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:
  \[ \text{external } \text{ml\_foo} : \text{int }\rightarrow \text{int\_list }\rightarrow \text{unit } = \text{“c\_foo”} \]

• C:
  \[ \text{value c\_foo(value int\_arg, value int\_list\_arg);} \]

  • 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

```plaintext
```

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

Example: “Pattern Matching”

```c
if (Is_long(x)) {
  if (Int_val(x) == 0)
    /* B */
  else {
    if (Tag_val(x) == 0)
      /* A */
    else {
      if (Tag_val(x) == 1)
        /* C */
      else {
        if (Tag_val(x) == 0)
          /* B */
        else {
          if (Tag_val(x) == 1)
            /* C */
          else {
            /* D */
          }
        }
      }
    }
  }
}
```

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

Overlap of Physical Representations

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

  ```
  if (tag == 0)
    /* int */
  else {
    if (tag == 0)
      /* unit */
    else {
      if (tag == 1)
        /* int */
      /* Bar of type t'' = Bar ... */
    }
  }
  ```
**Representational Types**

- **Representational** type \((C, S)\) models such data
  - \(C = \#\) of nullary constructors, 0 if none
  - \(S = \text{arg types of other constructors, 0 if none}\)

- **Examples**:
  - \(\text{int} \Rightarrow (\infty, 0)\)
  - \(\text{int} \times \text{int} \Rightarrow (0, (\infty, 0) \times (\infty, 0))\)
  - \(\text{type } t = A \text{ of } \text{int} | B | C \text{ of } \text{int} \times \text{int} | D \Rightarrow (2, (\infty, 0) + (\infty, 0) \times (\infty, 0))\)

---

**Multi-Lingual Type System**

\[
\begin{align*}
\text{ct} & ::= \text{void} | \text{int} | \text{mt} \text{ value} | \text{ct} \times \\
& \quad | \text{ct} \times \cdots \times \text{ct} \\
\text{mt} & ::= \alpha | \text{mt} \rightarrow \text{mt} | \text{ct} \text{ custom} | (\Psi, \Sigma) \\
\Psi & ::= \psi | n | T \\
\Sigma & ::= \sigma | 0 | \Pi + \Sigma \\
\Pi & ::= \pi | 0 | \text{mt} \times \Pi
\end{align*}
\]

---

**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

---

**Original Type Systems**

- **mltype** ::= unit | int | mlttype \times mlttype | \text{S} + \cdots + \text{S} | mlttype ref
- **mltype** \rightarrow mlttype
- **S** ::= Const | Const of mlttype

(a) OCaml Type Grammar

- **ctype** ::= void | int | value | ctype \times \\
  | ctype \times \cdots \times \text{ctype} \rightarrow \text{ctype}

(b) C Type Grammar

---

**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

---

**Dataflow Analysis**

- **Extend the C type** \text{value} to

\[
\begin{align*}
(C, S) \text{ value}[B][!] & \Rightarrow \text{boxed or unboxed} \\
\text{Representational Type} & \Rightarrow \text{boxed or unboxed} \\
\text{offset} & \Rightarrow \text{value (if int)} \\
\text{block tag (if ptr)} & \Rightarrow \text{(C, S) flow-insensitive (a value has one OCaml type)}
\end{align*}
\]

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

```plaintext
def value succ(value v) {
    v: (ψ', σ') value[Top0]{Top}
}
def int next = Int_val(v) + 1; σ'=0
return Val_int(next);
}
```

Inferring Sum Types

```plaintext
def if (is_long(x)) {
    if (Int_val(x) == 0) λ B σ
    if (Int_val(x) == 1)     λ D σ
} else {
    if (Tag_val(x) == 0)     λ A σ
    if (Tag_val(x) == 1)     λ C σ
}
```

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
Γ, P + n : int{T, 0, n}

VAL DEF EXP
Γ, P + e : mt value[boxed, n, m]
πm = α0 × ... × αn × π  ψ, π, σ, α, π fresh
Γ, P + e : αn value{T, 0, T}
```

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

```
Initial environment
Map from labels to environments
Output environment
```
Types Rules for Statements (cont’d)

SEQ STM
\[ Γ, G, P ⊢ s_1 ; s_2, Γ’ \]
\[ Γ, G, P ⊢ s_1 ; s_2, Γ’’ \]

LABEL STM
\[ G(L), G, P ⊢ s, Γ’ \]
\[ Γ ⊆ G(L) \]

GOTO STM
\[ G := G(L) ∪ G(Γ) \]
\[ Γ, G, P ⊢ \text{goto } L, \text{reset}(Γ) \]

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);
  bar(arg);
  CAMLlocal1(temp);
  temp = alloc_tuple(2);
  CAMLreturn(Val_unit);
}

- 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) = unprotected locals and parameters at call
  - Check: If path from f to function that may call GC, require P(f) = 0

foo( ) ———> bar( ) ———> alloc_tuple( )

\[ P(\text{foo}) = \{ \text{arg} \} \]
\[ \text{error: non-empty} \]
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:
  \[ \text{ct} \times \cdots \times \text{ct} \to \text{GC} \text{ ct} \]
- Also uses standard liveness analysis
  - Don’t warn about unprotected but dead locals

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:
  \[
  \begin{align*}
  \text{ct} &::= \text{void} | \text{int} | \text{mt value} | \text{ct} * \\
  \text{mt} &::= \alpha | \text{mt} \to \text{mt} | \text{ct custom} | (\Psi, \Sigma)
  \end{align*}
  \]

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

Type Rule

**APP**
\[
\begin{align*}
\Gamma, P \vdash f : &\text{ct}_1 \times \cdots \times \text{ct}_n \to \text{GC}\text{ ct} \\
\Gamma, P \vdash e_1 : &e_1 : \text{ct}_1 \{B_i, 0, X_i\} \\
\Gamma, P \vdash \text{car_func} &: \to \text{GC} \\
\text{GC} &\subseteq \text{GC} \\
\text{gc} &\subseteq \text{GC} \Rightarrow (\text{ValPre}(\Gamma) \land \text{live}(\Gamma)) \subseteq P \\
\Gamma, P \vdash f(e_1, \ldots, e_n) : &e_1 \{\Gamma, 0, \top\}
\end{align*}
\]

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 \( GC \leq GC' \)
    - Solved with reachability (atomic subtyping constraints/qualifiers)
  - Also generates some additional constraints (not shown) that can be solved easily

Implementation: Phase 2, C

- Second tool built using CIL
  - This is the tool that issues warnings etc.
  - \text{Int}_\text{val()}, \text{Tag}_\text{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

<table>
<thead>
<tr>
<th>Program</th>
<th>1 source</th>
<th>OCaml src</th>
<th>Time (s)</th>
<th>Warnings</th>
<th>Time (s)</th>
<th>False Poses</th>
<th>Imprecisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>mustache</td>
<td>116</td>
<td>129</td>
<td>1.5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>wsdchrome</td>
<td>102</td>
<td>105</td>
<td>1.5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>webstart</td>
<td>72</td>
<td>101</td>
<td>1.5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mustache</td>
<td>87</td>
<td>119</td>
<td>1.5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>webstart</td>
<td>70</td>
<td>105</td>
<td>1.5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>vips</td>
<td>709</td>
<td>703</td>
<td>2.5</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libjpeg</td>
<td>100</td>
<td>97</td>
<td>2.5</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>xz</td>
<td>2012</td>
<td>1915</td>
<td>6.1</td>
<td>1</td>
<td>2</td>
<td>73</td>
<td>14</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td>2121</td>
<td></td>
<td>21</td>
<td>21</td>
<td>204</td>
<td>30</td>
</tr>
</tbody>
</table>

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^n)//l == (t^n) (v + sizeof(t^n))
    - 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/