Interoperability

OCaml PRO – Grégoire Henry

March 9-13 2015

Outline

Low-level interface with C
Generating bindings

Low-level interface with C

Part of the standard library is defined as **external function call**, using the C ABI. For instance, from the `stdlib/sys.ml` file:

```
1 : external getenv: string -> string = "caml_sys_getenv"
```

And from the `byterun/sys.c`, that is linked by default with the standard runtime (`libbyterun.o` or `libasmrun.o`):

```
1 : CAMLprim value caml_sys_getenv(value var)
2 : {
3 :   char * res = getenv(String_val(var));
4 :   if (res == 0) caml_raise_not_found();
5 :   return caml_copy_string(res);
6 : }
```

The C function should as many parameters than the OCaml one. Due to the uniform representation there is only one C type for all OCaml value.

The virtual machine is not able to handle function with more than 5 arguments. For such functions, we have to define two C primitives:

```
1 : value prim_nat (value x1, ..., value xn) { ... }
2 : value prim_byt (value *tab, int n)
3 : { return prim_nat(tab[0],tab[1],...,tab[n-1]) ; }
```

and, in OCaml side, we have to precise both primitive names:

```
1 : external ocaml_name : type = "prim_byt" "prim_nat"
```

Call to non-allocating external function may be optimised by the native compiler:

```
1 : external ocaml_name : type = "prim" "noalloc"
2 : external ocaml_name : type = "prim_byt" "noalloc" "prim_nat"
```

When a function has `float` parameters or return value, the native primitive may use unboxed float:

```
1 : external add_float
2 :   : float -> int -> float
3 :   = "add_byt" "add_nat" "float"
```

```
1 : float add_nat(float x, value y) {
2 :   return (x + (float)Long_val(y));
3 : }
4 : value add_byt(value x, value y) {
5 :   return caml_copy_double(Double_val(x), y);
6 : }
```

The OCaml runtime provides functions and macros to introspect, allocate, and mutate values. For instance, from `mlvalues.h`:

```
1 : #define Is_long(x)   (((x) & 1) != 0)
2 : #define Is_block(x)  (((x) & 1) == 0)
3 : #define Val_long(x)     (((intnat)(x) << 1) + 1)
4 : #define Long_val(x)     ((x) >> 1)
```

Or from `byterun/alloc.h`:

```
1 : CAMLextern value caml_alloc (mlsize_t, tag_t);
2 : CAMLextern value caml_alloc_small (mlsize_t, tag_t);
```

Or from `byterun/memory.h`:

```
1 : CAMLexport void caml_modify (value *fp, value val)
```

Out of heap pointer are not valid OCaml value.
Still, they may be wrapped into a **custom block**.

Blocks with tag `Custom_tag` contain both arbitrary user data and a pointer to a C struct, with type `struct custom_operations`, that associates user-provided finalization, comparison, hashing, serialization and deserialization functions to this block.

Custom block example, from byterun/io.c:

```
1 : static struct custom_operations channel_operations = {
2 :   "_chan",
3 :   caml_finalize_channel, compare_channel, hash_channel,
4 :   custom_serialize_default, custom_deserialize_default,
5 :   custom_compare_ext_default
6 : };
7 :
8 : CAMLexport value caml_alloc_channel(struct channel *chan)
9 : {
10 :   value res;
11 :   chan->refcount++;
12 :   res = caml_alloc_custom(&channel_operations,
13 :                           sizeof(struct channel *),
14 :                           1, 1000);
15 :   Data_custom_val(res) = (void *)chan;
16 :   return res;
17 : }
```

Being cooperative with the GC

The most difficult part: cooperating with a GC that may move value!

```
1 : value alloc_list(value pair) /* Contrived example! */
2 : {
3 :   CAMLparam0 (pair);
4 :   CAMLlocal2 (tail, r);
5 :
6 :   r = caml_alloc_small(2, 0); /* minor heap allocation */
7 :   Field(r, 0) = Field(pair, 0);
8 :   Field(r, 1) = Val_int(0);
9 :
10 :  tail = caml_alloc_shr(2, 0); /* major heap allocation */
11 :  caml_initialize(&Field(tail, 0), Field(pair, 1));
12 :  caml_initialize(&Field(tail, 1), Val_int(0));
13 :
14 :  caml_modify(&Field(r, 1), tail);
15 :  CAMLreturn (r);
16 : }
```

Be sure to respect the 6 rules of the OCaml manual.

Exceptions

Raising predefined exception, from byterun/fail.h:

```
1 : CAMLextern void caml_failwith (char const *);
2 : CAMLextern void caml_invalid_argument (char const *);
```

Raising custom exceptions:

```
1 : exception My_exception of int
2 : let () = Callback.register_exception
3 :              "My_module.my_exception" (My_exception 0)
```

```
1 : void raise_my_exn(int arg) {
2 :   static value * exn = NULL;
3 :   if (exn == NULL) {
4 :     /* First time around, look up by name */
5 :     exn = caml_named_value("My_module.my_exception");
6 :   }
7 :   raise_with_arg(*exn, Val_int(arg));
8 : }
```

Callback from C to OCaml

Calling an OCaml function from C, from byterun/callback.h:

```
value caml_callback (value closure, value arg);
value caml_callback2 (value closure, value arg1, value arg2);
```

Example:

```
let f x = Printf.printf "f is applied to %d\n%!" d
let _ = Callback.register "test function" f
```

```
void call_caml_f(int arg) {
  caml_callback(*caml_named_value("test function"),
                Val_int(arg));
}
CAMLprim value caml_apply(value vf, value vx) {
  CAMLparam2(vf, vx);
  CAMLlocal1(vy);
  vy = caml_callback(vf, vx);
  CAMLreturn(vy);
}
```

Blocking section

From stdlib/sys.ml:

```
1 : external chdir: string -> unit = "caml_sys_chdir"
```

From byterun/sys.c:

```
1 : CAMLprim value caml_sys_chdir(value dirname)
2 : {
3 :   CAMLparam1(dirname);
4 :   char * p;
5 :   int ret;
6 :   p = caml_strdup(String_val(dirname));
7 :   caml_enter_blocking_section();
8 :   ret = chdir(p);
9 :   caml_leave_blocking_section();
10 :  caml_stat_free(p);
11 :  if (ret != 0) caml_sys_error(dirname);
12 :  CAMLreturn(Val_unit);
13 : }
```

Code review

- byterun/compare.c
- byterun/intern.c
- byterun/hash.c

How to link...

- `ocamlopt -o test test.cmx my_c_file.o -cclib -lmylib`
- `ocamlopt -o test test.ml my_c_file.c -cclib -lmylib`

Generating bindings

Bindings may be generated, for instance with the `camlidl` (unmaintened for 10 years but still working!).

```
1 : quote(C,"#include <stdlib.h>")
2 :
3 : [string] char * getenv([in,string] char * varname);
4 :
5 : void putenv([in,string] char * name_val);
6 :
7 : int execv([in,string] char * path,
8 :           [in,null_terminated,string*] char ** argv);
```

Bindings may be generated, for instance with the more recent ctypes library.
This library is also able to use `libffi` and dlopen to dynamically generate bindings.
For instance, the following toplevel session:

```
# #use "topfind";;
# #require "ctypes.foreign";;
# #require "ctypes.top";;
# open Ctypes ;;
# open PosixTypes ;;
# open Foreign ;;
# let getenv =
    foreign "getenv" (string @-> returning string);;
val getenv : string -> string = <fun>
# getenv "USER";;
- : string = "henry"
```