OCaml — C API

Declaring Primitives

1 to 5 arguments:
`external` *ocamlfun* `:` *ocamltype* `= "c_stub"`
more than 5 arguments:
`external` *ocamlfun* `:` *ocamltype* `= "tupled_C_stub" "c_stub"`
The C_stub function has as many arguments as the OCaml function, while tupled_C_stub has the C prototype:
`CAMLprim value` tuple_c_stub`(value * argv, int argn)`
where `argn` is the number of values stored in `argv`.

Include Files

`caml/mlvalues.h`	value type, and conversion macros
`caml/alloc.h`	allocation functions
`caml/memory.h`	memory-related functions and macros
`caml/fail.h`	functions for raising exceptions
`caml/callback.h`	callback from C to OCaml
`caml/custom.h`	operations on custom blocks
`caml/intext.h`	de/serialization for custom blocks

Linking with C code

Static Linking

`-custom`

Dynamic Linking

`.so` and `.dll`

Value Representation

The `value` type is either (1) an unboxed integer, (2) a pointer to a block inside the OCaml heap, or (3) a pointer outside the OCaml heap (not scanned by GC).
A `value` pointing within OCaml heap should always point to the beginning of an OCaml block.
Integers are encoded as (n<<1) + 1, and so limited to 31 bits on 32 bits arch, 63 bits on 64 bits arch.
Blocks in the heap are arrays of `value`s, preceded by a header `value`. The header `value` contains the number of `value`s in the array, a tag on one character, and 3 bits for garbage-collection.
Interesting tag values:

0..`No_scan_tag` − 1	Standard OCaml Blocks.
`Closure_tag`	A closure, i.e. code pointers plus env.
`String_tag`	An OCaml string.
`Double_tag`	An OCaml float.
`Double_array_tag`	An Array of OCaml floats.
`Abstract_tag`	A block that should not be scanned.
`Custom_tag`	A block with custom functions.

Representations

The 6 GC Safety Rules

Rule 1: A function that has parameters or local variables of type `value` must begin with a call to one of the `CAMLparam` macros and return with `CAMLreturn`, `CAMLreturn0`, or `CAMLreturnT`.

Rule 2: Local variables of type value must be declared with one of the `CAMLlocal` macros. Arrays of values are declared with `CAMLlocalN`. These macros must be used at the beginning of the function, not in a nested block.

Rule 3: Assignments to the fields of structured blocks must be done with the `Store_field` macro (for normal blocks) or `Store_double_field` macro (for arrays and records of floating-point numbers). Other assignments must not use `Store_field` nor `Store_double_field`.

Rule 4: Global variables containing values must be registered with the garbage collector using the `caml_register_global_root` function.

Rule 5: After a structured block (a block with tag less than `No_scan_tag`) is allocated with the low-level functions, all fields of this block must be filled with well-formed values before the next allocation operation. If the block has been allocated with `caml_alloc_small`, filling is performed by direct assignment to the fields of the block:
`Field(v, n) = vn ;`
If the block has been allocated with `caml_alloc_shr`, filling is performed through the `caml_initialize` function:
`caml_initialize(&Field(v, n), vn );`

Rule 6: Direct assignment to a field of a block, as in
`Field(v, n) = w;`
is safe only if `v` is a block newly allocated by `caml_alloc_small`; that is, if no allocation took place between the allocation of `v` and the assignment to the field. In all other cases, never assign directly. If the block has just been allocated by `caml_alloc_shr`, use `caml_initialize` to assign a value to a field for the first time:
`caml_initialize(&Field(v, n), w);`
Otherwise, you are updating a field that previously contained a well-formed value; then, call the `caml_modify function`:
`caml_modify(&Field(v, n), w);`

The “noalloc” flag

Macros

`Is_long(v)` is true if value v is an immediate integer, false otherwise
`Is_block(v)` is true if value v is a pointer to a block, and false if it is an immediate integer.
`Val_long(l)` C long int to OCaml int.
`Long_val(v)` OCaml int to C long int.
`Val_int(i)` C int to OCaml int.
`Int_val(v)` OCaml int to C int.
`Val_bool(x)` C truth value to OCaml bool.
`Bool_val(v)` OCaml bool to C truth value.
`Val_true`, `Val_false` OCaml true and false.
`Wosize_val(v)` returns the size of the block v, in words, excluding the header.
`Tag_val(v)` returns the tag of the block v.
`Field(v, n)` returns the value contained in the nth field of the structured block v. Fields are numbered from 0 to `Wosize_val(v) -- 1`.
`Store_field(b, n, v)` stores the value v in the field number n of value b, which must be a structured block.
`caml_string_length(v)` returns the length (number of characters) of the string v.
`String_val(v)` returns a pointer to the first byte of the string v, with type char *. This pointer is a valid C string: there is a null character after the last character in the string. However, Caml strings can contain embedded null characters, that will confuse the usual C functions over strings.
`Double_val(v)` returns the floating-point number contained in value v, with type double.
`Double_field(v, n)` returns the nth element of the array of floating-point numbers v (a block tagged `Double_array_tag`).
`Store_double_field(v, n, d)` stores the double precision floating-point number d in the nth element of the array of floating-point numbers v.

Allocation Functions

`caml_alloc(n, t)` returns a fresh block of size n with tag t. If t is less than `No_scan_tag`, then the fields of the block are initialized with a valid value in order to satisfy the GC constraints.

`caml_alloc_tuple(n)` returns a fresh block of size n words, with tag 0.

`caml_alloc_string(n)` returns a string value of length n characters. The string initially contains garbage.

`caml_copy_string(s)` returns a string value containing a copy of the null-terminated C string s (a char *).

`caml_copy_double(d)` returns a floating-point value initialized with the double d.

Functions

Callbacks from C to OCaml