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:

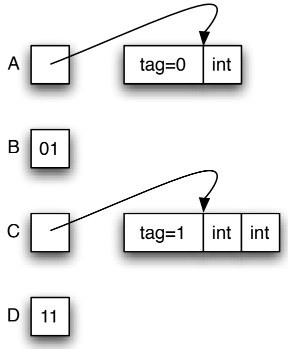
```
typedef long value;
value c_foo(value int_arg, value int_list_arg);
```

  - 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
```

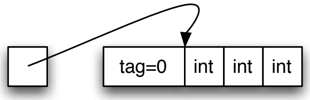


Primitive Types

- 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 | ...



tag=0 | int | int | int

Example: "Pattern Matching"

```
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)
}
```

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

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

```
value bar(value list) {
  CAMLparam1(list);
  CAMLlocal1(temp);
  temp = alloc_tuple(2);
  CAMLreturn(Val_unit);
}
```

```
value foo(value arg) {
  bar(arg);
  return(arg);
}
```

- What's wrong with foo?
  - Doesn't register its parameter

Representational Types

- Types to model C's view of OCaml data

# of nullary constructors $\longrightarrow$ $mt ::= (C, S)$ $\longleftarrow$ arg types of other constructors

$S ::= \sigma \mid P + S \mid \varepsilon$

$P ::= \pi \mid mt \times P \mid \varepsilon$

Examples:

int ⇒ (∞, ε)

int * int ⇒ $(0, (\infty,0)\times(\infty,0) + \varepsilon)$

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

⇒ $(2, (\infty,0) + (\infty,0)\times(\infty,0) + \varepsilon)$

Tracking OCaml Types through C

- Extend the C type value to

(C, S) value $\{B, T\}$

Representational Type $\longrightarrow$ (C, S)

boxed or unboxed $\longrightarrow$ B

value (if int) block tag (if ptr) $\longrightarrow$ T

(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

x: (ψ, σ) value{Top, Top}

if (Is_long(x)) { $\longleftarrow$ x: ...{unboxed, Top}

$\psi \geq 1 \longrightarrow$ if (Int_val(x) == 0) /* B */ $\longleftarrow$ x: ...{unboxed, 0}

...

$\psi \geq 2 \longrightarrow$ if (Int_val(x) == 1) /* D */ $\longleftarrow$ x: ...{unboxed, 1}

...

} else { $\longleftarrow$ x: ...{boxed, Top}

$\sigma = \pi + \sigma' \longrightarrow$ if (Tag_val(x) == 0) /* A */

$\pi = \mathrm{int} \times \pi' \longrightarrow$ Field(x, 0) = Val_int(0) $\longleftarrow$ x: ...{boxed, 0}

$\sigma' = \pi'' + \sigma'' \longrightarrow$ if (Tag_val(x) == 1) /* C */

$\pi'' = \alpha \times \mathrm{int} \times \pi''' \longrightarrow$ Field(x, 1) = Val_int(0) $\longleftarrow$ x: ...{boxed, 1}

}

Inferring Sum Types

$\psi \geq 1$

$\psi \geq 2$

$\sigma = \pi + \sigma'$

$\pi = \mathrm{int} \times \pi'$

$\sigma' = \pi'' + \sigma''$

$\pi'' = \alpha \times \mathrm{int} \times \pi'''$

Solution to constraints:

x: (ψ, σ) value

$\psi \geq 2$

$\sigma = \mathrm{int} \times \pi' + \alpha \times \mathrm{int} \times \pi'' + \sigma''$

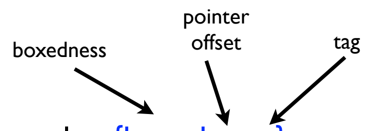
Compatible the OCaml type

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

⇒ $(2, (\infty,0) + (\infty,0)\times(\infty,0) + \varepsilon)$

Example Type Rules

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



$$\dfrac{\begin{array}{c} A \vdash e :\ mt\ \text{value}\ \{\text{boxed}, n, m\} \\ mt = (C,\ P_0 + \cdots + P_m + S) \\ P_m = mt_0 \times \cdots \times mt_n \times P \end{array}}{A \vdash {}^*e : mt_n\ \text{value}\{\text{Top}, 0, \text{Top}\}}$$

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

$$\dfrac{\begin{array}{c} A \vdash x :\ mt\ \text{value}\ \{B, 0, T\} \\ A' = A[x \rightarrow mt\ \text{value}\{\text{unboxed}, 0, T\} \\ A' \leq G(L) \end{array}}{A \vdash \text{if unboxed}(x)\ \text{then}\ L,\ A[x \rightarrow mt\ \text{value}\ \{\text{boxed}, 0, T\}}$$

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 $P(f_i) = 0$

$$\text{foo}() \longrightarrow \text{bar}() \longrightarrow \text{alloc\_tuple}()$$

P(foo) = { arg }   error: non-empty

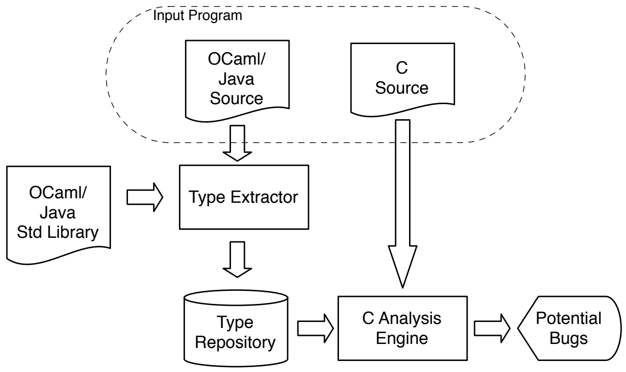
Soundness

- 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

Implementation (Both)



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

runtime exns or hard crashes

non-fatal but suspicious

correct code

insufficient info

Program	C-loc	O-loc	Ext	Time	Err	Wrn	FPos	Imp
apm-1.00	124	156	4	0.01s	0	0	0	0
camlzip-1.01	139	820	9	0.01s	0	0	0	1
ocaml-mad-0.1.0	139	38	3	0.01s	1	0	0	0
ocaml-ssl-0.1.0	187	151	14	0.02s	4	2	0	0
ocaml-glpk-0.1.1	305	147	30	0.03s	4	1	0	1
gz-0.5.5	572	192	29	0.02s	0	1	0	1
ocaml-vorbis-0.1.1	1183	443	7	0.07s	1	0	0	2
ftplib-0.12	1401	21	17	0.06s	1	2	0	1
lablgl-1.00	1586	1357	324	0.40s	4	5	140	20
cryptokit-1.2	2173	2315	24	0.03s	0	0	0	1
lablgtk-2.2.0	5998	14847	1307	3.83s	9	11	74	48
				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 + I == (t*) (v + sizeof(t*))
    - 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:

```
Class Foo {
 int x;
 private native void bar(Foo);
}
```

- 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:

```
Class Foo {
 int x;
 private native void bar(Foo);
}
```

- C:

obj.class

```
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:

```
Class Foo {
 int x;
 private native void bar(Foo);
}
```

- C:

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

I = Int

obj.x

Example JNI Code

- Java:

```
Class Foo {
 int x;
 private native void bar(Foo);
}
```

- C:

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

y = obj.x;

Example JNI Code

- Java:

```
Class Foo {
 int x;
 private native void bar(Foo);
}
```

- C:

Same type

```
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:

```
Class Foo {
 int x;
 private native void bar(Foo);
}
```

- C:

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

Not obj!

Example JNI Code

- Java:

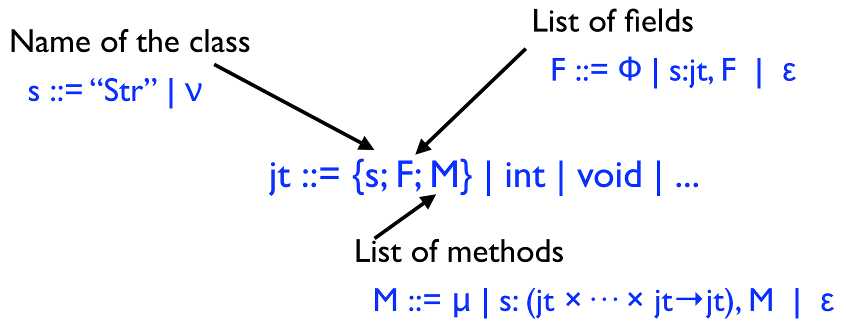
```
Class Foo {
 int x;
 private native void bar(Foo);
}
```

- 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" | ν

List of fields

F ::= Φ | s:jt, F | ε

jt ::= {s; F; M} | int | void | ...

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{ν}
    - 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 : {ν;ϕ;μ} jobject → {ν;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{ν} → String(ν) Class jobject

Wrapper Functions

```
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 `field`
  - Polymorphic in *type* of obj and *contents* of field

    ```
    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

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

$$\forall\ \nu_1,\nu_3,\mu_3\ .\ \{\nu_3;\ \nu_1\text{:int}, \ldots;\mu_3\}\ \text{jobject} \times \text{str}\{\nu_1\} \rightarrow \text{int}$$

- Second arg is some string ν_1
- First arg is some object with an int field of name ν_1
- The function returns an int

Polymorphism via Semiunification

- Generate *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

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

runtime exns or hard crashes

non-fatal but suspicious

correct code

insufficient info

Program	C-loc	J-loc	Ext	Time	Err	Wrn	FPos	Imp
libgconf-java-2.10.1	1119	670	93	1.32s	0	0	10	0
libglade-java-2.10.1	149	1022	6	0.64s	0	0	0	1
libgnome-java-2.10.1	5606	5135	599	6.53s	45	0	0	1
libgtk-java-2.6.2	27095	32395	3201	1.04s	74	8	36	18
libgtkhtml-java-2.6.0	455	729	72	0.65s	27	0	0	0
libgtkmozembed-java-1.7.0	166	498	23	0.66s	0	0	0	0
libvte-java-0.11.11	437	184	36	0.67s	0	26	0	0
jnetfilter	1113	1599	105	5.38s	9	0	0	0
libreadline-java-0.8.0	1459	324	17	0.63s	0	0	0	1
pgpjava	10136	123	12	1.11s	0	1	0	1
posix1.0	978	293	26	0.70s	0	1	0	0
Java Mustang compiler	532k	1974k	2495	630s	1	88	96	2620
				Total	156	124	142	2642

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, jintarray)
  - 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