Interfacing C with OCaml

Niki Yoshiuchi

November 30, 2011

Overview

- Why?
- A Quick Example
  - C Code
  - OCaml Code
  - Compiling
- Function signatures
  - Arity > 5
- value type
  - Integral types
  - Blocks
  - Structured data
- Linking
  - Static
  - Dynamic
- What's left?

Why?

- Bindings to C libraries
- Multi-core threading
  - If the C code doesn't require the OCaml runtime we can release the lock
- Breaking the type system
  - If you are going to do this, it's probably better to use the Obj module

A Quick Example: C Code

```
1  #include <caml/memory.h>
2  #include <caml/mlvalues.h>
3
4  CAMLprim value caml_add(value a, value b)
5  {
6      CAMLparam2(a, b);
7      int c;
8      c = Int_val(a) + Int_val(b);
9      CAMLreturn(Val_int(c));
10 }
```

A Quick Example: OCaml Code

```
1 external add : int -> int -> int = "caml_add"
2
3 let _ =
4     let three = add 1 2 in
5     print_int three;
6     print_newline ()
```

A Quick Example: Compiling

```
1 gcc -I /usr/local/lib/ocaml -c stubs.c
2 ocamlc -c main.ml
3 ar rcs libstubs.a stubs.o
4 ocamlc -o main.byte -custom libstubs.a main.cmo
```

Function Signatures

The OCaml code is simple - declare your function using the *external* keyword, specify its type and set it equal to the name of the C implementation.

- OCaml:

```
external name : type = "C-function-name"
```

- C:

```
CAMLprim value name(value arg1, value arg2)
```

Arity > 5

For functions with Arity n > 5, we must implement two C functions - one with n arguments and one with an array. The former is used for native compilation, the latter for byte code compilation.

- OCaml:

```
external name : type = "byte-code-name"
    "native-code-name"
```

- C (native):

```
CAMLprim value native_code(value arg1, value arg2,
value arg3, value arg4, value arg5, value arg6)
```

- C (byte):

```
CAMLprim value byte_code(value *argv, int argc) {
    native_code(argv[0], argv[1], argv[2],
    argv[3], argv[4], argv[5]);
}
```

value type

All variables passed between OCaml and C are of type *value*. A *value* can be:

- A integral type (unboxed)
- A pointer to a heap allocated object
- An pointer to a object allocated outside of the heap

There are two macros to help you determine what type is encapsulated in a *value*

- Is_long(v) - returns true if v is an integral type
- Is_block(v) - returns true if v is a pointer to a block

integral types

There are a number of macros for retreiving and storing integral types. They are of the form *Val_type* (think "value of type") and *Type_val* (think "type of value")

- Val_long(l)
- Val_int(i)
- Val_bool(b)
- Val_true
- Val_false
- Val_unit
- Long_val(v)
- Int_val(i)
- Bool_val(v)

Going back to our quick example

Let's take another look at our quick example to see how these macros are used:

```
1  #include <caml/memory.h>
2  #include <caml/mlvalues.h>
3
4  CAMLprim value caml_add(value a, value b)
5  {
6      CAMLparam2(a, b);
7      int c;
8      c = Int_val(a) + Int_val(b);
9      CAMLreturn(Val_int(c));
10 }
```

Blocks

If a *value* is a block, then it has a tag (accessible via the *Tag_val(v)* macro) which will return one of the following values:

Tag	Description
0 to No_scan_tag-1	A structured block. Each field is a *value*.
Closure_tag	A closure representing a functional value.
String_tag	A character string.
Double_tag	A double precision float.
Abstract_tag	An abstract datatype.
Custom_tag	A custom datatype.

Doubles, strings, Ints

Doubles, strings, Int32s, Int64s and Nativeints are all stored as pointers to heap allocated objects. They are accessible via a set of macros of the form *Type_val*:

- Double_val(v)
- String_val(v)
- Int32_val(v)
- Int64_val(v)
- Nativeint_val(v)

Allocating doubles, strings and Ints

In order to pass a double, string or Int32/Int64/Nativeint from C to OCaml, you must allocate an object on OCaml's heap.

- caml_alloc(n, t) - Allocates a block with tag *t* and size *n*
- caml_alloc_string(n) - Allocates a string of length n
- caml_copy_string(s) - Copies the null-terminated string s
- caml_copy_double(d)
- caml_copy_int32(i), caml_copy_int64(i), caml_copy_nativeint(i)

Garbage Collection

If your C code manipulates any heap allocated objects or performs any allocations, you must register it with the garbage collector. OCaml provides a number of macros:

- CAMLparam0 to CAMLparam5 - registers 0 to 5 parameters
- CAMLxparam1 to CAMLxparam5 - for registering additional parameters for functions with arity > 5
- CAMLlocal1 to CAMLlocal5 - for declaring local *value* variables
- CAMLlocalN(x, n) - for declaring an array of *values* of size *n*
- CAMLreturn0, CAMLreturn(x) and CAMLreturnT(t, x) - these replace the *return* statement

Garbage Collection

- CAMLparam# and CAMLxparam# should be the first things called in your function. All your function's parameters should be registered using them. If you have 3 parameters, then use CAMLparam3, etc.
- CAMLlocal# and CAMLlocalN should be used to declare any local *value* variables used in yoru function.
- CAMLreturn should replace all the *return* statements in your function. This is required if you used any of the CAMLparam or CAMLlocal functions.

Double example

```
1  #include <caml/mlvalues.h>
2  #include <caml/memory.h>
3  #include <caml/alloc.h>
4
5  CAMLprim value caml_double_add(value dbl1,
6          value dbl2)
7  {
8      CAMLparam2(dbl1, dbl2);
9      CAMLlocal1(ret);
10     double d;
11     d = Double_val(dbl1) + Double_val(dbl2);
12     ret = caml_copy_double(d);
13     CAMLreturn(ret);
14 }
```

Structured Data

- Variants
- Records and tuples
- Lists
- Arrays
- Big Arrays

Variants

- Constant constructors are represented as unboxed integers. The value is the order in which the constructor appears, starting from 0.
- Non-constant constructors are represented as blocks. The tag for each block is the order in which the non-constant constructor appears, starting from 0.

Variant Example

```
1  type t =
2      | Nothing (* Val_int(0) *)
3      | Int of int (* Block with tag 0 *)
4      | Double of float (* Block with tag 1 *)
5      | String of string (* Block with tag 2 *)
6      | Something (* Val_int(1) *)
7      | Variant of t * t (* Block with tag 3 *)
8
9  external variant : t -> unit = "caml_variant"
10
11 let _ =
12     let v = Variant (Int 5,
13         Variant (String "Hello",
14         Nothing)) in
15     variant v
```

Variant example

```c
#include <stdio.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/alloc.h>

CAMLprim value caml_variant(value t) {
    CAMLparam1(t);
    if(Is_block(t)) {
        switch(Tag_val(t)) {
            case 0:
                printf("Int of %d\n", Int_val(Field(t, 0)));
                break;
            case 1:
                printf("Double of %f\n", Double_val(Field(t, 0)));
                break;
            case 2:
                printf("String of %s\n", String_val(Field(t, 0)));
                break;
            case 3:
                printf("Begin Variant of\n");
                caml_variant(Field(t, 0));
                caml_variant(Field(t, 1));
                printf("Variant\n");
                break;
        }
    }
    else {
        if(Int_val(t) == 0)
            printf("Nothing\n");
        else if(Int_val(t) == 1)
            printf("Something\n");
    }
    CAMLreturn(Val_unit);
}
```

Tuples and Records

Tuples and records are both represented by 0 tagged blocks. Fields are accessed in numerical order using the *Field(v, n)* macro. *Field(v, n)* returns an lvalue and can be used to both set and get the field. Tuples and records can be allocated using the *caml_alloc_tuple(n)* function.

Record example

```
1  type t = {first : string; second : float}
2
3  external create : string -> float -> t =
4      "caml_create"
5
6  let _ =
7      let r = create "Hello" 3.1415 in
8      print_endline r.first;
9      print_float r.second;
10     print_newline ()
```

Record example

```
1  #include <caml/mlvalues.h>
2  #include <caml/memory.h>
3  #include <caml/alloc.h>
4
5  CAMLprim value caml_create(value str, value fl) {
6      CAMLparam2(str, fl);
7      CAMLlocal1(ret);
8
9      ret = caml_alloc_tuple(2);
10     Field(ret, 0) = str;
11     Field(ret, 1) = fl;
12
13     CAMLreturn(ret);
14 }
```

Lists

- [] is a Val_int of 0
- Blocks with tag 0
- Block size of 2
- Field(v, 0) contains the head
- Field(v, 1) contains the tail

List example

```
1 external print_list : int list -> unit =
2     "caml_print_list"
3
4 let () = print_list [1;2;3;4;5]
```

List example

```
1  #include <stdio.h>
2  #include <caml/mlvalues.h>
3  #include <caml/memory.h>
4  #include <caml/alloc.h>
5
6  CAMLprim value caml_print_list(value list) {
7      CAMLparam1(list);
8      CAMLlocal2(head, tail);
9
10     if(Is_block(list))
11     {
12         head = Field(list, 0);
13         tail = Field(list, 1);
14         printf("%d ", Int_val(head));
15         CAMLreturn(caml_print_list(tail));
16     }
17
18     printf("\n");
19     CAMLreturn(Val_unit);
20 }
```

Arrays

- Block with tag 0
- Use the *Wosize_val(v)* macro to get the size of the array
- Use Field(v, n) to get or set the nth element
- caml_alloc_tuple(n) to allocate an array of size n
- Double arrays
  - Block with tag Double_array_tag
  - Use Double_field(v, n) to get the nth element
  - Store_double_field(v, n, d) to set the nth element
  - caml_alloc(n, Double_array_tag) to allocate an array of size n

Array example

```
1 external create_array : int -> int array =
2     "caml_create_array"
3 external print_array : int array -> unit =
4     "caml_print_array"
5
6 let _ =
7     let a = create_array 5 in
8     print_array a
```

Array example

```
1  #include <stdio.h>
2  #include <caml/mlvalues.h>
3  #include <caml/memory.h>
4  #include <caml/alloc.h>
5
6  CAMLprim value caml_create_array(value size)
7  {
8      CAMLparam1(size);
9      CAMLlocal1(ret);
10     int i;
11     ret = caml_alloc_tuple(Int_val(size));
12     for(i=0; i<Int_val(size); ++i)
13         Field(ret, i) = Val_int(i);
14     CAMLreturn(ret);
15 }
16
17 CAMLprim value caml_print_array(value a)
18 {
19     CAMLparam1(a);
20     int size, i;
21     size = Wosize_val(a);
22     for(i=0; i<size; ++i)
23         printf("%d ", Int_val(Field(a, i)));
24     printf("\n");
25     CAMLreturn(Val_unit);
26 }
```

Double array example

```
1 external create_array : int -> float ->
2     float array = "caml_create_array"
3 external print_array : float array -> unit =
4     "caml_print_array"
5
6 let _ =
7     let a = create_array 5 0.5 in
8     print_array a
```

Double array example

```
1  #include <stdio.h>
2  #include <caml/mlvalues.h>
3  #include <caml/memory.h>
4  #include <caml/alloc.h>
5
6  CAMLprim value caml_create_array(value size, value inc)
7  {
8      CAMLparam2(size, inc);
9      CAMLlocal1(ret);
10     double d, step;
11     int i;
12     ret = caml_alloc(Int_val(size), Double_array_tag);
13     d = 0.0;
14     step = Double_val(inc);
15     for(i=0; i<Int_val(size); ++i)
16         Store_double_field(ret, i, d + i*step);
17     CAMLreturn(ret);
18 }
19
20 CAMLprim value caml_print_array(value a)
21 {
22     CAMLparam1(a);
23     int size, i;
24     size = Wosize_val(a);
25     for(i=0; i<size; ++i)
26         printf("%f ", Double_field(a, i));
27     printf("\n");
28     CAMLreturn(Val_unit);
29 }
```

Bigarrays

- Native to C (or Fortran)
- pack data efficiently
- Are limited to only certain types, mainly integers, floats, chars and complex numbers
- in OCaml there are three different Bigarrays:
  - Array1 - Single dimension arrays
  - Array2 - Two dimensional arrays
  - Genarray - Multi-dimensional arrays
- C doesn't care. You get a void * and num_dims and flags

What do Bigarrays look like?

This is from OCaml 3.12.1:

```
1 struct caml_ba_array {
2   void * data;
3   intnat num_dims;
4   intnat flags;
5   struct caml_ba_proxy * proxy;
6   intnat dim[1] ;
7 };
```

Bigarray - types

- CAML_BA_FLOAT32 - Single-precision floats
- CAML_BA_FLOAT64 - Double-precision floats
- CAML_BA_SINT8 - Signed 8-bit integers
- CAML_BA_UINT8 - Unsigned 8-bit integers
- CAML_BA_SINT16 - Signed 16-bit integers
- CAML_BA_UINT16 - Unsigned 16-bit integers
- CAML_BA_INT32 - Signed 32-bit integers
- CAML_BA_INT64 - Signed 64-bit integers
- CAML_BA_CAML_INT - Caml-style integers (signed 31 or 63 bits)
- CAML_BA_NATIVE_INT - Platform-native long integers (32 or 64 bits)
- CAML_BA_COMPLEX32 - Single-precision complex
- CAML_BA_COMPLEX64 - Double-precision complex

Bigarray - layouts and flags

OCaml supports both row-major (C style) and column-major (Fortan style) arrays. This is specified by one of two flags:

- CAML_BA_C_LAYOUT
- CAML_BA_FORTRAN_LAYOUT

Bigarrays also have flags specifying how the memory is managed:

- CAML_BA_EXTERNAL - Data is not allocated by Caml
- CAML_BA_MANAGED - Data is allocated by Caml
- CAML_BA_MAPPED_FILE - Data is a memory mapped file

Bigarrays - allocation

There are two functions provided to allocate Bigarrays in C:

- value caml_ba_alloc(int flags, int num_dims, void * data, intnat * dim);
  - flags - The type and layout ORed together
  - num_dims - the number of dimensions
  - data - a pointer to the C array
  - dim - a pointer to an array containing the size of each dimension
- value caml_ba_alloc_dims(int flags, int num_dims, void * data, ...);
  - Pretty much the same, except it takes a va_args list of dimensions

Bigarrays - access

There are two macros - one returns the Bigarray struct, the other the data pointer

- Caml_ba_array_val(v) - returns the struct
- Caml_ba_data_val(v) - returns the data pointer

Bigarray example

```
1  open Bigarray
2
3  external create_ba : int -> int ->
4      (int, int64_elt, c_layout) Array2.t
5      = "caml_create_ba"
6
7  external print_ba :
8      (int, int64_elt, c_layout) Array2.t -> unit
9      = "caml_print_ba"
10
11 let _ =
12     let ba = create_ba 5 5 in
13     print_ba ba
```

Bigarray example

```c
#include <stdio.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/alloc.h>
#include <caml/bigarray.h>

CAMLprim value caml_create_ba(value w, value h)
{
    CAMLparam2(w, h);
    CAMLlocal1(ret);
    int i, width, height;
    long *data;

    width = Int_val(w);
    height = Int_val(h);
    /* oh noes!  leaking memory! */
    data = malloc(sizeof(long)*width*height);
    for(i=0; i<width*height; ++i)
        data[i] = i;
    ret = caml_ba_alloc_dims(CAML_BA_INT64 | CAML_BA_C_LAYOUT,
            2, data, width, height);
    CAMLreturn(ret);
}

CAMLprim value caml_print_ba(value ba)
{
    CAMLparam1(ba);
    int width, height;
    int x, y;
    long *data;
    width = Caml_ba_array_val(ba)->dim[0];
    height = Caml_ba_array_val(ba)->dim[1];
    data = Caml_ba_data_val(ba);
    for(y=0; y<height; ++y) {
        for(x=0; x<width; ++x) {
            printf("%d ", data[y*width + x]);
        }
        printf("\n");
    }
    CAMLreturn(Val_unit);
}
```

Linking

Use Oasis.

Static Linking

To build the C library:

```
gcc -I /usr/local/lib/ocaml -c stubs.c
ar rcs libstubs.a stubs.o
```

To link the OCaml code:

```
ocamlc -o main.byte -custom libstubs.a main.cmo
ocamlopt -o main.native libstubs.a main.cmx
```

Dynamic Linking

To build the C library:

```
gcc -I /usr/local/lib/ocaml -c -fPIC
    stubs.c -o stubs.o
gcc -shared -Wl,-soname,libstubs.so -o libstubs.so
```

To link the OCaml code:

```
ocamlc -o main.byte libstubs.so main.cmo
ocamlopt -o main.native libstubs.so main.cmx
```

What's left?"

- Callbacks
- Exceptions
- Custom data
- Non-blocking calls
- C threads

Fin.

Thanks for coming!