Checking Type Safety of Foreign Function Calls

Michael Furr and Jeffrey S. Foster

Introduction

- Many 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
  - Pre-existing library (e.g., system routines)
  - Suitability of language for particular problem
  - Performance of other language

Dangers of FFIs

- Unfortunately, FFIs are often easy to misuse
  - Little or no checking done at language boundary
- Goal: Enforce safety of multi-lingual programs
  - Are types respected by the interface?
    - Is an integer on one side and integer on the other?
  - Are resources used correctly?
    - Are GC invariants respected?

Today

Checking Safety of OCaml's FFI to C [PLDI 2005]

- OCaml: Strongly-typed, mostly-functional, GC
- C: Type-unsafe, imperative, explicit alloc/free
- FFI is lightweight and fairly typical
  - Most of the work done by C "glue" code
    - Macros and functions to manipulate OCaml data
- Ideas apply to other systems

Our Approach

- Static (compile-time) analysis tool
  - Finds FFI errors in multi-lingual OCaml/C 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 precisely

The OCaml FFI

- OCaml:

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

- C:

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

  - value can be either a primitive (int, unit) or a pointer to the ML heap (int list)
  - Linker checks for presence of symbol
    - No other checks

The value type

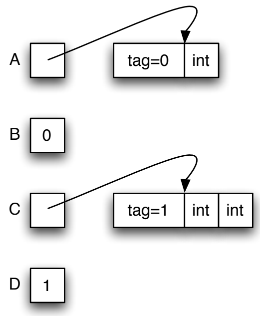
- value represents both primitives and pointers:

typedef long value;

  - "Conflating" foreign types together common design
    - E.g., most classes have type jobject in JNI
  - Manipulated using macros and functions
  - *No checking* that value is used correctly...

Physical Representations of Data

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



Accessing Primitives

- Unboxed data (e.g., int) has low bit set to 1
  - 0 : int = B = unit
  - Enables GC to distinguish pointers
- Val_int() and Int_val() perform shifting ops
  - Can you guess which is which?
  - Worse: Can apply either to a pointer
    - Since value is a typedef of long

Accessing Structured Blocks

- Field(x, i) – read ith field of x
  - Expands to *((value *) x + i)
- Tag_val() – read tag in header
  - Tag of a tuple or record not in sum is 0
    - Notice overlapping physical representation
- Both can be misused
  - Apply to a primitive, access outside of block
- Use Is_long() to distinguish unboxed/boxed data

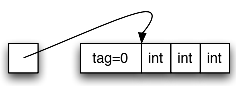
Example: "Pattern Matching"

```
if (Is_long(x)) {

  if (Int_val(x) == 0)
    /* B */
  if (Int_val(x) == 1)
    /* D */

} else {

  if (Tag_val(x) == 0)
    /* A */
  if (Tag_val(x) == 1)
    /* C */

}
```

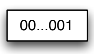
```
type t =
  A of int
| B
| C of int * int
| D
```


Overlap of Physical Representations

- Our goal: Track OCaml types through C code
  - But C code can see physical overlap of OCaml data



- Could be int * int * int
- Could be Foo of type t' = Foo of int * int * int | ...



- Could be 0 : int or unit or Bar of type t'' = Bar | ...

Representational Types

- *Representational* type (C, S) models such data
  - C = # of nullary constructors, 0 if none
  - S = arg types of other constructors, 0 if none
- Examples:
  - int ⇒ (∞, 0)
  - int * int ⇒ (0, (∞,0)*(∞,0))
  - type t = A of int | B | C of int * int | D
    ⇒ (2, (∞,0) + (∞,0)*(∞,0))

Original Type Systems

$$
\begin{array}{rcl}
mltype & ::= & \texttt{unit} \mid \texttt{int} \mid mltype \times mltype \\
 & \mid & S + \cdots + S \mid mltype\ \texttt{ref} \\
 & \mid & mltype \rightarrow mltype \\
S & ::= & Constr \mid Constr\ \texttt{of}\ mltype
\end{array}
$$

(a) OCaml Type Grammar

$$
\begin{array}{rcl}
ctype & :: & \texttt{void} \mid \texttt{int} \mid \texttt{value} \mid ctype\ * \\
 & \mid & ctype \times \ldots \times ctype \rightarrow ctype
\end{array}
$$

(b) C Type Grammar

Multi-Lingual Type System

$$
\begin{array}{rcl}
ct & ::= & \texttt{void} \mid \texttt{int} \mid mt\ \texttt{value} \mid ct\ * \\
 & \mid & ct \times \cdots \times ct \rightarrow ct
\end{array}
$$

$$
\begin{array}{rcl}
mt & ::= & \alpha \mid mt \rightarrow mt \mid ct\ \texttt{custom} \mid (\Psi, \Sigma) \\
\Psi & ::= & \psi \mid n \mid \top \\
\Sigma & ::= & \sigma \mid \emptyset \mid \Pi + \Sigma \\
\Pi & ::= & \pi \mid \emptyset \mid mt \times \Pi
\end{array}
$$

Type Inference

- Input: A program written in OCaml and C
- Step 1: Analyze OCaml source
  - Extract types of external functions
  - Convert into representational types
- Step 2: Analyze C source
  - Infer ML types for value arguments
  - Check for consistency with results from step 1

The Need for Flow-Sensitivity

- Recall our pattern matching code

```
if (Is_long(x)) {
  if (Int_val(x) == 0)
    ...
} else {
  if (Tag_val(x) == 0)
    ...Field(x, 0)...
}
```

- For inference, need to track
  - Results of conditional tests
  - Precise integer values
  - Offsets into structured blocks

Dataflow Analysis

- Extend the C type value to

boxed or unboxed

(C, S) value[B{I}] T

Representational Type

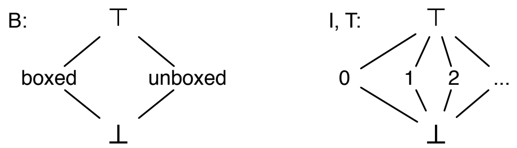
offset

value (if int)
block tag (if ptr)

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

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

Lattices for B, I, T



Inferring Integers

```
value succ(value v) {          v: (ψ, σ) value[Top{0}]{Top}
  int next = Int_val(v) + 1;        σ=0
  return Val_int(next);
}
```


Inferring Tuples

```
value fst(value v) {          v: (ψ, σ) value[Top{0}]{Top}
  value f = Field(v, 0);          ψ=0, σ=π+0,
                                   π = α * π′
  return f;
}
```


Inferring Sum Types

```
                                     v: (ψ, σ) value[Top{0}]{Top}
            if (Is_long(x)) {
                                  ← v: ...[unboxed{0}]{Top}
   ψ≥1        if (Int_val(x) == 0)
                /* B */           ← v: ...[unboxed{0}]{0}
   ψ≥2        if (Int_val(x) == 1)
                /* D */           ← v: ...[unboxed{0}]{1}
            } else {
                                  ← v: ...[boxed{0}]{Top}
   σ=π+σ′     if (Tag_val(x) == 0)
                /* A */           ← v: ...[boxed{0}]{0}
   σ′=π′+σ″   if (Tag_val(x) == 1)
                /* C */           ← v: ...[boxed{0}]{1}
            }
```


Type Rules for Expressions

- Rules construct and consume types and tags
  - These rules are *not* flow-sensitive, since expressions don't have side effects

INT EXP

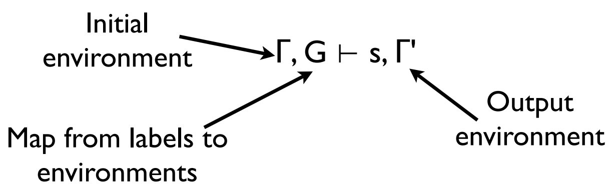
$$\frac{}{\Gamma, P \vdash n : \mathtt{int}\{\top, 0, n\}}$$

VAL DEREF EXP

$$\frac{\begin{array}{c}\Gamma, P \vdash e : mt\ \mathtt{value}\{\mathrm{boxed}, n, m\} \\ mt = (\psi, \pi_0 + \cdots + \pi_m + \sigma) \\ \pi_m = \alpha_0 \times \ldots \times \alpha_n \times \pi \qquad \psi, \pi_i, \sigma, \alpha_i, \pi \text{ fresh}\end{array}}{\Gamma, P \vdash *e : \alpha_n\ \mathtt{value}\{\top, 0, \top\}}$$

Types Rules for Statements

- In practice, only need flow-sensitive locals
  - Tracking the heap is much more complicated
- Idea: Make Γ both an input and an *output*
  - Also need to track Γ at join points



Types Rules for Statements (cont'd)

SEQ STMT

$$\frac{\Gamma, G, P \vdash s_1, \Gamma' \qquad \Gamma', G, P \vdash s_2, \Gamma''}{\Gamma, G, P \vdash s_1\ ;\ s_2, \Gamma''}$$

LBL STMT

$$\frac{G(L), G, P \vdash s, \Gamma' \qquad \Gamma \sqsubseteq G(L)}{\Gamma, G, P \vdash L\colon s, \Gamma'}$$

GOTO STMT

$$\frac{G := G[L \mapsto G(L) \sqcup \Gamma]}{\Gamma, G, P \vdash \texttt{goto}\ L, reset(\Gamma)}$$

Types Rules for Statements (cont'd)

VSET STMT

$$\frac{\Gamma, P \vdash e : ct\{B, I, T\}}{\Gamma, G, P \vdash x := e, \Gamma[x \mapsto ct\{B, I, T\}]}$$

IF UNBOXED STMT

$$\frac{\begin{array}{c}\Gamma, P \vdash x : mt\ \texttt{value}\{B, 0, T\} \\ \Gamma' = \Gamma[x \mapsto mt\ \texttt{value}\{\mathsf{unboxed}, 0, T\}] \\ G := G[L \mapsto G(L) \sqcup \Gamma']\end{array}}{\Gamma, G, P \vdash \texttt{if\_unboxed}(x)\ \texttt{then}\ L, \Gamma[x \mapsto mt\ \texttt{value}\{\mathsf{boxed}, 0, T\}]}$$

Soundness

- We can prove soundness via standard subject-reduction techniques
  - Proof for restricted version of the system
- Theorem: If a program is well-typed, then it does not get stuck
  - I.e., OCaml data is never used at the wrong type

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

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 $P(f_i) = 0$

foo() ⟶ bar() ⟶ alloc_tuple()

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

Checking GC Safety with Effects

- Formally, use *effects* to check GC safety
  - Effects "may call GC" and "will not call GC"
  - Add to C function types:

$$ct \times \cdots \times ct \rightarrow_{GC} ct$$

$$GC \quad ::= \quad \gamma \mid \mathsf{gc} \mid \mathsf{nogc}$$

- Also uses standard liveness analysis
  - Don't warn about unprotected but dead locals

Type Rule

$$\textsc{App}\quad \frac{\begin{array}{c}\Gamma, P \vdash f : ct'_1 \times \cdots \times ct'_n \rightarrow_{GC'} ct \\ \Gamma, P \vdash e_i : ct_i\{B_i, 0, T_i\} \qquad ct_i = ct'_i \qquad i \in 1..n \\ \Gamma, P \vdash cur\_func : \cdot \rightarrow_{GC} \cdot \\ GC' \sqsubseteq GC \qquad \mathsf{gc} \sqsubseteq GC \Rightarrow (\mathit{ValPtrs}(\Gamma) \cap \mathit{live}(\Gamma)) \subseteq P\end{array}}{\Gamma, P \vdash f(e_1, \ldots, e_n) : ct\{\top, 0, \top\}}$$

Custom Types

- C data can be passed to OCaml opaquely
  - E.g., pointers to window or button objects
  - Assigned opaque type by programmer
- No guarantee types are used safely
  - Could perform C type cast by going through OCaml!
- Our systems extends ML types with C types:

$$ct \quad ::= \quad \mathtt{void} \mid \mathtt{int} \mid mt\ \mathtt{value} \mid ct\ *$$

$$mt \quad ::= \quad \alpha \mid mt \rightarrow mt \mid ct\ \mathsf{custom} \mid (\Psi, \Sigma)$$

Algorithm

- Apply type inference rules iteratively, until we reach a fixpoint with B, I, and T facts
  - Generates constraints *ct* = *ct*' and *mt* = *mt*'
    - Solved with standard type unification
  - Generates constraints *GC* ≤ *GC*'
    - Solved with reachability (atomic subtyping constraints/ qualifiers)
  - Also generates some additional constraints (not shown) that can be solved easily

Implementation: 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

Implementation: 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

Handling Features of C

- 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

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

Experimental Results

Program	C loc	OCaml loc	Time (s)	Errors	Warnings	False Pos	Imprecision
apm-1.00	124	156	1.3	0	0	0	0
camlzip-1.01	139	820	1.7	0	0	0	1
ocaml-mad-0.1.0	139	38	4.2	1	0	0	0
ocaml-ssl-0.1.0	187	151	1.5	4	2	0	0
ocaml-glpk-0.1.1	305	147	1.3	4	1	0	1
gz-0.5.5	572	192	2.2	0	1	0	1
ocaml-vorbis-0.1.1	1183	443	2.8	1	0	0	2
ftplib-0.12	1401	21	1.7	1	2	0	1
lablgl-1.00	1586	1357	7.5	4	5	140	20
cryptokit-1.2	2173	2315	5.4	0	0	0	1
lablgtk-2.2.0	5998	14847	61.3	9	11	74	48
Total				24	22	214	75

Note: Time includes compilation

Common Errors

- Forgetting to register C pointer to ML heap
  - 3 errors
- Forgetting to release a registered pointer
  - 2 errors
- Remainder are 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
    - OCaml: `external f : ?x: int -> unit = "f"`
    - C: `value f(value x) { int bar = Int_val(x); ... }`
  - Others similar

Warnings: Questionable Coding

- 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);`

Imprecision and False Positives

- Tags and offsets are sometimes Top
- Globals and function pointers
- Polymorphic variants
- Pointer arithmetic disguised as long arithmetic
  - (t*)v + 1 == (t*) (v + sizeof(t*))
    - Our system gets confused

Future Work

- Ensure immutable data not changed by C code
  - Could yield unexpected results
- Improved handling of polymorphic variants
  - Will require some programmer annotations
- Check safety of unsafe code within OCaml
- Extend to other FFIs

Conclusion

- FFIs are a useful part of a language
- FFI code is messy
  - But not complicated, hence analyzable
- Our system: A multi-lingual safety checker
  - The first we know of to check glue code
  - Shows that FFI need not compromise safety

http://www.cs.umd.edu/~furr/saffire/