24 Memory Representation of Values

The FFI interface we described in Chapter 23 (Foreign Function Interface) hides the precise details of how values are exchanged across C libraries and the OCaml runtime. There is a simple reason for this: using this interface directly is a delicate operation that requires understanding a few different moving parts before you can get it right. You first need to know the mapping between OCaml types and their runtime memory representation. You also need to ensure that your code is interfacing correctly with OCaml runtime's memory management.

However, knowledge of the OCaml internals is useful beyond just writing foreign function interfaces. As you build and maintain more complex OCaml applications, you'll need to interface with various external system tools that operate on compiled OCaml binaries. For example, profiling tools report output based on the runtime memory layout, and debuggers execute binaries without any knowledge of the static OCaml types. To use these tools effectively, you'll need to do some translation between the OCaml and C worlds.

Luckily, the OCaml toolchain is very predictable. The compiler minimizes the amount of optimization magic that it performs, and relies instead on its straightforward execution model for good performance. With some experience, you can know rather precisely where a block of performance-critical OCaml code is spending its time.

Why Do OCaml Types Disappear at Runtime?

The OCaml compiler runs through several phases during the compilation process. The first phase is syntax checking, during which source files are parsed into abstract syntax trees (ASTs). The next stage is a *type checking* pass over the AST. In a validly typed program, a function cannot be applied with an unexpected type. For example, the `print_endline` function must receive a single `string` argument, and an `int` will result in a type error.

Since OCaml verifies these properties at compile time, it doesn't need to keep track of as much information at runtime. Thus, later stages of the compiler can discard and simplify the type declarations to a much more minimal subset that's actually required to distinguish polymorphic values at runtime. This is a major performance win versus something like a Java or .NET method call, where the runtime must look up the concrete instance of the object and dispatch the method call. Those languages amortize some of

https://doi.org/10.1017/9781009129220.028 Published online by Cambridge University Press

the cost via "Just-in-Time" dynamic patching, but OCaml prefers runtime simplicity instead.

We'll explain this compilation pipeline in more detail in Chapter 26 (The Compiler Frontend: Parsing and Type Checking) and Chapter 27 (The Compiler Backend: Bytecode and Native code).

This chapter covers the precise mapping from OCaml types to runtime values and walks you through them via the toplevel. We'll cover how these values are managed by the runtime later on in Chapter 25 (Understanding the Garbage Collector).

24.1 OCaml Blocks and Values

A running OCaml program uses blocks of memory (i.e., contiguous sequences of words in RAM) to represent values such as tuples, records, closures, or arrays. An OCaml program implicitly allocates a block of memory when such a value is created:

```
# type t = { foo: int; bar: int };;
type t = { foo : int; bar : int; }
# let x = { foo = 13; bar = 14 };;
val x : t = {foo = 13; bar = 14}
```

The type declaration `t` doesn't take up any memory at runtime, but the subsequent `let` binding allocates a new block of memory with two words of available space. One word holds the `foo` field, and the other word holds the `bar` field. The OCaml compiler translates such an expression into an explicit allocation for the block from OCaml's runtime system.

OCaml uses a uniform memory representation in which every OCaml variable is stored as a *value*. An OCaml value is a single memory word that is either an immediate integer or a pointer to some other memory. The OCaml runtime tracks all values so that it can free them when they are no longer needed. It thus needs to be able to distinguish between integer and pointer values, since it scans pointers to find further values but doesn't follow integers that don't point to anything meaningful beyond their immediate value.

24.1.1 Distinguishing Integers and Pointers at Runtime

Wrapping primitive types (such as integers) inside another data structure that records extra metadata about the value is known as *boxing*. Values are boxed in order to make it easier for the garbage collector (GC) to do its job, but at the expense of an extra level of indirection to access the data within the boxed value.

OCaml values don't all have to be boxed at runtime. Instead, values use a single tag bit per word to distinguish integers and pointers at runtime. The value is an integer if the lowest bit of the block word is nonzero, and a pointer if the lowest bit of the block word is zero. Several OCaml types map onto this integer representation, including `bool`, `int`, the empty list, and `unit`. Some types, like variants, sometimes use this integer representation and sometimes don't. In particular, for variants, constant constructors, i.e., constructors with no arguments like `None`, are represented as integers, but constructors like `Some` that carry associated values are boxed.

This representation means that integers are unboxed runtime values in OCaml so that they can be stored directly without having to allocate a wrapper block. They can be passed directly to other function calls in registers and are generally the cheapest and fastest values to use in OCaml.

A value is treated as a memory pointer if its lowest bit is zero. A pointer value can still be stored unmodified despite this, since pointers are guaranteed to be word-aligned (with the bottom bits always being zero).

The only problem that remains with this memory representation is distinguishing between pointers to OCaml values (which should be followed by the GC) and pointers into the system heap to C values (which shouldn't be followed).

The mechanism for this is simple, since the runtime system keeps track of the heap blocks it has allocated for OCaml values. If the pointer is inside a heap chunk that is marked as being managed by the OCaml runtime, it is assumed to point to an OCaml value. If it points outside the OCaml runtime area, it is treated as an opaque C pointer to some other system resource.

Some History About OCaml's Word-Aligned Pointers

The alert reader may be wondering how OCaml can guarantee that all of its pointers are word-aligned. In the old days, when RISC chips such as Sparc, MIPS, and Alpha were commonplace, unaligned memory accesses were forbidden by the instruction set architecture and would result in a CPU exception that terminated the program. Thus, all pointers were historically rounded off to the architecture word size (usually 32 or 64 bits).

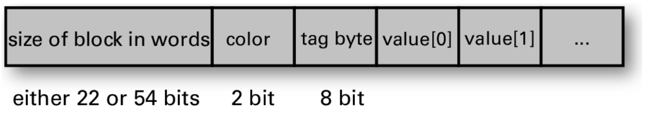
Modern CISC processors such as the Intel x86 do support unaligned memory accesses, but the chip still runs faster if accesses are word-aligned. OCaml therefore simply mandates that all pointers be word-aligned, which guarantees that the bottom few bits of any valid pointer will be zero. Setting the bottom bit to a nonzero value is a simple way to mark an integer, at the cost of losing that single bit of precision.

An even more alert reader will be wondering about the performance implications are for integer arithmetic using this tagged representation. Since the bottom bit is set, any operation on the integer has to shift the bottom bit right to recover the "native" value. The native code OCaml compiler generates efficient x86 assembly code in this case, taking advantage of modern processor instructions to hide the extra shifts where possible. Addition is a single `LEA` x86 instruction, subtraction can be two instructions, and multiplication is only a few more.

24.2 Blocks and Values

An OCaml *block* is the basic unit of allocation on the heap. A block consists of a one-word header (either 32 or 64 bits depending on the CPU architecture) followed by variable-length data that is either opaque bytes or an array of *fields*. The header has a multipurpose tag byte that defines whether to interpret the subsequent data as opaque bytes or OCaml fields.

The GC never inspects opaque bytes. If the tag indicates an array of OCaml fields are present, their contents are all treated as more valid OCaml values. The GC always inspects fields and follows them as part of the collection process described earlier.



The `size` field records the length of the block in memory words. This is 22 bits on 32-bit platforms, which is the reason OCaml strings are limited to 16 MB on that architecture. If you need bigger strings, either switch to a 64-bit host, or use the `Bigarray` module.

The 2-bit `color` field is used by the GC to keep track of its state during mark-and-sweep collection. We'll come back to this field in Chapter 25 (Understanding the Garbage Collector). This tag isn't exposed to OCaml source code in any case.

A block's tag byte is multipurpose, and indicates whether the data array represents opaque bytes or fields. If a block's tag is greater than or equal to `No_scan_tag` (251), then the block's data are all opaque bytes, and are not scanned by the collector. The most common such block is the `string` type, which we describe in more detail later in this chapter.

The exact representation of values inside a block depends on their static OCaml type. All OCaml types are distilled down into `values`, and summarized below.

- `int` or `char` are stored directly as a value, shifted left by 1 bit, with the least significant bit set to 1.
- `unit`, `[]`, `false` are all stored as OCaml `int` 0.
- `true` is stored as OCaml `int` 1.
- `Foo | Bar` variants are stored as ascending OCaml `ints`, starting from 0.
- `Foo | Bar of int` variants with parameters are boxed, while variants with no parameters are unboxed.
- Polymorphic variants with parameters are boxed with an extra header word to store the value, as compared to normal variants. Polymorphic variants with no parameters are unboxed.
- Floating-point numbers are stored as a block with a single field containing the double-precision float.
- Strings are word-aligned byte arrays with an explicit length.

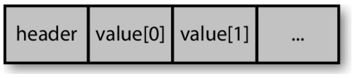
- `[1; 2; 3]` lists are stored as `1::2::3::[]` where `[]` is an int, and `h::t` a block with tag 0 and two parameters.
- Tuples, records, and arrays are stored as a C array of values. Arrays can be variable size, but tuples and records are fixed-size.
- Records or arrays that are all float use a special tag for unboxed arrays of floats, or records that only have `float` fields.

24.2.1 Integers, Characters, and Other Basic Types

Many basic types are efficiently stored as unboxed integers at runtime. The native `int` type is the most obvious, although it drops a single bit of precision due to the tag bit. Other atomic types such as `unit` and the empty list `[]` value are stored as constant integers. Boolean values have a value of `1` and `0` for `true` and `false`, respectively.

These basic types such as empty lists and `unit` are very efficient to use, since integers are never allocated on the heap. They can be passed directly in registers and not appear on the stack if you don't have too many parameters to your functions. Modern architectures such as `x86_64` have a lot of spare registers to further improve the efficiency of using unboxed integers.

24.3 Tuples, Records, and Arrays



Tuples, records, and arrays are all represented identically at runtime as a block with tag `0`. Tuples and records have constant sizes determined at compile time, whereas arrays can be of variable length. While arrays are restricted to containing a single type of element in the OCaml type system, this is not required by the memory representation.

You can check the difference between a block and a direct integer yourself using the `Obj` module, which exposes the internal representation of values to OCaml code:

```
# Obj.is_block (Obj.repr (1,2,3));;
- : bool = true
# Obj.is_block (Obj.repr 1);;
- : bool = false
```

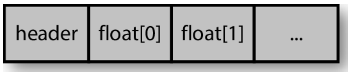
The `Obj.repr` function retrieves the runtime representation of any OCaml value. `Obj.is_block` checks the bottom bit to determine if the value is a block header or an unboxed integer.

24.3.1 Floating-Point Numbers and Arrays

Floating-point numbers in OCaml are always stored as full, double-precision values. Individual floating-point values are stored as a block with a single field that contains the number. This block has the `Double_tag` set, which signals to the collector that the floating-point value is not to be scanned:

```
# Obj.tag (Obj.repr 1.0);;
- : int = 253
# Obj.double_tag;;
- : int = 253
```

Since each floating-point value is boxed in a separate memory block, it can be inefficient to handle large arrays of floats in comparison to unboxed integers. OCaml therefore special-cases records or arrays that contain *only* `float` types. These are stored in a block that contains the floats packed directly in the data section, with `Double_array_tag` set to signal to the collector that the contents are not OCaml values.



First, let's check that float arrays do in fact have a different tag number from normal floating-point values:

```
# Obj.double_tag;;
- : int = 253
# Obj.double_array_tag;;
- : int = 254
```

This tells us that float arrays have a tag value of 254. Now let's test some sample values using the `Obj.tag` function to check that the allocated block has the expected runtime tag, and also use `Obj.double_field` to retrieve a float from within the block:

```
# Obj.tag (Obj.repr [| 1.0; 2.0; 3.0 |]);;
- : int = 254
# Obj.tag (Obj.repr (1.0, 2.0, 3.0) );;
- : int = 0
# Obj.double_field (Obj.repr [| 1.1; 2.2; 3.3 |]) 1;;
- : float = 2.2
# Obj.double_field (Obj.repr 1.234) 0;;
- : float = 1.234
```

The first thing we tested was that a float array has the correct unboxed float array tag value (254). However, the next line tests a tuple of floating-point values instead, which are *not* optimized in the same way and have the normal tuple tag value (0).

Only records and arrays can have the float array optimization, and for records, every single field must be a float.

24.4 Variants and Lists

Basic variant types with no extra parameters for any of their branches are simply stored as an OCaml integer, starting with `0` for the first option and in ascending order:

```
# type t = Apple | Orange | Pear;;
type t = Apple | Orange | Pear
# ((Obj.magic (Obj.repr Apple)) : int);;
- : int = 0
# ((Obj.magic (Obj.repr Pear)) : int);;
- : int = 2
# Obj.is_block (Obj.repr Apple);;
- : bool = false
```

`Obj.magic` unsafely forces a type cast between any two OCaml types; in this example, the `int` type hint retrieves the runtime integer value. The `Obj.is_block` confirms that the value isn't a more complex block, but just an OCaml `int`.

Variants that have parameters are a little more complex. They are stored as blocks, with the value *tags* ascending from 0 (counting from leftmost variants with parameters). The parameters are stored as words in the block:

```
# type t = Apple | Orange of int | Pear of string | Kiwi;;
type t = Apple | Orange of int | Pear of string | Kiwi
# Obj.is_block (Obj.repr (Orange 1234));;
- : bool = true
# Obj.tag (Obj.repr (Orange 1234));;
- : int = 0
# Obj.tag (Obj.repr (Pear "xyz"));;
- : int = 1
# (Obj.magic (Obj.field (Obj.repr (Orange 1234)) 0) : int);;
- : int = 1234
# (Obj.magic (Obj.field (Obj.repr (Pear "xyz")) 0) : string);;
- : string = "xyz"
```

In the preceding example, the `Apple` and `Kiwi` values are still stored as normal OCaml integers with values `0` and `1`, respectively. The `Orange` and `Pear` values both have parameters and are stored as blocks whose tags ascend from `0` (and so `Pear` has a tag of `1`, as the use of `Obj.tag` verifies). Finally, the parameters are fields that contain OCaml values within the block, and `Obj.field` can be used to retrieve them.

Lists are stored with a representation that is exactly the same as if the list was written as a variant type with `Nil` and `Cons`. The empty list `[]` is an integer `0`, and subsequent blocks have tag `0` and two parameters: a block with the current value, and a pointer to the rest of the list.

Obj Module Considered Harmful

`Obj` is an undocumented module that exposes the internals of the OCaml compiler and runtime. It is very useful for examining and understanding how your code will behave at runtime but should *never* be used for production code unless you understand the implications. The module bypasses the OCaml type system, making memory corruption and segmentation faults possible.

Some theorem provers such as Coq do output code that uses `Obj` internally, but the external module signatures never expose it. Unless you too have a machine proof of correctness to accompany your use of `Obj`, stay away from it except for debugging!

Due to this encoding, there is a limit around 240 variants with parameters that applies to each type definition, but the only limit on the number of variants without parameters is the size of the native integer (either 31 or 63 bits). This limit arises because of the size of the tag byte, and that some of the high-numbered tags are reserved.

24.5 Polymorphic Variants

Polymorphic variants are more flexible than normal variants when writing code but are slightly less efficient at runtime. This is because there isn't as much static compile-time information available to optimize their memory layout.

A polymorphic variant without any parameters is stored as an unboxed integer and so only takes up one word of memory, just like a normal variant. This integer value is determined by applying a hash function to the *name* of the variant. The hash function is exposed via the `compiler-libs` package that reveals some of the internals of the OCaml compiler:

```
# #require "ocaml-compiler-libs.common";;
# Btype.hash_variant "Foo";;
- : int = 3505894
# (Obj.magic (Obj.repr `Foo) : int);;
- : int = 3505894
```

The hash function is designed to give the same results on 32-bit and 64-bit architectures, so the memory representation is stable across different CPUs and host types.

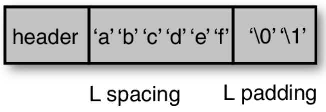
Polymorphic variants use more memory space than normal variants when parameters are included in the data type constructors. Normal variants use the tag byte to encode the variant value and save the fields for the contents, but this single byte is insufficient to encode the hashed value for polymorphic variants. They must allocate a new block (with tag `0`) and store the value in there instead. Polymorphic variants with constructors thus use one word of memory more than normal variant constructors.

Another inefficiency over normal variants is when a polymorphic variant constructor has more than one parameter. Normal variants hold parameters as a single flat block with multiple fields for each entry, but polymorphic variants must adopt a more flexible uniform memory representation, since they may be reused in a different context across compilation units. They allocate a tuple block for the parameters that is pointed to from the argument field of the variant. There are thus three additional words for such variants, along with an extra memory indirection due to the tuple.

The extra space usage is generally not significant in a typical application, and polymorphic variants offer a great deal more flexibility than normal variants. However, if you're writing code that demands high performance or must run within tight memory bounds, the runtime layout is at least very predictable. The OCaml compiler never switches memory representation due to optimization passes. This lets you predict the precise runtime layout by referring to these guidelines and your source code.

24.6 String Values

OCaml `strings` (and their mutable cousins, `bytes`) are standard OCaml blocks with the header size defining the size of the string in machine words. The `String_tag` (252) is higher than the `No_scan_tag`, indicating that the contents of the block are opaque to the collector. The block contents are the contents of the string, with padding bytes to align the block on a word boundary.



On a 32-bit machine, the padding is calculated based on the modulo of the string length and word size to ensure the result is word-aligned. A 64-bit machine extends the potential padding up to 7 bytes instead of 3. Given a string length modulo 4:

- `0` has padding `00 00 00 03`
- `1` has padding `00 00 02`
- `2` has padding `00 01`
- `3` has padding `00`

This string representation is a clever way to ensure that the contents are always zero-terminated by the padding word and to still compute its length efficiently without scanning the whole string. The following formula is used:

```
number_of_words_in_block * sizeof(word) - last_byte_of_block - 1
```

The guaranteed `NULL` termination comes in handy when passing a string to C, but is not relied upon to compute the length from OCaml code. OCaml strings can thus contain `NULL` bytes at any point within the string.

Care should be taken that any C library functions that receive these buffers can also cope with arbitrary bytes within the buffer contents and are not expecting C strings. For instance, the C `memcopy` or `memmove` standard library functions can operate on arbitrary data, but `strlen` or `strcpy` both require a `NULL`-terminated buffer, and neither has a mechanism for encoding a `NULL` value within its contents.

24.7 Custom Heap Blocks

OCaml supports *custom* heap blocks via a `Custom_tag` that lets the runtime perform user-defined operations over OCaml values. A custom block lives in the OCaml heap like an ordinary block and can be of whatever size the user desires. The `Custom_tag` (255) is higher than `No_scan_tag` and so isn't scanned by the GC.

The first word of the data within the custom block is a C pointer to a `struct` of custom operations. The custom block cannot have pointers to OCaml blocks and is opaque to the GC:

```
struct custom_operations {
  char *identifier;
  void (*finalize)(value v);
  int (*compare)(value v1, value v2);
  intnat (*hash)(value v);
  void (*serialize)(value v,
                    /*out*/ uintnat * wsize_32 /*size in bytes*/,
                    /*out*/ uintnat * wsize_64 /*size in bytes*/);
  uintnat (*deserialize)(void * dst);
  int (*compare_ext)(value v1, value v2);
};
```

The custom operations specify how the runtime should perform polymorphic comparison, hashing and binary marshaling. They also optionally contain a *finalizer* that the runtime calls just before the block is garbage-collected. This finalizer has nothing to do with ordinary OCaml finalizers (as created by `Gc.finalize` and explained in Chapter 25 (Understanding the Garbage Collector)). They are instead used to call C cleanup functions such as `free`.

24.7.1 Managing External Memory with Bigarray

A common use of custom blocks is to manage external system memory directly from within OCaml. The Bigarray interface was originally intended to exchange data with Fortran code, and maps a block of system memory as a multidimensional array that can be accessed from OCaml. Bigarray operations work directly on the external memory without requiring it to be copied into the OCaml heap (which is a potentially expensive operation for large arrays).

Bigarray sees a lot of use beyond just scientific computing, and several Core libraries use it for general-purpose I/O:

`Iobuf` The `Iobuf` module maps I/O buffers as a one-dimensional array of bytes. It provides a sliding window interface that lets consumer processes read from the buffer while it's being filled by producers. This lets OCaml use I/O buffers that have been externally allocated by the operating system without any extra data copying.

`Bigstring` The `Bigstring` module provides a `String`-like interface that uses `Bigarray` internally. The `Bigbuffer` collects these into extensible string buffers that can operate entirely on external system memory.

The Lacaml[1] library isn't part of Core but provides the recommended interfaces to the widely used BLAS and LAPACK mathematical Fortran libraries. These allow

[1] http://mmottl.github.io/lacaml/

developers to write high-performance numerical code for applications that require linear algebra. It supports large vectors and matrices, but with static typing safety of OCaml to make it easier to write safe algorithms.