ctypes: foreign calls in your native language

Jeremy Yallop
University of Cambridge

Saying hello to C

The ctypes library provides a typed, high-level interface for describing C types, accessing C data, and calling C functions. Using ctypes, you can bind to foreign functions without writing or generating C.

Getting started with ctypes is straightforward. The "Hello, C" of foriegn function calls involves one line of code to give the name and type of the printing function `puts` and a second line to call the function:

```
# let puts = foreign "puts" (string @→ returning int);;
val puts : string → int = <fun>
# puts "Hello, C!";;
Hello, C!
```

Getting C to give you the time of day

C libraries often expose structured types for passing or returning data. We'll use the POSIX time interface as a typical example, presenting C and OCaml in a two-column format. The left-hand side shows part of the POSIX interface, while the right-hand side gives the corresponding ctypes code that describes the various structs and functions in OCaml.

```
struct timeval {
  time_t tv_sec;
  suseconds_t tv_usec;
};

struct timezone;

int gettimeofday(struct timeval *tv,
                 struct timezone *tz);

char *ctime(const time_t *timep);
```

```
let timeval = structure "timeval"
let tv_sec = timeval *:* time_t
let tv_usec = timeval *:* suseconds_t
let () = seal timeval

let timezone = structure "timezone"

let gettimeofday = foreign "gettimeofday"
  (ptr timeval @→ ptr timezone @→ returning int)

let ctime = foreign "ctime" (ptr time_t @→ returning string)
```

Having described the relevant types and functions to OCaml, we now have everything we need to allocate a `struct timeval` value, pass its address to `gettimeofday`, and display the result. The `make`, `addr` and `(@.)` functions respectively allocate a struct, take its address, and access one of its fields.

```
# let tv = make timeval in begin
    ignore (gettimeofday (addr tv) (from_voidp timezone null));
    Printf.sprintf "The date and time are %s" (ctime (tv @. tv_sec))
  end
- : string = "The date and time are Fri Jun 7 18:32:28 2013"
```

Asking C to call you back

C functions which accept function pointers as arguments can be difficult to use from OCaml. A function pointer is a reference to one of a statically-determined set of global names; it is too narrow an interface to readily accept OCaml functions, which can be created dynamically. However, ctypes makes passing OCaml functions to C as straightforward as passing first-order values, as the next example demonstrates. Our example is based on `qsort`, the canonical example of a function pointer consumer. The C interface and the ctypes bindings again occupy the left and right columns respectively.

```
typedef int cmp(void *l, void *r);

void qsort(void *p, size_t els, size_t sz, cmp *compare);
```

```
let cmp = ptr void @→ ptr void @→ returning int

let qsort = foreign "qsort"
 (ptr void @→ size_t @→ size_t @→ funptr cmp @→
 returning void)
```

Calling `qsort` is not difficult, but involves a number of coercions. We need to pass `size_t` values for the element size and array length, but our `sizeof` and `Array.length`[1] functions return `int`. Similarly, `qsort` expects an untyped pointer to the first element of the array, but the pointer returned by `Array.start` is typed. We also need to coerce the arguments to the callback function from generic pointers to pointers of the appropriate type. We can hide all these coercions inside a function with a simple interface:

```
let sort : α. α array → (α → α → int) → unit
  = fun arr cmp →
      let ty = Array.element_type arr in
        qsort
          (to_voidp (Array.start arr))
          (Size_t.of_int (Array.length arr))
          (Size_t.of_int (sizeof ty))
          (fun l r → cmp (!@(from_voidp ty l)) (!@(from_voidp ty r)))
```

Now we can sort arrays using the C `qsort` function and an OCaml comparison function:

```
# let int_arr = Array.of_list int [5; 3; 1; 2; 4] and str_array = Array.of_list string ["one"; "two"; "three"; "four"]
val int_arr : int array = <abstr>
val str_arr : string array = <abstr>
# sort int_arr Pervasives.compare; Array.to_list int_arr
− : int list = [1; 2; 3; 4; 5]
# sort str_arr (fun l r → Pervasives.compare r l); Array.to_list str_arr
− : string list = ["two"; "three"; "one"; "four"]
```

Related work

The central idea behind ctypes —writing OCaml-to-C bindings in OCaml rather than in C— has been used successfully in other languages. For example, James Bielman's *Common Foreign Function Interface* for Common Lisp, Matthias Blume's *No-Longer-Foreign Function Interface* for SML and Thomas Heller's *ctypes* library for Python (from which this OCaml library draws both its name and its inspiration) are all based on this approach.

Future plans

We have a number of directions in mind for future development.

Performance is naturally a concern for library that is intended to replace C code. While ctypes has been written with performance in mind, a certain level of interpretative overhead remains which we are keen to eliminate. Here the generality of the library puts it at a disadvantage compared with hand-written C, since the type of foreign functions is not known to ctypes until runtime. We are investigating an solution based on MetaOCaml-style staging and offshoring, i.e. specialising each foreign function binding once the type *is* known, before the function is called.

We are also investigating an OCaml-Java backend for C types. The prospect of having C bindings written for OCaml work without modification on OCaml-Java is appealing, and initial experiments seem encouraging.

A third area for development involves taking advantage of the greater abstraction introduced by ctypes to give stronger guarantees about calls to C. For example, having ctypes remap the pages of the OCaml heap as read-only when dropping into a C call might help to catch inadvertent modifications.

[1]The name `Array` in this abstract refers to the ctypes array module, not to the standard library module.