

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

2

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

3

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

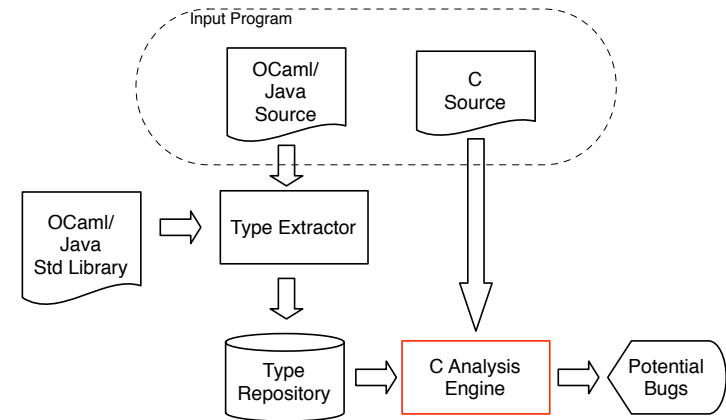
4

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

5

Architecture



6

OCaml FFI

OCaml:

```
external ml_fun: int → int list → unit = "c_fun"
```

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

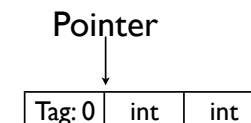
7

OCaml Types in C

Integers are stored unboxed with lowest bit set to 1

```
31 integer bits | 1
```

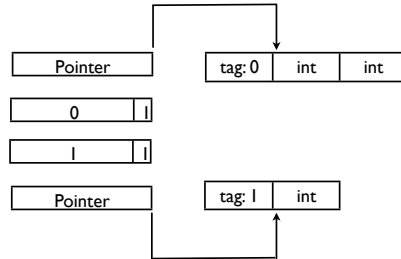
Complex types like `(int×int)` are represented in boxed form:



8

Sum Types

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



9

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

10

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 */  
  }  
}
```

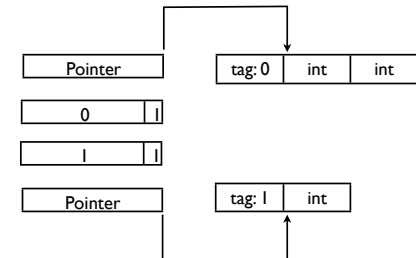
```
type jargon =  
  Foo of int×int  
| Bar  
| Baz  
| Qux of int
```

Need to track the results of conditionals

11

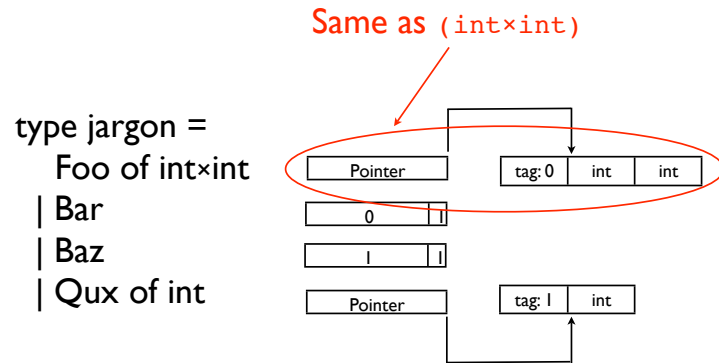
Representation Overlap

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



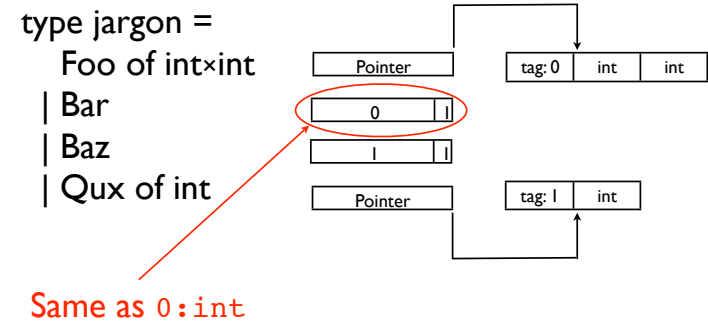
12

Representation Overlap



13

Representation Overlap



14

Representational Types

Introduce *representational* type (Ψ, Σ) to model arbitrary OCaml data as viewed by C

Ψ - represents the unboxed elements

Σ - represents the size and structure of the boxed elements

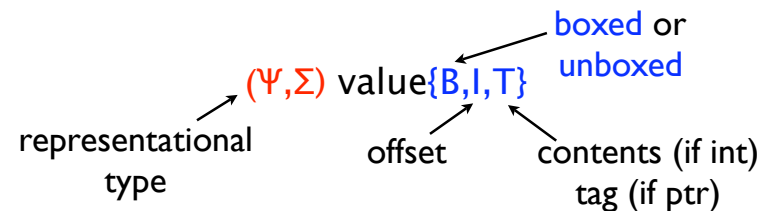
Some Examples:

- int: (∞, \emptyset)
- int×int: $(0, (\infty, \emptyset) \times (\infty, \emptyset))$
- Foo of int × int | Bar | Baz | Qux of int:
 $(2, (\infty, \emptyset) \times (\infty, \emptyset) + (\infty, \emptyset))$

15

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

16

Inferring Sum Types

```
value helper(value j) { j: ( $\psi, \sigma$ ) value{T,0,T}
  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)
      /* ... */
  }
}
```

17

Inferring Sum Types

```
value helper(value j) { j: ( $\psi, \sigma$ ) value{T,0,T}
  if(Is_long(j)) {  $\leftarrow$  j: ...{unboxed,0,T}
    if(Int_val(j) == 0)
      /* ... */
    if(Int_val(j) == 1)
      /* ... */
  } else {
    if(Tag_val(j) == 0)
      /* ... */
    if(Tag_val(j) == 1)
      /* ... */
  }
}
```

18

Inferring Sum Types

```
value helper(value j) { j: ( $\psi, \sigma$ ) value{T,0,T}
  if(Is_long(j)) {  $\leftarrow$  j: ...{unboxed,0,T}
     $1 \leq \psi \rightarrow$  if(Int_val(j) == 0)
      /* ... */
    if(Int_val(j) == 1)
      /* ... */
  } else {
    if(Tag_val(j) == 0)
      /* ... */
    if(Tag_val(j) == 1)
      /* ... */
  }
}
```

19

Inferring Sum Types

```
value helper(value j) { j: ( $\psi, \sigma$ ) value{T,0,T}
  if(Is_long(j)) {  $\leftarrow$  j: ...{unboxed,0,T}
     $1 \leq \psi \rightarrow$  if(Int_val(j) == 0)
      /* ... */  $\leftarrow$  j: ...{unboxed,0,0}
    if(Int_val(j) == 1)
      /* ... */
  } else {
    if(Tag_val(j) == 0)
      /* ... */
    if(Tag_val(j) == 1)
      /* ... */
  }
}
```

20

Inferring Sum Types

```

value helper(value j) { j: (ψ,σ) value{T,0,T}
  if(Is_long(j)) {
    1 ≤ ψ → if(Int_val(j) == 0) j: ...{unboxed,0,T}
            /* ... */ ← j: ... {unboxed,0,0}
    2 ≤ ψ → if(Int_val(j) == 1) j: ... {unboxed,0,1}
            /* ... */ ← j: ... {unboxed,0,1}
  } else {
    if(Tag_val(j) == 0)
      /* ... */
    if(Tag_val(j) == 1)
      /* ... */
  }
}

```

21

Inferring Sum Types

```

value helper(value j) { j: (ψ,σ) value{T,0,T}
  if(Is_long(j)) {
    if(Int_val(j) == 0)
      /* ... */
    if(Int_val(j) == 1)
      /* ... */
  } else {
    σ = π0 + ? → if(Tag_val(j) == 0) j: (ψ,σ) value{boxed,0,T}
                  /* ... */ ← j: ... {boxed,0,0}
    if(Tag_val(j) == 1)
      /* ... */
  }
}

```

22

Inferring Sum Types

```

value helper(value j) { j: (ψ,σ) value{T,0,T}
  if(Is_long(j)) {
    if(Int_val(j) == 0)
      /* ... */
    if(Int_val(j) == 1)
      /* ... */
  } else {
    σ = π0 + ? → if(Tag_val(j) == 0) j: (ψ,σ) value{boxed,0,T}
                  /* ... */ ← j: ... {boxed,0,0}
    σ = π0 + π1 + ? → if(Tag_val(j) == 1) j: ... {boxed,0,1}
                       /* ... */ ← j: ... {boxed,0,1}
  }
}

```

23

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)
      /* ... */
  }
  2 ≤ ψ
  σ = π0 + π1 + ? } j: (ψ,σ) value{T,0,T}
}

```

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

24

Type Judgements for Expressions

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

$$\begin{array}{c}
 \text{(VAL Deref EXP)} \\
 \Gamma, P \vdash e : mt \text{ value}\{\text{boxed}, n, m\} \\
 mt = (\Psi, \Pi_0 + \dots + \Pi_m + \dots + \emptyset) \\
 \Pi_m = mt_0 \times \dots \times mt_n \times \dots \times \emptyset \\
 \hline
 \Gamma, P \vdash *e : mt_n \text{ value}\{\top, 0, \top\}
 \end{array}$$

25

Flow-Sensitive Statements

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

$$\begin{array}{c}
 \text{(IF UNBOXED STMT)} \\
 \Gamma, P \vdash x : mt \text{ value}\{B, 0, T\} \\
 \Gamma[x \mapsto mt \text{ value}\{\text{unboxed}, 0, T\}] \sqsubseteq G(L) \\
 \hline
 \Gamma, G, P \vdash \text{if_unboxed}(x) \text{ then } L, \Gamma[x \mapsto mt \text{ value}\{\text{boxed}, 0, T\}]
 \end{array}$$

26

Safe?

```

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

27

Safe?

```

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

28

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

29

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

30

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!

31

Results

Program	C	ML LOC	Time(s)	Error	Warning	False-	Imprecisio	Tota
apm	139	156	1.3	0	0	0	0	0
camlzip	124	820	1.7	0	0	0	1	1
ocaml-mad	139	38	4.2	1	0	0	0	1
ocaml-ssl	187	151	1.5	4	2	0	0	6
ocaml-glpk	305	147	1.3	4	1	0	1	6
gz	572	192	2.2	0	1	0	1	2
ocaml-	1183	443	2.8	1	0	0	2	3
ftplib	1401	21	1.7	1	2	0	1	4
lablgl	1586	1357	7.5	4	5	140	20	169
cryptokit	2173	2315	5.4	0	0	0	1	1
lablgtk	5998	14847	61.3	9	11	74	48	142
Total				24	22	214	75	335

2 GHz P4 Xeon, 2GB Memory

32

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

33

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
```

34

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

35

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);
    ...
}
```

36

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);  
    ...  
}
```

obj.class

37

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);  
    ...  
}
```

obj.x

I = Int

38

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);  
    ...  
}
```

} y = obj.x;

39

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);  
    ...  
}
```

Same type

40

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);
    ...
}
```

Not obj!

41

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);
    ...
}
```

Types must match!

42

Wrapper Functions

```
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`

```
my_getIntField(obj1, "x");
my_getIntField(obj2, "offset");
```

Statically check `x ∈ obj1`, `offset ∈ obj2`

43

Multilingual Types for the JNI

- Like with OCaml, embed Java types in C types
 - `jobject` 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 `jobject` types

44

Instantiation Constraints

$f : \alpha \rightarrow \text{unit}$

```

g() {
   $\tau$  x;
   $\tau'$  y;
  ...
  f(x);
  f(y);
}
    
```

$\alpha \leq_1 \tau$
 $\alpha \leq_2 \tau'$

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

49

Instantiation Constraints

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

$$(\beta \times \gamma) \leq_i \tau$$

50

Instantiation Constraints

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

$$(\beta \times \gamma) \leq_i \tau \quad \Rightarrow \quad \tau = \tau_1 \times \tau_2$$

51

Instantiation Constraints

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

$$(\beta \times \gamma) \leq_i \tau \quad \Rightarrow \quad \begin{aligned} \tau &= \tau_1 \times \tau_2 \\ \beta &\leq_i \tau_1 \\ \gamma &\leq_i \tau_2 \end{aligned}$$

52

Instantiation Constraints

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

$$\begin{aligned} (\beta \times \gamma) \leq_i \tau &\Rightarrow \begin{aligned} \tau &= \tau_1 \times \tau_2 \\ \beta &\leq_i \tau_1 \\ \gamma &\leq_i \tau_2 \end{aligned} \\ \text{int} \leq_i \tau &\Rightarrow \tau = \text{int} \end{aligned}$$

53

Instantiation Constraints

- The substitutions must also be consistent within a call site:

$f : \alpha \times \alpha \rightarrow \text{unit}$
...

$$\begin{array}{l} \tau \ x \\ \tau' \ y \\ f(x, y); \end{array} \left\{ \begin{array}{ll} \alpha \leq_i \tau & S_i(\alpha) = \tau \\ \alpha \leq_i \tau' & S_i(\alpha) = \tau' \end{array} \right.$$

Therefore, $\tau = \tau'$

54

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

55

Example

```
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`?

56

Example

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

GetObjectClass : $\{V_2;\Phi_2;\mu_2\}$ jobject \rightarrow $\{V_2;\Phi_2;\mu_2\}$ Class jobject

$$\begin{array}{l} \{V_2;\Phi_2;\mu_2\} \leq_1 \alpha_1 \quad \alpha_1 = \{V_3;\Phi_3;\mu_3\} \quad V_2 \leq_1 V_3 \quad V_2 \leq_1 V_4 \\ \{V_2;\Phi_2;\mu_2\} \text{ Class} \leq_1 \alpha_2 \quad \alpha_2 = \{V_4;\Phi_4;\mu_4\} \text{ Class} \quad \Phi_2 \leq_1 \Phi_3 \quad \Phi_2 \leq_1 \Phi_4 \\ \mu_2 \leq_1 \mu_3 \quad \mu_2 \leq_1 \mu_4 \end{array}$$

obj: α_1 jobject
field: str $\{V_1\}$
cls: α_2 jobject

61

Example

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

GetObjectClass : $\{V_2;\Phi_2;\mu_2\}$ jobject \rightarrow $\{V_2;\Phi_2;\mu_2\}$ Class jobject

$$\begin{array}{l} \{V_2;\Phi_2;\mu_2\} \leq_1 \alpha_1 \quad \alpha_1 = \{V_3;\Phi_3;\mu_3\} \quad V_2 \leq_1 V_3 \quad V_2 \leq_1 V_4 \\ \{V_2;\Phi_2;\mu_2\} \text{ Class} \leq_1 \alpha_2 \quad \alpha_2 = \{V_4;\Phi_4;\mu_4\} \text{ Class} \quad \Phi_2 \leq_1 \Phi_3 \quad \Phi_2 \leq_1 \Phi_4 \\ \mu_2 \leq_1 \mu_3 \quad \mu_2 \leq_1 \mu_4 \end{array}$$

obj: $\{V_3;\Phi_3;\mu_3\}$ jobject
field: str $\{V_1\}$
cls: $\{V_3;\Phi_3;\mu_3\}$ Class jobject

62

Example

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

$$\begin{array}{l} V_5:\text{JTStr}\{V_6\};\dots \leq_1 \Phi_3 \quad \Phi_3 = V_1:\text{JTStr}\{"I"\};\dots \\ V_5 \leq_1 V_1 \end{array}$$

obj: $\{V_3;\Phi_3;\mu_3\}$ jobject
field: str $\{V_1\}$
cls: $\{V_3;\Phi_3;\mu_3\}$ Class jobject

63

Example

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

$$\begin{array}{l} V_5:\text{JTStr}\{V_6\};\dots \leq_1 \Phi_3 \quad \Phi_3 = V_1:\text{JTStr}\{"I"\};\dots \\ V_5 \leq_1 V_1 \quad \Phi_3 = V_1:\text{Int};\dots \end{array}$$

obj: $\{V_3;V_1:\text{Int};\dots;\mu_3\}$ jobject
field: str $\{V_1\}$
cls: $\{V_3;V_1:\text{Int};\dots;\mu_3\}$ Class jobject

64

Example

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

my_getIntField: $\{V_3; V_1: \text{Int}; \dots; \mu_3\}$ jobject \times str $\{V_1\}$ \rightarrow int

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

65

Results

Program	C LOC	Java	Time(s)	Errors	Warning	False Pos	Imp
libgconf-java-2.10.1	1119	670	2.4	0	0	10	0
libglade-java-2.10.1	149	1022	6.9	0	0	0	1
libgnome-java-2.10.1	5606	5135	17.4	45	0	0	1
libgtk-java-2.6.2	27095	32395	36.3	74	8	34	18
libgtkhtml-java-2.6.0	455	729	2.9	27	0	0	0
libgtkmozembed-java-1.7.0	166	498	3.3	0	0	0	0
libvte-java-0.11.11	437	184	2.5	0	26	0	0
jnetfilter	1113	1599	17.3	9	0	0	0
libreadline-0.8.0	1459	324	2.2	0	0	0	1
pgpjava	10136	123	2.7	0	1	0	1
posix1.0	978	293	1.8	0	1	0	0
Total				155	36	44	22

66

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 \neq byte[]
- 14 functions were named incorrectly
 - Functions must follow a strict convention to be called from Java

67

Warnings

- 1 malformed Java class string
- 2 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
- 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

68

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

69

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

70