the languages
- One approach: interface generators (e.g., SWIG)
  - Not as flexible as hand written code
  - Hand written glue code is low level and it is easy to make mistakes
  - Little or no static checking is provided

Saffire

- Static Analysis of Foreign Function InteRfacEs
  - Static type inference for OCaml and Java FFIs
- Most programmer work is done on the C side
  - Concentrate our analysis there
  - Ensure that C code uses high level types correctly
- General C code is rather difficult to analyze
  - However, “glue code” tends to use C in simple ways
Implementation

- OCaml and Java are very different languages
  - Contrast pattern matching vs method dispatch
  - Combined, provide a cross section of FFI designs
- High level approach to checking FFIs quite similar
  - Implementations shared a lot of infrastructure
    - Both built with CIL
  - Type systems specialized to the high level language

OCaml FFI

OCaml:

```ocaml
external ml_fun: int -> int list -> unit = "c_fun"
```

C:

```c
value c_fun(value int_arg, value int_list_arg)...
```

- `value` can be either a primitive (`int`) or a pointer into the heap (`int list`)
- No static checking that `value` is used correctly

OCaml Types in C

Integers are stored unboxed with lowest bit set to 1

```
31 integer bits 1
```

Complex types like `(int x int)` are represented in boxed form:

```
Pointer

Tag: 0 int int
```
### Sum Types

```
type jargon =
  Foo of int×int
  | Bar
  | Baz
  | Qux of int
```

### Accessing Values

- `Int_val()` and `Val_int()` are used to shift in/out the lowest bit for integers
  - Easy to confuse
  - Can be applied to pointers without warning
- `Tag_val()` and `Field(v,i)` extract tag and data from complex types
  - Can be applied to integers without warning
- `Is_long()` used to distinguish pointers and integers

### Pattern Matching

```
value speak(value j) {
  if(is_long(j)) {
    if(Int_val(j) == 0) /* Bar */
    if(Int_val(j) == 1) /* Baz */
  } else {
    if(Tag_val(j) == 0) /* Foo */
    if(Tag_val(j) == 1) /* Qux */
  }
}
```

### Representation Overlap

```
type jargon =
  Foo of int×int
  | Bar
  | Baz
  | Qux of int
```

Need to track the results of conditionals
Representation Overlap

Type jargon = Foo of int\times int
| Bar
| Baz
| Qux of int

Same as \((\text{int}\times\text{int})\)

Representation Overlap

Type jargon = Foo of int\times int
| Bar
| Baz
| Qux of int

Same as \(0:\text{int}\)

Representational Types

Introduce representational type \((\Psi, \Sigma)\) to model arbitrary OCaml data as viewed by C:
- \(\Psi\) - represents the unboxed elements
- \(\Sigma\) - represents the size and structure of the boxed elements

Some Examples:
- \(\text{int}\): \((\infty, \emptyset)\)
- \(\text{int}\times\text{int}\): \((0, (\infty, \emptyset) \times (\infty, \emptyset))\)
- Foo of int \times int | Bar | Baz | Qux of int:
  \((2, (\infty, \emptyset) \times (\infty, \emptyset) + (\infty, \emptyset))\)

Type Inference

Augment value with flow-insensitive representational type and flow-sensitive tags:
- B,I,T are flow sensitive and vary from statement to statement
- May be set to Top if unknown
Inferring Sum Types

value helper(value j) {
    if(Is_long(j)) {
        if(Int_val(j) == 0)
            /* ... */
        if(Int_val(j) == 1)
            /* ... */
    } else {
        if(Tag_val(j) == 0)
            /* ... */
        if(Tag_val(j) == 1)
            /* ... */
    }
}

value helper(value j) {
    if(Is_long(j)) {
        if(Int_val(j) == 0)
            /* ... */
        if(Int_val(j) == 1)
            /* ... */
    } else {
        if(Tag_val(j) == 0)
            /* ... */
        if(Tag_val(j) == 1)
            /* ... */
    }
}
Inferring Sum Types

```plaintext
code
value helper(value j) {
    j: (ψ,0) value(T,0,T)
    if(Is_long(j)) {
        1 ≤ ψ  → (Int_val(j) == 0)
        2 ≤ ψ  → (Int_val(j) == 1)
        j: ... {boxed,0,0}
        if(Tag_val(j) == 0)  /* ... */
        if(Tag_val(j) == 1)  /* ... */
    } else {
        j: ... {boxed,0,1}
        if(Tag_val(j) == 0)  /* ... */
        if(Tag_val(j) == 1)  /* ... */
    }
}
```

Inferring Sum Types

```plaintext
code
value helper(value j) {
    j: (ψ,0) value(T,0,T)
    if(Is_long(j)) {
        1 ≤ ψ  → (Int_val(j) == 0)
        2 ≤ ψ  → (Int_val(j) == 1)
        j: ... {boxed,0,0}
        if(Tag_val(j) == 0)  /* ... */
        if(Tag_val(j) == 1)  /* ... */
    } else {
        j: ... {boxed,0,1}
        if(Tag_val(j) == 0)  /* ... */
        if(Tag_val(j) == 1)  /* ... */
    }
}
```

Inferring Sum Types

```plaintext
code
type jargon = Foo of int×int | Bar | Baz | Qux of int
```

Inferring Sum Types

```plaintext
code
type jargon = Foo of int×int | Bar | Baz | Qux of int
```

Inferring Sum Types

```plaintext
code
value helper(value j) {
    j: (ψ,0) value(T,0,T)
    if(Is_long(j)) {
        1 ≤ ψ  → (Int_val(j) == 0)
        2 ≤ ψ  → (Int_val(j) == 1)
        j: ... {boxed,0,0}
        if(Tag_val(j) == 0)  /* ... */
        if(Tag_val(j) == 1)  /* ... */
    } else {
        j: ... {boxed,0,1}
        if(Tag_val(j) == 0)  /* ... */
        if(Tag_val(j) == 1)  /* ... */
    }
}
```
Type Judgements for Expressions

• Expressions: $\Gamma \vdash e : \tau\{B,I,T\}$
  - In type environment $\Gamma$, $e$ has type $\tau$ with tags B, I, and T
  - Type rules for operations check tags

\[
\begin{align*}
\text{(Val Deref Exp)} & \\
\Gamma, P \vdash e : \text{mt value}\{\text{boxed}, n, m\} & \\
mt = (\Psi, \Pi_0 + \cdots + \Pi_m + \cdots + 0) & \\
\Pi_m = \text{mt}_0 \times \cdots \times \text{mt}_n \times \cdots \times 0 & \\
\Gamma, P \vdash *e : \text{mt}_n \text{value}\{\top, 0, \top\} &
\end{align*}
\]

Flow-Sensitive Statements

• Statements: $\Gamma, G \vdash s ; \Gamma'$
  - Contain an in-environment and an out-environment
  - In type environment $\Gamma$, $s$ is well typed and evaluating $s$ produces the new environment $\Gamma'$
  - $G$ maps source labels to environments, for branches
    - Different branches may use different environments

\[
\begin{align*}
\text{(If unboxed Stmt)} & \\
\Gamma, P \vdash x : \text{mt value}\{B, 0, T\} & \\
\Gamma[x \mapsto \text{mt value}\{\text{unboxed}, 0, T\}] \sqsubseteq G(L) & \\
\Gamma, G, P \vdash \text{if unboxed}(x) \text{ then } L, \Gamma[x \mapsto \text{mt value}\{\text{boxed}, 0, T\}] &
\end{align*}
\]

Safe?

```haskell
value cons(value hd, value tl) {
  value v = alloc_tuple(2);
  Field(v,0) = hd;
  Field(v,1) = tl;
  return(v);
end
```

Safe?

```haskell
value cons(value hd, value tl) {
  value v = alloc_tuple(2);
  Field(v,0) = hd;
  Field(v,1) = tl;
  return(v);
end
May invoke GC
```
Safe?

value cons(value hd, value tl) {
    value v = alloc_tuple(2);
    Field(v,0) = hd;
    Field(v,1) = tl;
    return(v);
end

Invalid pointer

Garbage Collection

• Allocating OCaml values can cause a GC cycle
• C functions must inform GC of all ML pointers
  ▪ Dead values are collected, live values may be moved!
• Must track which functions cause OCaml to allocate
• Augment function types with effects and solve via call-graph reachability

My 631 Project!

Garbage Collection

Results

<table>
<thead>
<tr>
<th>Program</th>
<th>C</th>
<th>ML LOC</th>
<th>Time(s)</th>
<th>Error</th>
<th>Warning</th>
<th>False-</th>
<th>Impreciso</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>apm</td>
<td>139</td>
<td>156</td>
<td>1.3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>camlzip</td>
<td>124</td>
<td>820</td>
<td>1.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ocaml-mad</td>
<td>139</td>
<td>38</td>
<td>4.2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ocaml-ssl</td>
<td>187</td>
<td>151</td>
<td>1.5</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>ocaml-glpk</td>
<td>305</td>
<td>147</td>
<td>1.3</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>gz</td>
<td>572</td>
<td>192</td>
<td>2.2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ocaml-vorbis</td>
<td>1183</td>
<td>443</td>
<td>2.8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>fplib</td>
<td>1401</td>
<td>21</td>
<td>1.7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>labgl</td>
<td>1586</td>
<td>1357</td>
<td>7.5</td>
<td>4</td>
<td>5</td>
<td>140</td>
<td>20</td>
<td>169</td>
</tr>
<tr>
<td>cryptokit</td>
<td>2173</td>
<td>2315</td>
<td>5.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>labgtk</td>
<td>5998</td>
<td>14847</td>
<td>61.3</td>
<td>9</td>
<td>11</td>
<td>74</td>
<td>48</td>
<td>142</td>
</tr>
<tr>
<td>Total</td>
<td>24</td>
<td>22</td>
<td>214</td>
<td>75</td>
<td>335</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2 GHz P4 Xeon, 2GB Memory
Errors

24 total errors:

- 5 errors due to GC violations
- 13 uses of Val_int instead of Int_val (or v.v.)
- Treating optional argument as actual argument:
  
  OCaml: `external f: ?x:int → unit = "f"
  C: value f(value x) {
  int bar = Int_val(x);

- Other similar typing errors

Warnings

22 total warnings

- Omitting a final parameter of type unit:
  
  OCaml: `external f: int → unit → unit = "f"
  C: value f(value x);

- Questionable use of 'a types:
  
  OCaml: `type input, output;
  external seek: int → 'a → unit = "seek"
  C: value seek(value pos, value chan) {
  int strm = Field(chan,0);
  fseek(strm,...); // strm is either
  // input or output

False Positives and Imprecision

214 False Positives (all in 2 benchmarks):

- Polymorphic variants (unsupported)
- Pointer operations disguised as integer ops:
  
  (((t*)v+1) == (t*)(v+sizeof(t))

75 Imprecision messages:

- Tags and offsets sometimes Top
- Globals and function pointers

JNI Usage

- Java defines a native method
  
  Class Foo {
  int x;
  private native void bar(Foo);
  }

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

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

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

- C implements
  ```c
  obj.class
  ```

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

- Same type

- C implements
  ```c
  obj.x
  ```
JNI Usage

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

- C implements

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

  Not obj!

Wrapper Functions

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

- Function accepts any object `obj` which has an integer field named by `field`
- Function should be polymorphic in the type of `obj` and the contents of `field`
  ```c
  my_getIntField(obj1,"x");
  my_getIntField(obj2,"offset");
  ```
  Statically check `x ∈ obj1, offset ∈ obj2`

Multilingual Types for the JNI

- Like with OCaml, embed Java types in C types
  - `jobect` instead of value
- Extend C strings types to include their value
  - As string variables resolve to string constants replace them with the Java types they represent
- Use instantiation constraints to support polymorphism
- No low level tag tests
  - JNI values are treated as black boxes
  - Large API for manipulating `jobect` types

Types must match!
Object Types

c_t ::= ... | j_t object  
j_t ::= α | {s;F;M} | j_t Class | ...

• Objects representations include the class name (s) and the set of fields (F) and methods (M)
• Instances of Class given type j_t Class instead of {"java/lang/Class";...}

GetObjectClass : {v;φ;µ} object → {v;φ;µ} Class object

Types as Strings

Sometimes only know a type by name, not structure:

\[ \text{FindClass} : \text{str}\{v\} \rightarrow \text{JTStr}\{v\} \text{ Class object} \]

• JTStr\{s\} represents the Java type named s
• When s is resolved to a constant, replace JTStr\{“Str”\} with actual type:
  
  JTStr\{“I”\} => Int  
  JTStr\{“java/lang/String”\} => {java.lang.String;...}

Precise Handling of String Values

c_t ::= ... | \text{str}\{s\}  
s ::= “Str” | v

• Extend C strings to include their value:
  
  “java/lang/String” : \text{str}\{“java/lang/String”\}

• Also include variables for inference:
  
  char *y : \text{str}\{v\}

JNI API
Instantiation Constraints

- Each call site is numbered uniquely
- Since $x$ and $y$ are passed at different call sites, $\tau$ need not equal $\tau'$

```plaintext
g() {
    $\tau x$;
    $\tau' y$;
    ...
    $f(x)$; $\alpha \leq_1 \tau$
    $f(y)$; $\alpha \leq_2 \tau'$
}
```

Instantiation Constraints

- If $\alpha \leq_1 \tau$ then there must exist a substitution $S_i$ such that $S_i(\alpha) = \tau$
- Thus, any structure in $\alpha$ must be copied to $\tau'$:

$$
(\beta \times \gamma) \leq_1 \tau
$$
**Instantiation Constraints**

- If $\alpha \leq_1 \tau$ then there must exist a substitution $S_1$ such that $S_1(\alpha) = \tau$

- Thus, any structure in $\alpha$ must be copied to $\tau$:

$$T = T_1 \times T_2$$

$$\beta \times \gamma \leq_1 T \Rightarrow \beta \leq_1 T_1$$

$$\gamma \leq_1 T_2$$

$$\text{int} \leq_1 T \Rightarrow T = \text{int}$$

---

**Polymorphism**

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);
}
```

How do we infer the type of `my_getIntField`?
Example

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

Example

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

Example

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

Example

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

Example

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

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

GetObjectClass : \{v_2;\phi_2;\mu_2\} object \rightarrow \{v_2;\phi_2;\mu_2\} Class object

\{v_2;\phi_2;\mu_2\} s_1 \alpha_1 \quad \alpha_3 = \{v_3;\phi_3;\mu_3\} \quad v_2 \preceq v_3 \quad v_2 \preceq v_4

\{v_2;\phi_2;\mu_2\} Class \s_1 \alpha_2 \quad \alpha_2 = \{v_4;\phi_4;\mu_4\} Class \quad \mu_2 \preceq v_3 \quad \mu_2 \preceq v_4

obj: \alpha_1 \text{ object } 
field: \text{str}(v_1) 
cls: \alpha_2 \text{ object } 
```

Example

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

GetObjectClass : \{v_2;\phi_2;\mu_2\} object \rightarrow \{v_2;\phi_2;\mu_2\} Class object

\{v_2;\phi_2;\mu_2\} s_1 \alpha_1 \quad \alpha_3 = \{v_3;\phi_3;\mu_3\} \quad v_2 \preceq v_3 \quad v_2 \preceq v_4

\{v_2;\phi_2;\mu_2\} Class \s_1 \alpha_2 \quad \alpha_2 = \{v_4;\phi_4;\mu_4\} Class \quad \mu_2 \preceq v_3 \quad \mu_2 \preceq v_4

obj: \{v_3;\phi_3;\mu_3\} object 
field: \text{str}(v_1) 
cls: \{v_3;\phi_3;\mu_3\} \text{ Class object } 
```

Example

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

```
V_5:JTStr(V_6);... \preceq_1 \phi_3 \quad \phi_3 = V_1:JTStr("I");...
V_5 \preceq_1 V_1
```

obj: \{v_3;\phi_3;\mu_3\} object 
field: \text{str}(v_1) 
cls: \{v_3;\phi_3;\mu_3\} \text{ Class object } 

Example

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

```
V_5:JTStr(V_6);... \preceq_1 \phi_3 \quad \phi_3 = V_1:JTStr("I");...
V_5 \preceq_1 V_1
```

obj: \{v_3;V_1:Int;...;\mu_3\} object 
field: \text{str}(v_1) 
cls: \{v_3;V_1:Int;...;\mu_3\} \text{ Class object } 
```
Example

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

my_getIntField: \{v_3;v_1:Int;...;v_4\} object \times str(v_1) \rightarrow int

accepts any object named \(v_3\) which has an integer field named by \(v_1\)

Errors

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

Results

<table>
<thead>
<tr>
<th>Program</th>
<th>C LOC</th>
<th>Java</th>
<th>Time(s)</th>
<th>Errors</th>
<th>Warning</th>
<th>False Pos</th>
<th>Imp</th>
</tr>
</thead>
<tbody>
<tr>
<td>libgconf-java-2.10.1</td>
<td>1119</td>
<td>670</td>
<td>2.4</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>6.9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>libgnome-java-2.10.1</td>
<td>5606</td>
<td>5135</td>
<td>17.4</td>
<td>45</td>
<td>8</td>
<td>34</td>
<td>18</td>
</tr>
<tr>
<td>libgtk-java-2.6.2</td>
<td>27095</td>
<td>32395</td>
<td>36.3</td>
<td>74</td>
<td>45</td>
<td>67</td>
<td>18</td>
</tr>
<tr>
<td>libgtkxml-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>libtkhtml-java-2.6.2</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>libtkxml-1.7.0</td>
<td>437</td>
<td>184</td>
<td>2.5</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>17.3</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libreadline-0.8.0</td>
<td>1459</td>
<td>324</td>
<td>2.2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pgpjava</td>
<td>10136</td>
<td>123</td>
<td>2.7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>posix1.0</td>
<td>978</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></td>
<td></td>
<td>155</td>
<td>36</td>
<td>44</td>
<td>22</td>
<td></td>
</tr>
</tbody>
</table>

Warnings

- 1 malformed Java class string
- 2 incorrect type declarations
  - JNI contains several typedef’s for object (e.g., jstring, jintarray)
  - 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
False Positives 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 partially specified method
  - 6 passing arguments to JNI functions packed in an array

Conclusion

- Developed multi-lingual type inference systems to check FFIs
- OCaml:
  - Representational types model values in memory
  - Uses dataflow analysis to guide typing rules
- JNI
  - Precisely track C strings by embedding them in types
  - Treat functions polymorphically, even in the contents of string variables
- Found many errors in practice