12

Interoperability with C

Developing programs in a given language very often requires one to integrate libraries written in other languages. The two main reasons for this are:

- to use libraries that cannot be written in the language, thus extending its functionality;
- to use high-performance libraries already implemented in another language.

A program then becomes an assembly of software components written in various languages, where each component has been written in the language most appropriate for the part of the problem it addresses. Those software components interoperate by exchanging values and requesting computations.

The Objective Caml language offers such a mechanism for interoperability with the C language. This mechanism allows Objective Caml code to call C functions with Caml-provided arguments, and to get back the result of the computation in Objective Caml. The converse is also possible: a C program can trigger an Objective Caml computation, then work on its result.

The choice of C as interoperability language is justified by the following reasons:

- it is a standardized language (ISO C);
- C is a popular implementation language for operating systems (Unix, Windows, MacOS, etc.);
- a great many libraries are written in C;
- most programming languages offer a C interface, thus it is possible to interface Objective Caml with these languages by going through C.

The C language can therefore be viewed as the esperanto of programming languages.

Cooperation between C and Objective Caml raises a number of difficulties that we review below.

- **Machine representation of data**
  For instance, values of base types (*int*, *char*, *float*) have different machine representations in the two languages. This requires conversion between the representations, in both directions. The same holds for data structures such as records, sum types[1], or arrays.
- **The Objective Caml garbage collector**
  Standard C does not provide garbage collection. (However, garbage collectors are easily written in C.) Moreover, calling a C function from Objective Caml must not modify the memory in ways incompatible with the Objective Caml GC.
- **Aborted computations**
  Standard C does not support exceptions, and provides different mechanisms for aborting computations. This complicates Objective Caml's exception handling.
- **Sharing common resources**
  For instance, files and other input-output devices are shared between Objective Caml and C, but each language maintains its own input-output buffers. This may violate the proper sequencing of input-output operations in mixed programs.

Programs written in Objective Caml benefit from the safety of static typing and automatic memory management. This safety must not be compromised by improper use of C libraries and interfacing with other languages through C. The programmer must therefore adhere to rather strict rules to ensure that both languages coexist peacefully.

Chapter outline

This chapter introduces the tools that allow interoperability between Objective Caml and C by building executables containing code fragments written in both languages. These tools include functions to convert between the data representations of each language, allocation functions using the Objective Caml heap and garbage collector, and functions to raise Objective Caml exceptions from C.

The first section shows how to call C functions from Objective Caml and how to build executables and interactive toplevel interpreters including the C code implementing those functions. The second section explores the C representation of Objective Caml values. The third section explains how to create and modify Objective Caml values from C. It discusses the interactions between C allocations and the Objective Caml garbage collector, and presents the mechanisms ensuring safe allocation from C. The fourth section describes exception handling: how to raise exceptions and how to handle them. The fifth section reverses the roles: it shows how to include Objective Caml code in an application whose main program is written in C.

1. Objective Caml's sum types are discriminated unions. Refer to chapter 2, page 45 for a full description.

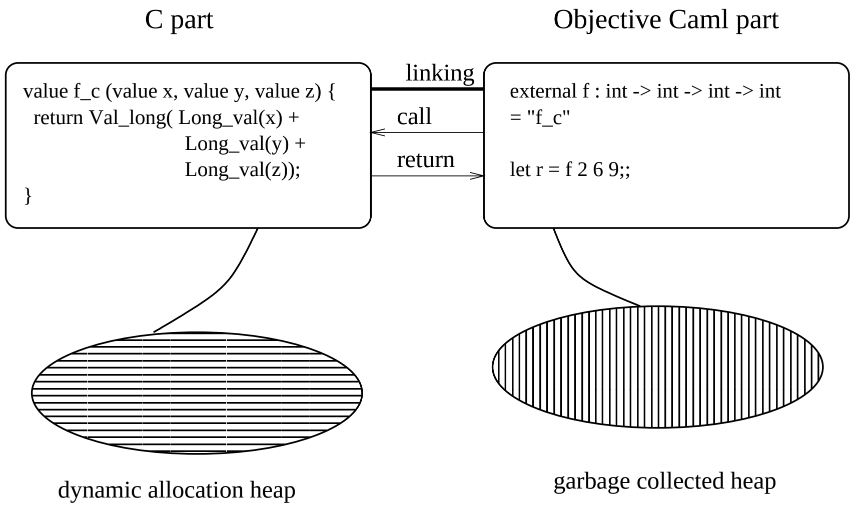
Note

This chapter assumes a working knowledge of the C language. Moreover, reading chapter 9 can be helpful in understanding the issues raised by automatic memory management.

Communication between C and Objective Caml

Communication between parts of a program written in C and in Objective Caml is accomplished by creating an executable (or a new toplevel interpreter) containing both parts. These parts can be separately compiled. It is therefore the responsibility of the linking phase[2] to establish the connection between Objective Caml function names and C function names, and to create the final executable. To this end, the Objective Caml part of the program contains external declarations describing this connection.

Figure 12.1 shows a sample program composed of a C part and an Objective Caml part. Each part comprises code (function definitions and toplevel expressions for Objective



```
value f_c (value x, value y, value z) {
  return Val_long( Long_val(x) +
                   Long_val(y) +
                   Long_val(z));
}
```

```
external f : int -> int -> int -> int
= "f_c"

let r = f 2 6 9;;
```

Figure 12.1: Communication between Objective Caml and C.

Caml) and a memory area for dynamic allocation. Calling the function `f` with three Objective Caml integer arguments triggers a call to the C function `f_c`. The body of the C function converts the three Objective Caml integers to C integers, computes their sum, and returns the result converted to an Objective Caml integer.

2. Linking is performed differently for the bytecode compiler and the native-code compiler.

We now introduce the basic mechanisms for interfacing C with Objective Caml: external declarations, calling conventions for C functions invoked from Objective Caml, and linking options. Then, we show an example using input-output.

External declarations

External function declarations in Objective Caml associate a C function definition with an Objective Caml name, while giving the type of the latter.

The syntax is as follows:

Syntax : $\boxed{\texttt{external}\ \textit{caml\_name}\ \texttt{:}\ \textit{type}\ \texttt{= "}\textit{C\_name}\texttt{"}}$

This declaration indicates that calling the function *caml_name* from Objective Caml code performs a call to the C function *C_name* with the given arguments. Thus, the example in figure 12.1 declares the function `f` as the Objective Caml equivalent of the C function `f_c`.

An external function can be declared in an interface (*i.e.*, in an `.mli` file) either as an external or as a regular value:

Syntax :

```
external caml_name : type = "C_name"
val caml_name : type
```

In the latter case, calls to the C function first go through the general function application mechanism of Objective Caml. This is slightly less efficient, but hides the implementation of the function as a C function.

Declaration of the C functions

C functions intended to be called from Objective Caml must have the same number of arguments as described in their external declarations. These arguments have type `value`, which is the C type for Objective Caml values. Since those values have uniform representations (see chapter 9), a single C type suffices to encode all Objective Caml values. On page 323, we will present the facilities for encoding and decoding values, and illustrate them by a function that explores the representations of Objective Caml values.

The example in figure 12.1 respects the constraints mentioned above. The function `f_c`, associated with an Objective Caml function of type *int* -> *int* -> *int* -> *int*, is indeed a function with three parameters of type `value` returning a result of type `value`.

The Objective Caml bytecode interpreter evaluates calls to external functions differently, depending on the number of arguments[3]. If the number of arguments is less than or equal to five, the arguments are passed directly to the C function. If the number of arguments is greater than five, the C function's first parameter will get an array containing all of the arguments, and the C function's second parameter will get the number of arguments. These two cases must therefore be distinguished for external C functions that can be called from the bytecode interpreter. On the other hand, the Objective Caml native-code compiler always calls external functions by passing all the arguments directly, as function parameters.

External functions with more than five arguments

For external functions with more than five arguments, the programmer must provide two C functions: one for bytecode and the other for native-code. The syntax of external declarations allows the declaration of one Objective Caml function associated with two C functions:

Syntax : **`external`** *caml_name* **`:`** *type* **`=`** `"`*C_name_bytecode*`"` `"`*C_name_native*`"`

The function *C_name_bytecode* takes two parameters: an array of values of type `value` (*i.e.* a C pointer of type `value*`) and an integer giving the number of elements in this array.

Example

The following C program defines two functions for adding together six integers: `plus_native`, callable from native code, and `plus_bytecode`, callable from the bytecode compiler. The C code must include the file `mlvalues.h` containing the definitions of C types, Objective Caml values, and conversion macros.

```
#include <stdio.h>
#include <caml/mlvalues.h>

value plus_native (value x1,value x2,value x3,value x4,value x5,value x6)
{
  printf("<< NATIVE PLUS >>\n") ; fflush(stdout) ;
  return Val_long ( Long_val(x1) + Long_val(x2) + Long_val(x3)
                  + Long_val(x4) + Long_val(x5) + Long_val(x6)) ;
}

value plus_bytecode (value * tab_val, int num_val)
{
  int i;
  long res;
```

3. Recall that a function such as `fst`, of type *`'a * 'b -> 'a`*, does not have two arguments, but only one that happens to be a pair; on the other hand, a function of type *`int -> int -> int`* has two arguments.

```
  printf("<< BYTECODED PLUS >> : ") ; fflush(stdout) ;
  for (i=0,res=0;i<num_val;i++) res += Long_val(tab_val[i]) ;
  return Val_long(res) ;
}
```

The following Objective Caml program `exOCAML.ml` calls these two C functions.

```
external plus : int → int → int → int → int → int → int
             = "plus_bytecode" "plus_native" ;;
print_int (plus 1 2 3 4 5 6) ;;
print_newline () ;;
```

We now compile these programs with the two Objective Caml compilers and a C compiler that we call `cc`. We must give it the access path for the `mlvalues.h` include file.

```
$ cc -c -I/usr/local/lib/ocaml  exC.c

$ ocamlc -custom exC.o exOCAML.ml -o ex_byte_code.exe
$ ex_byte_code.exe
<< BYTECODED PLUS >> : 21

$ ocamlopt exC.o exOCAML.ml -o ex_native.exe
$ ex_native.exe
<< NATIVE PLUS >> : 21
```

Note

To avoid writing the C function twice (with the same body but different calling conventions), it suffices to implement the bytecode version as a call to the native-code version, as in the following sketch:

```
value prim_nat (value x1, ..., value xn) { ... }
value prim_bc (value *tbl, int n)
{ return prim_nat(tbl[0],tbl[1],...,tbl[n-1]) ; }
```

Linking with C

The linking phase creates an executable from C and Objective Caml files compiled with their respective compilers. The result of the native-code compiler is shown in figure 12.2.

The compilation of the C and Objective Caml sources generates machine code that is stored in the static allocation area of the program. The dynamic allocation area contains the execution stack (corresponding to the function calls in progress) and the heaps for C and Objective Caml.

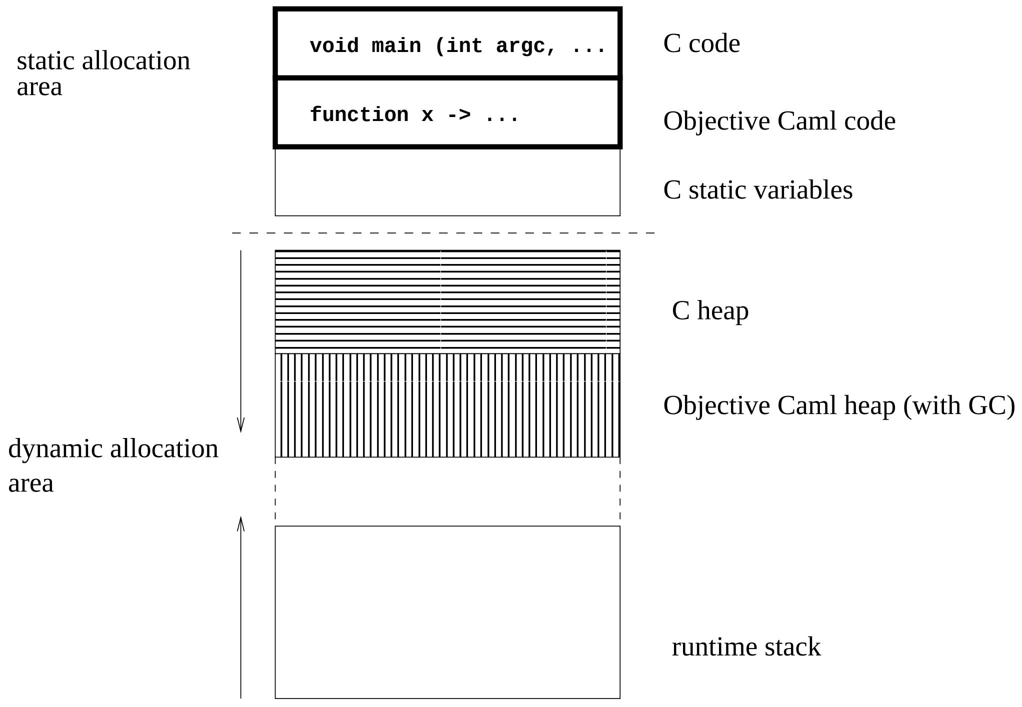


Figure 12.2: Mixed-language executable.

Run-time libraries

The C functions that can be called from a program using only the standard Objective Caml library are contained in the execution library of the abstract machine (see figure 7.3 page 200). For such a program, there is no need to provide additional libraries at link-time. However, when using Objective Caml libraries such as `Graphics`, `Num` or `Str`, the programmer must explicitly provide the corresponding C libraries at link-time. This is the purpose of the `-custom` compiler option (see see chapter 7, page 207). Similarly, when we wish to call our C functions from Objective Caml, we must provide the object file containing those C functions at link-time. The following example illustrates this.

The three linking modes

The linking commands differ slightly between the native-code compiler, the bytecode compiler, and the construction of toplevel interactive loops. The compiler options relevant to these linking modes are described in chapter 7.

To illustrate these linking modes, we consider again the example in figure 12.1. Assume the Objective Caml source file is named `progocaml.ml`. It uses the external function `f_c` defined in the C file `progC.c`. In turn, the function `f_c` refers to a C library `a_C_library.a`. Once all these files are compiled separately, we link them together using the following commands:

- bytecode:
  `ocamlc -custom -o vbc.exe progC.o a_C_library.a progocaml.cmo`
- native code:
  `ocamlopt progC.o -o vn.exe a_C_library.a progocaml.cmx`

We obtain two executable files: `vbc.exe` for the bytecode version, and `vn.exe` for the native-code version.

Building an enriched abstract machine

Another possibility is to augment the run-time library of the abstract machine with new C functions callable from Objective Caml. This is achieved by the following commands:

```
ocamlc -make-runtime -o new_ocamlrun progC.o a_C_library.a
```

We can then build a bytecode executable `vbcnam.exe` targeted to the new abstract machine:

```
ocamlc -o vbcnam.exe -use-runtime new_ocamlrun progocaml.cmo
```

To run this bytecode executable, either give it as the first argument to the new abstract machine, as in `new_ocaml vbcnam.exe` , or run it directly as `vbcnam.exe`

Note

Linking in `-custom` mode scans the object files (`.cmo`) to build a table of all external functions mentioned. The bytecode required to use them is generated and added to the bytecode corresponding to the Objective Caml code.

Building a toplevel interactive loop

To be able to use an external function in the toplevel interactive loop, we must first build a new toplevel interpreter containing the C code for the function, as well as an Objective Caml file containing its declaration.

We assume that we have compiled the file `progC.c` containing the function `f_c`. We then build the toplevel loop `ftop` as follows:

```
ocamlmktop -custom -o ftop progC.o a_C_library.a ex.ml
```

The file `ex.ml` contains the external declaration for the function `f`. The new toplevel interpreter `ftop` then knows this function and contains the corresponding C code, as found in `progC.o`.

Mixing input-output in C and in Objective Caml

The input-output functions in C and in Objective Caml do not share their file buffers. Consider the following C program:

```
#include <stdio.h>
#include <caml/mlvalues.h>
value hello_world (value v)
  { printf("Hello World !!");  fflush(stdout);  return v; }
```

Writes to standard output must be flushed explicitly (`fflush`) to guarantee that they will be printed in the intended order.

```
# external caml_hello_world : unit → unit = "hello_world"  ;;
external caml_hello_world : unit -> unit = "hello_world"
# print_string "<< " ;
  caml_hello_world () ;
  print_string " >>\n" ;
  flush stdout ;;
Hello World !!<<  >>
- : unit = ()
```

The outputs from C and from Objective Caml are not intermingled as expected, because each language buffers its outputs independently. To get the correct behavior, the Objective Caml part must be rewritten as follows:

```
# print_string "<< " ; flush stdout ;
  caml_hello_world () ;
  print_string " >>\n" ; flush stdout ;;
<< Hello World !! >>
- : unit = ()
```

By flushing the Objective Caml output buffer after each write, we ensure that the outputs from each language appear in the expected order.

Exploring Objective Caml values from C

The machine representation of Objective Caml values differs from that of C values, even for fundamental types such as integers. This is because the Objective Caml garbage collector needs to record additional information in values. Since Objective Caml values are represented uniformly, their representations all belong to the same C type, named (unsurprisingly) `value`.

When Objective Caml calls a C function, passing it one or several arguments, those arguments must be decoded before using them in the C function. Similarly, the result of this C function must be encoded before being returned to Objective Caml.

These conversions (decoding and encoding) are performed by a number of macros and C functions provided by the Objective Caml runtime system. These macros and functions are declared in the include files listed in figure 12.3. These include files are part of the Objective Caml installation, and can be found in the directory where Objective Caml libraries are installed[4]

`caml/mlvalues.h`	definition of the `value` type and basic value conversion macros.
`caml/alloc.h`	functions for allocating Objective Caml values.
`caml/memory.h`	macros for interfacing with the Objective Caml garbage collector.

Figure 12.3: Include files for the C interface.

Classification of Objective Caml representations

An Objective Caml representation, that is, a C datum of type `value`, is one of:

- an immediate value (represented as an integer);
- a pointer into the Objective Caml heap;
- a pointer pointing outside the Objective Caml heap.

The Objective Caml heap is the memory area that is managed by the Objective Caml garbage collector. C code can also allocate and manipulate data structures in its own memory space, and communicate pointers to these data structures to Objective Caml.

Figure 12.4 shows the macros for classifying representations and converting between C integers and their Objective Caml representation. Note that C offers several integer

`Is_long(v)`	is `v` an Objective Caml integer?
`Is_block(v)`	is `v` an Objective Caml pointer?
`Long_val(v)`	extract the integer contained in `v`, as a C "long"
`Int_val(v)`	extract the integer contained in `v`, as a C "int"
`Bool_val(v)`	extract the boolean contained in `v` (0 if `false`, non-zero if `true`)

Figure 12.4: Classification of representations and conversion of immediate values.

types of varying sizes (`short`, `int`, `long`, etc), while Objective Caml has only one integer type, `int`.

4. Under Unix, this directory is `/usr/local/lib/ocaml` by default, or sometimes `/usr/lib/ocaml`. Under Windows, the default location is `C: \OCAML\LIB`, or the value of the environment variable `CAMLLIB`, if set.

Accessing immediate values

All Objective Caml immediate values are represented as integers:

- integers are represented by their value;
- characters are represented by their ASCII code[5];
- constant constructors are represented by an integer corresponding to their position in the datatype declaration: the n^th constant constructor of a datatype is represented by the integer $n - 1$.

The following program defines a C function `inspect` that inspects the representation of its argument:

```
#include <stdio.h>
#include <caml/mlvalues.h>
value inspect (value v)
{
  if (Is_long(v))
    printf ("v is an integer (%ld) : %ld", (long) v, Long_val(v));
  else if (Is_block(v))
    printf ("v is a pointer");
  else
    printf ("v is neither an integer nor a pointer (???)");
  printf("   ");
  fflush(stdout) ;
  return v ;
}
```

The function `inspect` tests whether its argument is an Objective Caml integer. If so, it prints the integer twice, first viewed as a C long integer (without conversion), then converted by the `Long_val` macro, which extracts the actual integer represented in the argument.

On the following example, we see that the machine representation of integers in Objective Caml differs from that of C:

```
# external inspect : 'a -> 'a = "inspect" ;;
external inspect : 'a -> 'a = "inspect"
# inspect 123 ;;
v is an integer (247) : 123   - : int = 123
# inspect max_int;;
v is an integer (2147483647) : 1073741823   - : int = 1073741823
```

We can also inspect values of other predefined types, such as *char* and *bool*:

```
# inspect 'A' ;;
v is an integer (131) : 65   - : char = 'A'
# inspect true ;;
v is an integer (3) : 1   - : bool = true
```

5. More precisely, by their ISO Latin-1 code, which is an 8-bit character encoding extending ASCII with accented letters and signs for Western languages. Objective Caml does not yet handle wider internationalized character sets such as Unicode.

```
# inspect false ;;
v is an integer (1) : 0   - : bool = false
# inspect [] ;;
v is an integer (1) : 0   - : '_a list = []
```

Consider the Objective Caml type `foo` defined thus:

```
# type foo = C1 | C2 of int | C3 | C4 ;;
```

The `inspect` function shows that constant constructors and non-constant constructors of this type are represented differently:

```
# inspect C1 ;;
v is an integer (1) : 0   - : foo = C1
# inspect C4 ;;
v is an integer (5) : 2   - : foo = C4
# inspect (C2 1) ;;
v is a pointer   - : foo = C2 1
```

When the function `inspect` detects an immediate value, it prints first the "physical" representation of this value (*i.e.* the representation viewed as a word-sized C integer of C type `long`); then it prints the "logical" contents of this value (*i.e.* the Objective Caml integer it represents, as returned by the decoding macro `Long_val`). The examples above show that the "physical" and the "logical" contents differ. This difference is due to the tag bit[6] used by the garbage collector to distinguish immediate values from pointers (see chapter 9, page 253).

Representation of structured values

Non-immediate Objective Caml values are said to be structured values. Those values are allocated in the Objective Caml heap and represented as a pointer to the corresponding memory block. All memory blocks contain a header word indicating the kind of the block as well as its size expressed in machine words. Figure 12.5 shows the structure of a block for a 32-bit machine. The two "color" bits are used by the garbage collector for walking the memory graph (see chapter 9, page 254). The "tag" field, or "tag" for short, contains the kind of the block. The "size" field contains the size of the block, in words, excluding the header. The macros listed in figure 12.6 return the tag and size of a block. The tag of a memory block can take the values listed in figure 12.7. Depending on the block tag, different macros are used to access the contents of the blocks. These macros are described in figure 12.8. When the tag is less than `No_scan_tag`, the heap block is structured as an array of Objective Caml value representations. Each element of the array is called a "field" of the memory block. In accordance with C and Objective Caml conventions, the first field is at index 0, and the last field is at index `Wosize_val(v) - 1`.

6. Here, the tag bit is the least significant bit.

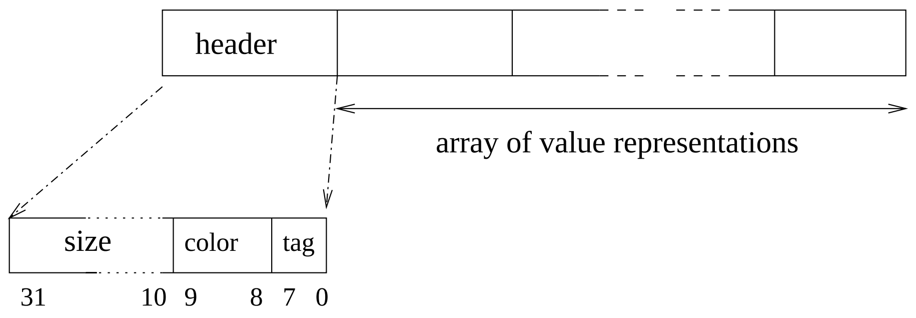


Figure 12.5: Structure of an Objective Caml heap block.

`Wosize_val(v)`	return the size of the block `v` (header excluded)
`Tag_val(v)`	return the tag of the block `v`

Figure 12.6: Accessing header information in memory blocks.

As we did earlier for immediate values, we now define a function to inspect memory blocks. The C function `print_block` takes an Objective Caml value representation, tests whether it is an immediate value or a memory block, and in the latter case prints the kind and contents of the block. It is called from the wrapper function `inspect_block`, which can be called from Objective Caml.

```
#include <stdio.h>
#include <caml/mlvalues.h>

void margin (int n)
  { while (n-- > 0) printf(".");  return; }

void print_block (value v,int m)
{
  int size, i;
```

from `0` to `No_scan_tag-1`	an array of Objective Caml value representations
`Closure_tag`	a function closure
`String_tag`	a character string
`Double_tag`	a double-precision float
`Double_array_tag`	an array of float
`Abstract_tag`	an abstract data type
`Final_tag`	an abstract data type equipped with a finalization function

Figure 12.7: Tags of memory blocks.

`Field(v,n)`	return the `n`th field of `v`.
`Code_val(v)`	return the code pointer for a closure.
`string_length(v)`	return the length of a string.
`Byte(v,n)`	return the `n` th character of a string, with C type `char`.
`Byte_u(v,n)`	same, but result has C type `unsigned char`.
`String_val(v)`	return the contents of a string with C type `(char *)`.
`Double_val(v)`	return the float contained in `v`.
`Double_field(v,n)`	return the `n` th float contained in the float array `v`.

Figure 12.8: Accessing the content of a memory block.

```
  margin(m);
  if (Is_long(v))
    { printf("immediate value (%d)\n", Long_val(v));  return; };
  printf ("memory block: size=%d  -  ", size=Wosize_val(v));
  switch (Tag_val(v))
   {
    case Closure_tag :
        printf("closure with %d free variables\n", size-1);
        margin(m+4); printf("code pointer: %p\n",Code_val(v)) ;
        for (i=1;i<size;i++)  print_block(Field(v,i), m+4);
        break;
    case String_tag :
        printf("string: %s (%s)\n", String_val(v),(char *) v);
        break;
    case Double_tag:
        printf("float: %g\n", Double_val(v));
        break;
    case Double_array_tag :
        printf ("float array: ");
        for (i=0;i<size/Double_wosize;i++)  printf("  %g", Double_field(v,i));
        printf("\n");
        break;
    case Abstract_tag : printf("abstract type\n"); break;
    case Final_tag : printf("abstract finalized type\n"); break;
    default:
        if (Tag_val(v)>=No_scan_tag) { printf("unknown tag"); break; };
        printf("structured block (tag=%d):\n",Tag_val(v));
        for (i=0;i<size;i++)  print_block(Field(v,i),m+4);
   }
  return ;
}

value inspect_block (value v)
```

```
  { print_block(v,4); fflush(stdout); return v; }
```

Each possible tag for a block corresponds to a case of the `switch` construct. In the case of a block containing an array of Objective Caml values, we recursively call `print_block` on each field of the array. We then redefine the `inspect` function:

```
# external inspect : 'a -> 'a = "inspect_block" ;;
external inspect : 'a -> 'a = "inspect_block"
```

We can now explore the representations of Objective Caml structured values. We must be careful not to apply `inspect_block` to a cyclic value, since the recursive traversal of the value would then loop indefinitely.

Arrays, tuples, and records

Arrays and tuples are represented by structured blocks. The n^{th} field of the block contains the representation of the n^{th} element of the array or tuple.

```
# inspect [| 1; 2; 3 |] ;;
....memory block: size=3  -  structured block (tag=0):
.......immediate value (1)
.......immediate value (2)
.......immediate value (3)
- : int array = [|1; 2; 3|]
# inspect ( 10 , true , () ) ;;
....memory block: size=3  -  structured block (tag=0):
.......immediate value (10)
.......immediate value (1)
.......immediate value (0)
- : int * bool * unit = 10, true, ()
```

Records are also represented as structured blocks. The values of the record fields appear in the order given at record declaration time. Mutable fields and immutable fields are represented identically.

```
# type foo = { fld1: int ; mutable fld2: int } ;;
type foo = { fld1: int; mutable fld2: int }
# inspect { fld1=10 ; fld2=20 } ;;
....memory block: size=2  -  structured block (tag=0):
.......immediate value (10)
.......immediate value (20)
- : foo = {fld1=10; fld2=20}
```

Warning Nothing prevents a C function from physically modifying an immutable record field. It is the programmers' responsibility to make sure that their C functions do not introduce inconsistencies in Objective Caml data structures.

Sum types

We previously saw that constant constructors are represented like integers. A non-constant constructor is represented by a block containing the constructor's arguments, with a tag identifying the constructor. The tag associated with a non-constant constructor represents its position in the type declaration: the first non-constant constructor has tag 0, the second one has tag 1, and so on.

```
# type foo = C1 of int * int * int | C2 of int | C3 | C4 of int * int ;;
type foo = | C1 of int * int * int | C2 of int | C3 | C4 of int * int
# inspect (C1 (1,2,3)) ;;
....memory block: size=3  -  structured block (tag=0):
.......immediate value (1)
.......immediate value (2)
.......immediate value (3)
- : foo = C1 (1, 2, 3)
# inspect (C4 (1,2)) ;;
....memory block: size=2  -  structured block (tag=2):
.......immediate value (1)
.......immediate value (2)
- : foo = C4 (1, 2)
```

Note

The type `list` is a sum type whose declaration is:
type *'a list* = [] | :: **of** *'a * 'a list*. This type has only one non-constant constructor (::). Thus, a non-empty list is represented by a memory block with tag 0.

Character strings

Characters inside strings occupy one byte each. Thus, the memory block representing a string uses one word per group of four characters (on a 32-bit machine) or eight characters (on a 64-bit machine).

Warning Objective Caml strings can contain the null character whose ASCII code is 0. In C, the null character represents the end of a string, and cannot appear inside a string.

```
#include <stdio.h>
#include <caml/mlvalues.h>

value explore_string (value v)
{
  char *s;
  int i,size;
  s = (char *) v;
  size = Wosize_val(v) * sizeof(value);
  for (i=0;i<size;i++)
    {
      int p = (unsigned int) s[i] ;
      if ((p>31) && (p<128)) printf("%c",s[i]); else printf("(#%u)",p);
    }
  printf("\n");
  fflush(stdout);
  return v;
}
```

The length and position of last character of an Objective Caml string are determined not by looking for a terminating null character, as in C, but by combining the size of the memory block that contains the string with the last byte of the last word of this block, which indicates the number of *unused* bytes in the last word. The following examples clarify the role played by this last byte.

```
# external explore : string → string = "explore_string" ;;
external explore : string -> string = "explore_string"
# ignore(explore "");
  ignore(explore "a");
  ignore(explore "ab");
  ignore(explore "abc");
  ignore(explore "abcd");
  ignore(explore "abcd\000") ;;
(#0)(#0)(#0)(#3)
a(#0)(#0)(#2)
ab(#0)(#1)
abc(#0)
abcd(#0)(#0)(#0)(#3)
abcd(#0)(#0)(#0)(#2)
- : unit = ()
```

In the last two examples ("abcd" and "abcd\000"), the strings are of length 4 and 5 respectively. This explains why the last byte takes two different values, although the other bytes of the string representations are identical.

Floats and float arrays

Objective Caml offers only one type (*float*) of floating-point numbers. This type corresponds to 64-bit, double-precision floating point numbers in C (type `double`). Values of type *float* are heap-allocated and represented by a memory block of size 2 words (on a 32-bit machine) or 1 word (on a 64-bit machine).

```
# inspect 1.5 ;;
....memory block: size=2  -  float: 1.5
- : float = 1.5
# inspect 0.0;;
....memory block: size=2  -  float: 0
- : float = 0
```

Arrays of floats are represented specially to reduce their memory occupancy: the floats contained in the array are stored consecutively in the memory block, rather than having each float heap-allocated separately. Therefore, float arrays possess a specific tag and specific access macros.

```
# inspect [| 1.5 ; 2.5 ; 3.5 |] ;;
....memory block: size=6  -  float array:   1.5  2.5  3.5
- : float array = [|1.5; 2.5; 3.5|]
```

This optimized representation encourages the use of Objective Caml for numerical computations that manipulate many float arrays: operations on array elements are much more efficient than if each float was heap-allocated separately.

Warning When allocating an Objective Caml float array from C, the size of the block should be the number of array elements multiplied by `Double_wosize`. The `Double_wosize` macro represents the number of words occupied by a double-precision float (2 words on a 32-bit machine, but only 1 word on a 64-bit machine).

With the exception of float arrays, floating-point numbers contained in other data structures are always treated as a structured, heap-allocated value. The following example shows the representation of a list of floats.

```
# inspect [ 3.14; 1.2; 7.6];;
....memory block: size=2  -  structured block (tag=0):
........memory block: size=2  -  float: 3.14
........memory block: size=2  -  structured block (tag=0):
............memory block: size=2  -  float: 1.2
............memory block: size=2  -  structured block (tag=0):
................memory block: size=2  -  float: 7.6
................immediate value (0)
- : float list = [3.14; 1.2; 7.6]
```

The list is viewed as a block with size 2, containing its head and its tail. The head of the list is a float, which is also a block of size 2.

Closures

A function value is represented by the code to be executed when the function is applied, and by its environment (see chapter 2, page 23). There are two ways to build a function value: either by explicit abstraction (as in `fun x -> x+1`) or by partial application of a curried function (as in `(fun x -> fun y -> x+y) 1`).

The environment of a closure can contain three kinds of variables: those declared globally, those declared locally, and the function parameters already instantiated by a partial application. The implementation treats those three kinds differently. Global variables are stored in a global environment that is not explicitly part of any closure. Local variables and instantiated parameters can appear in closures, as we now illustrate.

A closure with an empty environment is simply a memory block containing a pointer to the code of the function:

```
# let f = fun x y z → x+y+z ;;
val f : int -> int -> int -> int = <fun>
# inspect f ;;
....memory block: size=1  -  closure with 0 free variables
.......code pointer: 0x808c9d4
- : int -> int -> int -> int = <fun>
```

Functions with free local variables are represented by closures with non-empty environments. Here, the closure contains both a pointer to the code of the function, and the values of its free local variables.

```
# let g = let x = 1 and y = 2 in fun z → x+y+z ;;
val g : int -> int = <fun>
# inspect g ;;
....memory block: size=3  -  closure with 2 free variables
.......code pointer: 0x808ca38
.......immediate value (1)
.......immediate value (2)
- : int -> int = <fun>
```

The Objective Caml virtual machine treats partial applications of functions specially for better performance. A partial application of an abstraction is represented by a closure containing a value for each of the instantiated parameters, plus a pointer to the closure for the initial abstraction.

```
# let a1 = f 1 ;;
val a1 : int -> int -> int = <fun>
# inspect (a1) ;;
....memory block: size=3  -  closure with 2 free variables
.......code pointer: 0x808c9d0
.......memory block: size=1  -  closure with 0 free variables
...........code pointer: 0x808c9d4
.......immediate value (1)
- : int -> int -> int = <fun>
# let a2 = a1 2 ;;
val a2 : int -> int = <fun>
# inspect (a2) ;;
....memory block: size=4  -  closure with 3 free variables
.......code pointer: 0x808c9d0
.......memory block: size=1  -  closure with 0 free variables
...........code pointer: 0x808c9d4
.......immediate value (1)
.......immediate value (2)
- : int -> int = <fun>
```

Figure 12.9 depicts the result of the inspection above.

The function `f` has no free variables, hence the environment part of its closure is empty. The code pointer for a function with several arguments points to the code that should be

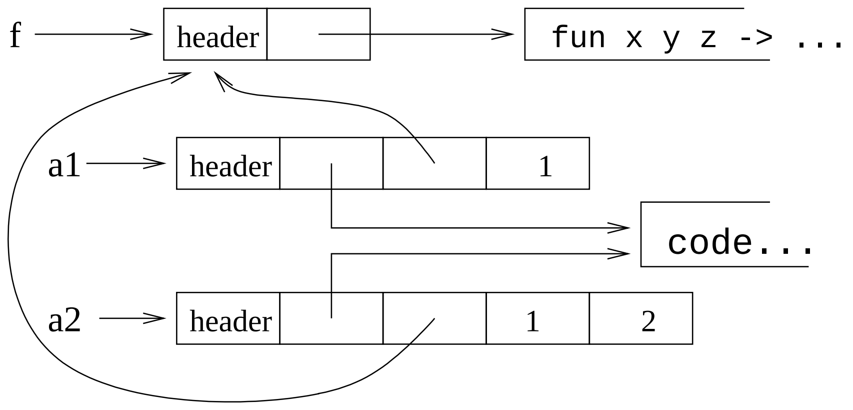


Figure 12.9: Closure representation.

called when all arguments are provided. In the case of `f`, this is the code corresponding to `x+y+z`. Partial applications of this function result in intermediate closures that point to a shared code (it is the same code pointer for `a1` and `a2`). The role of this code is to accumulate the arguments and detect when all arguments have been provided. If so, it pushes all arguments and calls the actual code for the function body; if not, it creates a new closure. For instance, the application of `a1` to 2 fails to provide all arguments to the function `f` (the last argument is still missing), hence a closure is created containing the first two arguments, 1 and 2. Notice that the closures resulting from partial applications always contain, in the first environment slot, a pointer to the original closure. The original closure will be called when all arguments have been gathered.

Mixing local declarations and partial applications results in the following representation:

```
# let g x = let y=2 in fun z → x+y+z ;;
val g : int -> int -> int = <fun>
# let a1 = g 1 ;;
val a1 : int -> int = <fun>
# inspect a1 ;;
....memory block: size=3  -  closure with 2 free variables
........code pointer: 0x808ca78
........immediate value (1)
........immediate value (2)
- : int -> int = <fun>
```

Abstract types

Values of an abstract type are represented like those of its implementation type. Actually, type information is used only during type-checking and compilation. During execution, the types are not needed – only the memory representation (tag bits on values, size and tag fields on memory blocks) needs to be communicated to the garbage collector.

For instance, a value of the abstract type *'a Stack.t* is represented as a reference to a list, since the type *'a Stack.t* is implemented as *'a list ref*.

```
# let p = Stack.create();;
val p : '_a Stack.t = <abstr>
# Stack.push 3 p;;
- : unit = ()
# inspect p;;
....memory block: size=1  -  structured block (tag=0):
.......memory block: size=2  -  structured block (tag=0):
...........immediate value (3)
...........immediate value (0)
- : int Stack.t = <abstr>
```

On the other hand, some abstract types are implemented by representations that cannot be expressed in Objective Caml. Typical examples include arrays of weak pointers and input-output channels. Often, values of those abstract types are represented as memory blocks with tag `Abstract_tag`.

```
# let w = Weak.create 10;;
val w : '_a Weak.t = <abstr>
# Weak.set w 0 (Some p);;
- : unit = ()
# inspect w;;
....memory block: size=11  -  abstract type
- : int Stack.t Weak.t = <abstr>
```

Sometimes, a finalization function is attached to those values. Finalization functions are C functions which are called by the garbage collector just before the value is collected. They are very useful to free external resources, such as an input-output buffer, just before the memory block referring to those resources disappears. For instance, inspection of the "standard output" channel reveals that the type *out_channel* is represented by abstract memory blocks with a finalization function:

```
# inspect (stdout) ;;
....memory block: size=2  -  abstract finalized type
- : out_channel = <abstr>
```

Creating and modifying Objective Caml values from C

A C function called from Objective Caml can modify its arguments in place, or return a newly-created value. This value must match the Objective Caml type for the function result. For base types, several C macros are provided to convert a C datum to an Objective Caml value. For structured types, the new value must be allocated in the

Objective Caml heap, with the correct size, and its fields initialized with values of the correct types. Considerable care is required here: it is easy to construct bad values from C, and these bad values may crash the Objective Caml program.

Any allocation in the Objective Caml heap can trigger a garbage collection, which will deallocate unused memory blocks and may move live blocks. Therefore, any Objective Caml value manipulated from C must be registered with the Objective Caml garbage collector, if they are to survive the allocation of a new block. These values must be treated as extra memory roots by the garbage collector. To this end, several macros are provided for registering extra roots with the garbage collector.

Finally, C code can allocate Objective Caml heap blocks that contain C data instead of Objective Caml values. This C data will then benefit from Objective Caml's automatic memory management. If the C data requires explicit deallocation, a finalization function can be attached to the heap block.

Modifying Objective Caml values

The following macros allow the creation of immediate Objective Caml values from the corresponding C data, and the modification of structured values in place.

`Val_long(l)`	return the value representing the long integer `l`
`Val_int(i)`	return the value representing the integer `l`
`Val_bool(x)`	return `false` if `x=0`, `true` otherwise
`Val_true`	the representation of `true`
`Val_false`	the representation of `false`
`Val_unit`	the representation of `()`
`Store_field(b,n,v)`	store the value `v` in the `n`-th field of block `b`
`Store_double_field(b,n,d)`	store the float `d` in the `n`-th field of the float array `b`

Figure 12.10: Creation of immediate values and modification of structured blocks.

Moreover, the macros `Byte` and `Byte_u` can be used on the left-hand side of an assignment to modify the characters of a string. The `Field` macro can also be used for assignment on blocks with tag `Abstract_tag` or `Final_tag`; use `Store_field` for blocks with tag between `0` and `No_scan_tag-1`. The following function reverses a character string in place:

```
#include <caml/mlvalues.h>
value swap_char(value v, int i, int j)
  { char c=Byte(v,i); Byte(v,i)=Byte(v,j); Byte(v,j)=c; }
value swap_string (value v)
{
  int i,j,t = string_length(v) ;
  for (i=0,j=t-1; i<t/2; i++,j--)  swap_char(v,i,j) ;
  return v ;
}
```

```
# external mirror : string → string = "swap_string" ;;
external mirror : string -> string = "swap_string"
# mirror "abcdefg" ;;
- : string = "gfedcba"
```

Allocating new blocks

The functions listed in figure 12.11 allocate new blocks in the Objective Caml heap. The

`alloc(n, t)`	return a new block of size `n` words and tag `t`
`alloc_tuple(n)`	same, with tag 0
`alloc_string(n)`	return an uninitialized string of length `n` characters
`copy_string(s)`	return a string initialized with the C string `s`
`copy_double(d)`	return a block containing the double float `d`
`alloc_array(f, a)`	return a block representing an array, initialized by applying the conversion function `f` to each element of the C array of pointers `a`, null-terminated.
`copy_string_array(p)`	return a block representing an array of strings, obtained from the C string array `p` (of type `char **`), null-terminated.

Figure 12.11: Functions for allocating blocks.

function `alloc_array` takes an array of pointers `a`, terminated by a null pointer, and a conversion function `f` taking a pointer and returning a value. The result of `alloc_array` is an Objective Caml array containing the results of applying `f` in turn to each pointer in `a`. In the following example, the function `make_str_array` uses `alloc_array` to convert a C array of strings.

```
#include <caml/mlvalues.h>
value make_str (char *s) { return copy_string(s); }
value make_str_array (char **p) { return alloc_array(make_str,p) ; }
```

It is sometimes necessary to allocate blocks of size 0, for instance to represent an empty Objective Caml array. Such a block is called an *atom*.

```
# inspect [| |] ;;
....memory block: size=0  -  structured block (tag=0):
- : '_a array = [||]
```

Because atoms are allocated statically and do not reside in the dynamic part of the Objective Caml heap, the allocation functions in figure 12.11 must not be used to allocate atoms. Instead, atoms are created in C by the macro `Atom(t)`, where `t` is the desired tag for the block of size 0.

Storing C data in the Objective Caml heap

It is sometimes convenient to use the Objective Caml heap to store arbitrary C data that does not respect the constraints imposed by the garbage collector. In this case, blocks with tag `Abstract_tag` must be used.

A natural example is the manipulation of native C integers (of size 32 or 64 bits) in Objective Caml. Since these integers are not tagged as the Objective Caml garbage collector expects, they must be kept in one-word heap blocks with tag `Abstract_tag`.

```
#include <caml/mlvalues.h>
#include <stdio.h>

value Cint_of_OCAMLint (value v)
{
  value res = alloc(1,Abstract_tag) ;
  Field(res,0) = Long_val(v) ;
  return res ;
}

value OCAMLint_of_Cint (value v)  { return Val_long(Field(v,0)) ; }

value Cplus (value v1,value v2)
{
  value res = alloc(1,Abstract_tag) ;
  Field(res,0) = Field(v1,0) + Field(v2,0) ;
  return res ;
}

value printCint (value v)
{
  printf ("%d",(long) Field(v,0)) ; fflush(stdout) ;
  return Val_unit ;
}
```

```
# type cint
  external cint_of_int : int → cint = "Cint_of_OCAMLint"
  external int_of_cint : cint → int = "OCAMLint_of_Cint"
  external plus_cint : cint → cint → cint = "Cplus"
  external print_cint : cint → unit = "printCint" ;;
```

We can now work on native C integers, without losing the use of the tag bit, while remaining compatible with Objective Caml's garbage collector. However, such integers are heap-allocated, instead of being immediate values, which renders arithmetic operations less efficient.

```
# let a = 1000000000 ;;
val a : int = 1000000000
# a+a ;;
- : int = -147483648
# let c = let b = cint_of_int a in plus_cint b b ;;
val c : cint = <abstr>
# print_cint c ; print_newline () ;;
2000000000
- : unit = ()
# int_of_cint c ;;
- : int = -147483648
```

Finalization functions

Abstract blocks can also contain pointers to memory blocks allocated outside the Objective Caml heap. We know that Objective Caml blocks that are no longer used by the program are deallocated by the garbage collector. But what happens to a block allocated in the C heap and referenced by an abstract block that was reclaimed by the GC? To avoid memory leaks, we can associate a *finalization function* to the abstract block; this function is called by the GC before reclaiming the abstract block.

An abstract block with an attached finalization function is allocated via the function `alloc_final (n, f, used, max)` .

- `n` is the size of the block, in words. The first word of the block is used to store the finalization function; hence the size occupied by the user data must be increased by one word.
- `f` is the finalization function itself, with type `void f (value)`. It receives the abstract block as argument, just before this block is reclaimed by the GC.
- `used` represents the memory space (outside the Objective Caml heap) occupied by the C data. `used` must be ¡= `max`.
- `max` is the maximum memory space outside the Objective Caml heap that we tolerate not being reclaimed immediately.

For efficiency reasons, the Objective Caml garbage collector does not reclaim heap blocks as soon as they become unused, but some time later. The ratio `used/max` controls the proportion of finalized abstract blocks that the garbage collector may leave allocated while they are no longer used. A ratio of 0 (that is, `used = 0`) lets the garbage collector work at its usual pace; higher ratios (no greater than 1) cause it to work harder and spend more CPU time finding unused finalized blocks and reclaiming them.

The following program manipulates arrays of C integers allocated in the C heap via `malloc`. To allow the Objective Caml garbage collector to reclaim these arrays automatically, the `create` function wraps them in a finalized abstract block, containing both a pointer to the array and the finalization function `finalize_it`.

```
#include <malloc.h>
#include <stdio.h>
#include <caml/mlvalues.h>

typedef struct {
        int size ;
        long * tab ; } IntTab ;

IntTab *alloc_it (int s)
{
  IntTab *res = malloc(sizeof(IntTab)) ;
  res->size = s ;
  res->tab = (long *) malloc(sizeof(long)*s) ;
  return res ;
}
void free_it (IntTab *p)  { free(p->tab) ; free(p) ; }
void put_it (int n,long q,IntTab *p)  { p->tab[n] = q ; }
long get_it (int n,IntTab *p)  { return p->tab[n]; }

void finalize_it (value v)
{
  IntTab *p = (IntTab *) Field(v,1) ;
  int i;
  printf("reclamation of an IntTab by finalization [") ;
  for (i=0;i<p->size;i++) printf("%d ",p->tab[i]) ;
  printf("]\n"); fflush(stdout) ;
  free_it ((IntTab *) Field(v,1)) ;
}
value create (value s)
{
  value block ;
  block = alloc_final (2, finalize_it,Int_val(s)*sizeof(IntTab),100000) ;
  Field(block,1) = (value) alloc_it(Int_val(s)) ;
  return block ;
}
value put (value n,value q,value t)
{
  put_it (Int_val(n), Long_val(q), (IntTab *) Field(t,1)) ;
  return Val_unit ;
}
value get (value n,value t)
{
  long res = get_it (Int_val(n), (IntTab *) Field(t,1)) ;
  return Val_long(res) ;
}
```

The C functions visible from Objective Caml are: `create`, `put` and `get`.

```
# type c_int_array
  external cia_create : int → c_int_array = "create"
  external cia_get : int → c_int_array → int = "get"
  external cia_put : int→ int → c_int_array → unit = "put" ;;
```

We can now manipulate our new data structure from Objective Caml:

```
# let tbl = cia_create 10 and tbl2 = cia_create 10
  in for i=0 to 9 do cia_put i (i*2) tbl done ;
     for i=0 to 9 do print_int (cia_get i tbl) ; print_string " " done ;
     print_newline () ;
     for i=0 to 9 do cia_put (9-i) (cia_get i tbl) tbl2 done ;
     for i=0 to 9 do print_int (cia_get i tbl2) ; print_string " " done ;;
0 2 4 6 8 10 12 14 16 18
18 16 14 12 10 8 6 4 2 0 - : unit = ()
```

We now force a garbage collection to check that the finalization function is called:

```
# Gc.full_major () ;;
reclaimation of an IntTab by finalization [18 16 14 12 10 8 6 4 2 0 ]
reclaimation of an IntTab by finalization [0 2 4 6 8 10 12 14 16 18 ]
- : unit = ()
```

In addition to freeing C heap blocks, finalization functions can also be used to close files, terminate processes, etc.

Garbage collection and C parameters and local variables

A C function can trigger a garbage collection, either during an allocation (if the heap is full), or voluntarily by calling `void Garbage_collection_function ()`.

Consider the following example. Can you spot the error?

```
#include <caml/mlvalues.h>
#include <caml/memory.h>

value identity (value x)
{
  Garbage_collection_function() ;
  return x;
}
```

```
# external id : 'a → 'a = "identity" ;;
external id : 'a -> 'a = "identity"
# id [1;2;3;4;5] ;;
- : int list = [538918066; 538918060; 538918054; 538918048; 538918042]
```

The list passed as parameter to `id`, hence to the C function `identity`, can be moved or reclaimed by the garbage collector. In the example, we forced a garbage collection, but any allocation in the Objective Caml heap could have triggered a garbage collection as well. The anonymous list passed to `id` was reclaimed by the garbage collector, because it is not reachable from the set of known roots. To avoid this, any C function that allocates anything in the Objective Caml heap must tell the garbage collector about the C function's parameters and local variables of type `value`. This is achieved by using the macros described next.

For parameters, these macros are used within the body of the C function as if they were additional declarations:

`CAMLparam1(v)`	:	for one parameter `v` of type `value`
`CAMLparam2(v1,v2)`	:	for two parameters
...		...
`CAMLparam5(v1,...,v5)`	:	for five parameters
`CAMLparam0` ;	:	required when there are no `value` parameters.

If the C function has more than five `value` parameters, the first five are declared with the `CAMLparam5` macro, and the remaining parameters with the macros `CAMLxparam1`, ..., `CAMLxparam5`, used as many times as necessary to list all `value` parameters.

`CAMLparam5(v1,...,v5);`		
`CAMLxparam5(v6,...,v10);`		
`CAMLxparam2(v11,v12);`	:	for 12 parameters of type `value`

For local variables, these macros are used instead of normal C declarations of the variables. Local variables of type `value` must also be registered with the garbage collector, using the macros `CAMLlocal1`, ..., `CAMLlocal5`. An array of values is declared with `CAMLlocalN(tbl,n)` where `n` is the number of elements of the array `tbl`. Finally, to return from the C function, we must use the macro `CAMLreturn` instead of C's `return` construct.

Here is the corrected version of the previous example:

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
value identity2 (value x)
{
  CAMLparam1(x) ;
  Garbage_collection_function() ;
  CAMLreturn x;
}
```

```
# external id : 'a → 'a = "identity2" ;;
external id : 'a -> 'a = "identity2"
# let a = id [1;2;3;4;5] ;;
val a : int list = [1; 2; 3; 4; 5]
```

We now obtain the expected result.

Calling an Objective Caml closure from C

To apply a closure (*i.e.* an Objective Caml function value) to one or several arguments from C, we can use the functions declared in the header file `callback.h`.

`callback(f,v)`	:	apply the closure `f` to the argument `v`,
`callback2(f,v1,v2)`	:	same, to two arguments,
`callback3(f,v1,v2,v3)`	:	same, to three arguments,
`callbackN(f,n,tbl)`	:	same, to `n` arguments stored in the array `tbl`.

All these functions return a `value`, which is the result of the application.

Registering Objective Caml functions with C

The `callback` functions require the Objective Caml function to be applied as a closure, that is, as a value that was passed as an argument to the C function. We can also register a closure from Objective Caml, giving it a name, then later refer to the closure by its name in a C function.

The function `register` from module `Callback` associates a name (of type `string`) with a closure or with any other Objective Caml value (of any type, that is, *'a*). This closure or value can be recovered from C using the C function `caml_named_value`, which takes a character string as argument and returns a pointer to the closure or value associated with that name, if it exists, or the null pointer otherwise.

An example is in order:

```
# let plus x y = x + y ;;
val plus : int -> int -> int = <fun>
# Callback.register "plus3_ocaml" (plus 3);;
- : unit = ()
```

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/callback.h>

value plus3_C (value v)
{
  CAMLparam1(v);
  CAMLlocal1(f);
  f = *(caml_named_value("plus3_ocaml")) ;
  CAMLreturn callback(f,v) ;
}
```

```
# external plusC : int → int = "plus3_C" ;;
```

```
external plusC : int -> int = "plus3_C"
# plusC 1 ;;
- : int = 4
# Callback.register "plus3_ocaml" (plus 5);;
- : unit = ()
# plusC 1 ;;
- : int = 6
```

Do not confuse the declaration of a C function with **external** and the registration of an Objective Caml closure with the function register. In the former case, the declaration is static, the correspondence between the two names is established at link time. In the latter case, the binding is dynamic: the correspondence between the name and the closure is performed at run time. In particular, the name–closure binding can be modified dynamically by registering a different closure with the same name, thus modifying the behavior of C functions using that name.

Exception handling in C and in Objective Caml

Different languages have different mechanisms for raising and handling exceptions: C relies on setjmp and longjmp, while Objective Caml has built-in constructs for exceptions (**try ... with**, **raise**). Of course, these mechanisms are not compatible: they do not keep the same information when setting up a handler. It is extremely hard to safely implement the nesting of exception handlers of different kinds, while ensuring that an exception correctly "jumps over" handlers. For this reason, only Objective Caml exceptions can be raised and handled from C; setjmp and longjmp in C cannot be caught from Objective Caml, and must not be used to skip over Objective Caml code.

All functions and macros introduced in this section are defined in the header file fail.h.

Raising a predefined exception

From a C function, it is easy to raise one of the exceptions Failure, Invalid_argument or Not_found from the Pervasives module: just use the following functions.

failwith(s)	:	raise the exception Failure(s)
invalid_argument(s)	:	raise the exception Invalid_argument(s)
raise_not_found()	:	raise the exception Not_found

In the first two cases, s is a C string (char *) that ends up as the argument to the exception raised.

Raising a user-defined exception

A registration mechanism similar to that for closures enables user-defined exceptions to be raised from C. We must first register the exception using the `Callback` module's `register_exception` function. Then, from C, we retrieve the exception identifier using the `caml_named_value` function (see page 343). Finally, we raise the exception, using one of the following functions:

`raise_constant(e)`	raise the exception `e` with no argument,
`raise_with_arg(e,v)`	raise the exception `e` with the value `v` as argument,
`raise_with_string(e,s)`	same, but the argument is taken from the C string `s`.

Here is an example C function that raises an Objective Caml exception:

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/fail.h>

value divide (value v1,value v2)
{
  CAMLparam2(v1,v2);
  if (Long_val(v2) == 0)
    raise_with_arg(*caml_named_value("divzero"),v1) ;
  CAMLreturn Val_long(Long_val(v1)/Long_val(v2)) ;
}
```

And here is an Objective Caml transcript showing the use of that C function:

```
# external divide : int → int → int = "divide" ;;
external divide : int -> int -> int = "divide"
# exception Division_zero of int ;;
exception Division_zero of int
# Callback.register_exception "divzero" (Division_zero 0) ;;
- : unit = ()
# divide 20 4 ;;
- : int = 5
# divide 22 0 ;;
Uncaught exception: Division_zero(22)
```

Catching an exception

In a C function, we cannot catch an exception raised from another C function. However, we can catch Objective Caml exceptions arising from the application of an Objective Caml function (callback). This is achieved via the functions `callback_exn`, callback2_exn, callback3_exn and callbackN_exn, which are similar to the standard callback functions, except that if the callback raises an exception, this exception is caught and returned as the result of the callback. The result value of the callback_exn functions must be tested with Is_exception_result(v); this predicate returns "true" if the result value represents an uncaught exception, and "false" otherwise. The macro Extract_exception(v) returns the exception value contained in an exceptional result value.

The C function divide_print below calls the Objective Caml function divide using callback2_exn, and checks whether the result is an exception. If so, it prints a message and raises the exception again; otherwise it prints the result.

```
#include <stdio.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/callback.h>
#include <caml/fail.h>

value divide_print (value v1,value v2)
{
  CAMLparam2(v1,v2) ;
  CAMLlocal3(div,dbz,res) ;
  div = * caml_named_value("divide") ;
  dbz = * caml_named_value("div_by_0") ;
  res = callback2_exn (div,v1,v2) ;
  if (Is_exception_result(res))
    {
      value exn=Extract_exception(res);
      if (Field(exn,0)==dbz)  printf("division by 0\n") ;
      else printf("other exception\n");
      fflush(stdout);
      if (Wosize_val(exn)==1)  raise_constant(Field(exn,0)) ;
      else raise_with_arg(Field(exn,0),Field(exn,1)) ;
    }
  printf("result = %d\n",Long_val(res)) ;
  fflush(stdout) ;
  CAMLreturn Val_unit ;
}
```

```
# Callback.register "divide" (/) ;;
- : unit = ()
# Callback.register_exception "div_by_0" Division_by_zero ;;
- : unit = ()
# external divide_print : int → int → unit = "divide_print" ;;
external divide_print : int -> int -> unit = "divide_print"
# divide_print 42 3 ;;
result = 14
- : unit = ()
```

```
# divide_print 21 0 ;;
division by 0
Uncaught exception: Division_by_zero
```

As the examples above show, it is possible to raise an exception from C and catch it in Objective Caml, and also to raise an exception from Objective Caml and catch it in C. However, a C program cannot by itself raise and catch an Objective Caml exception.

Main program in C

Until now, the entry point of our programs was in Objective Caml; the program could then call C functions. Nothing prevents us from writing the entry point in C, and having the C code call Objective Caml functions when desired. To do this, the program must define the usual C `main` function. This function will then initialize the Objective Caml runtime system by calling the function `caml_main(char **)`, which takes as an argument the array of command-line arguments that corresponds to the `Sys.argv` array in Objective Caml. Control is then passed to the Objective Caml code using callbacks (see page 343).

Linking Objective Caml code with C

The Objective Caml compiler can output C object files (with extension `.o`) instead of Objective Caml object files (with extension `.cmo` or `.cmx`). All we need to do is set the `-output-obj` compiler flag.

```
ocamlc -output-obj files.ml
ocamlopt -output-obj.cmxa files.ml
```

From the Objective Caml source files, an object file with default name `camlprog.o` is produced.

The final executable is obtained by linking, using the C compiler, and adding the library `-lcamlrun` if the Objective Caml code was compiled to bytecode, or the library `-lasmrun` if it was compiled to native code.

```
cc camlprog.o filesC.o -lcamlrun
cc camlprog.o filesC.o -lasmrun
```

Calling Objective Caml functions from the C program is performed as described previously, via the `callback` functions. The only difference is that the initialization of the Objective Caml runtime system is performed via the function `caml_startup` instead of `caml_main`.

Exercises

Polymorphic Printing Function

We wish to define a printing function `print` with type *'a -> unit* able to print any Objective Caml value. To this end, we extend and improve the `inspect` function.

1. In C, write the function `print_ws` which prints Objective Caml as follows:
   - immediate values: as C integers;
   - strings: between quotes;
   - floats: as usual;
   - arrays of floats: between `[| |]`
   - closures: as `< code, env >`
   - everything else: as a tuple, between `( )`

   The function should handle structured types recursively.
2. To avoid looping on circular values, and to display sharing properly, modify this function to keep track of the addresses of heap blocks it has already seen. If an address appears several times, name it when it is first printed (`v = name`), and just print the name when this address is encountered again.
   (a) Define a data structure to record the addresses, determine when they occur several times, and associate a name with each address.
   (b) Traverse the value once first to determine all the addresses it contains and record them in the data structure.
   (c) The second traversal prints the value while naming addresses at their first occurrences.
   (d) Define the function `print` combining both traversals.

Matrix Product

1. Define an abstract type *float_matrix* for matrices of floating-point numbers.
2. Define a C type for these matrices.
3. Write a C function to convert values of type *float array array* to values of type *float_matrix*.
4. Write a C function performing the reverse conversion.
5. Add the C functions computing the sum and the product of these matrices.
6. Interface them with Objective Caml and use them.

Counting Words: Main Program in C

The Unix command `wc` counts the number of characters, words and lines in a file. The goal of this exercise is to implement this command, while counting repeated words only once.

1. Write the program `wc` in C. This program will simply count words, lines and characters in the file whose name is passed on the command line.
2. Write in Objective Caml a function `add_word` that uses a hash table to record how many times the function was invoked with the same character string as argument.
3. Write two functions `num_repeated_words` and `num_unique_words` counting respectively the number of word repetitions and the number of unique words, as determined from the hash table built by `add_word`.
4. Register the three previous functions so that they can be called from a C program.
5. Rewrite the main function of the `wc` program so that it prints the number of unique words instead of the number of words.
6. Write the `main` function and the commands required to compile this program as an Objective Caml program.
7. Write the `main` function and the commands required to compile this program as a C program.

Summary

This chapter introduced the interface between the Objective Caml language and the C language. This interface allows C functions to operate on Objective Caml values. Using abstract Objective Caml types, the converse is also possible. An important feature of this interface is the ability to use the Objective Caml garbage collector to perform automatic reclamation of values created in C. This interface supports the combination, in the same program, of components developed in the two languages. Finally, Objective Caml exceptions can be raised and (with some limitations) handled from C.

To Learn More

For a better understanding of the C language, especially argument passing and data representations, the book *C: a reference manual* [HS94] is highly recommended.

Concerning exceptions and garbage collection, several works add these missing features to C. The technical report [Rob89] describes an implementation of exceptions in C, based on open macros and on the `setjmp` and `longjmp` functions from the C library. Hans Boehm distributes a conservative collector with ambiguous roots that can be added (as a library) to any C program:

Link: http://www.hpl.hp.com/personal/Hans_Boehm/gc/

Concerning interoperability between Objective Caml and C, the tools described in this chapter are rather low-level and difficult to use. However, they give the programmer full control on copying or sharing of data structures between the two languages. A higher-level tool called `CamlIDL` is available; it automatically generates the Objective Caml "stubs" (encapsulation functions) for calling C functions and converting data types. The C types and functions are described in a language called IDL (Interface Definition Language), similar to a subset of C++ and C. This description is then passed through the `CamlIDL` compiler, which generates the corresponding `.mli`, `.ml` and `.c` files. This tool is distributed from the following page:

Link: http://caml.inria.fr/camlidl/

Other interfaces exist between Objective Caml and languages other than C. They are available on the "Caml hump" page:

Link: http://caml.inria.fr/hump.html

They include several versions of interfaces with Fortran, and also an Objective Caml bytecode interpreter written in Java.

Finally, interoperability between Objective Caml and other languages can also be achieved via data exchanges between separate programs, possibly over the network. This approach is described in the chapter on distributed programming (see chapter 20).