# Richard WM Jones

AUGUST 4, 2009 · 6:09 PM

# A beginners guide to OCaml internals

In this 6 part series, I'm going to introduce the internals of the OCaml programming language (tutorial and other references here). This isn't going to be very comprehensive or in-depth. It's more of a digest of the readily available information that I found by reading the manual, header files, and some of the compiler code. It's definitely pitched at beginners, not experts.

By the end of it, you should be able to look at small pieces of OCaml and work out in your head how they get translated into machine code.

I'm definitely not a great expert on the internals of OCaml. This means a couple of things: (a) I've probably made some mistakes (post a comment if you see any). (b) I might "reveal too much" of the specifics of the current implementation: anything could change in a future version of OCaml, although the internals described here have been pretty stable for a long time.

Without further ado …

## Part 1: Values

In a running OCaml program, a **value** is either an "integer-like thing" or a pointer.

What's an "integer-like thing"? Any OCaml `int` or `char`, or some things which are stored like integers for speed, which include the constants `true` and `false`, the empty list `[]`, unit `()`, and some (but not all) variants.

For reasons that I'll explain later, OCaml stores values which are integer-like and values which are pointers differently.

A value containing a pointer is just stored as the pointer. Because addresses are always word-aligned, the bottom 2 bits of every pointer are always `0 0` (the bottom 3 bits are `0 0 0` for a 64 bit machine). OCaml therefore stores integers by setting the bottom bit to `1` and shifting the integer over 1 place:

```
+----------------------------------------+---+---+
| pointer                                | 0 | 0 |
+----------------------------------------+---+---+

+--------------------------------------------+---+
| integer (31 or 63 bits)                    | 1 |
+-------------------------------------------+---+
```

You can quickly tell if a value is an integer-like thing (ie. is the bottom bit 1?).

OCaml gives you some C macros and OCaml functions you can use on values. The C macros are `Is_long` (is it an integer?) and `Is_block` (is it a pointer?) in the header file `<caml/mlvalues.h>`. The OCaml functions are `is_int` and `is_block` (for pointers) in the "infamous" `Obj` module.

```
# let what_is_it foo =
    let foo_repr = Obj.repr foo in
    if Obj.is_int foo_repr then "integer-like" else "pointer" ;;
val what_is_it : 'a -> string = <fun>
# what_is_it 42 ;;
- : string = "integer-like"
# what_is_it () ;;
- : string = "integer-like"
# what_is_it [1;2;3] ;;
- : string = "pointer"
# type bar = Bar | Baz of int ;;
type bar = Bar | Baz of int
# what_is_it Bar ;;
- : string = "integer-like"
# what_is_it (Baz 5) ;;
- : string = "pointer"
```

Are ints slow? After all it seems like you have to do a lot of bit shifting to do any arithmetic with them.

It turns out that they are a little bit slower, but not by very much. The OCaml compiler can perform most arithmetic operations in between one and four machine instructions, and usually just one or two, making it the same or fractionally slower than a straight native integer. (Of course, there is a more serious penalty in the cases where you really need the full width of a 32 or 64 bit integer, for example in crypto or compression algorithms).

| | |
|---|---|
| Increment | addl $2, %eax |
| Add a constant | addl (constant*2), %eax |
| Add two values | lea -1(%eax, %ebx), %eax |
| Subtract | subl %ebx, %eax<br>incl %eax |

That these are equivalent isn't totally obvious, but this post explains why.

That concludes part 1. Tune in tomorrow for part 2, or if you're reading this *in the future*, use this handy table of contents:

- Part 2: Strings and other types
- Part 3: The minor heap
- Part 4: The major heap
- Part 5: Garbage collection
- Part 6: Conclusion, further reading

## Update

This guide is discussed on reddit here. Also on Hacker News here.

Share

- 
- 
- 
-

Feedback

Share this:

Reddit

Twitter 18

Email

Print

## 11 Responses to *A beginners guide to OCaml internals*

**Jon Harrop**

August 5, 2009 at 2:06 pm

Article was good but fish face at the top scared the shit out of me!

Reply

Pingback: OCaml internals part 2: Strings and other types « Richard WM Jones

**ted**
August 5, 2009 at 5:51 pm

Why does OCaml need to use the first two bits for remembering whether a value is an integer-like thing or a pointer? I thought it had this information at compile time.

Reply

**OlivierA**
August 11, 2009 at 1:34 pm

[…] I thought it had this information at compile time. […]

No it does not … for variant types a value is either represented by an integer or by a pointer. For instance, lists: the empty list is the integer 0, a cons cell is a pointer. So it needs to know at run-time whether the value is an int or a pointer.

Reply

**rixed**
August 11, 2009 at 8:39 pm

OCaml does know at compile time, except perhaps in some strange corner cases. Check some compiled code and you will see no check for bit 0, etc, like you would find on dynamically typed languages.

It's the garbage collector that needs this information.

Reply

**rich**
August 5, 2009 at 7:25 pm

> For reasons that I'll explain later, OCaml stores values which are integer-like and values which are pointers differently.

It's explained in part 4, but the reason the least significant bit is different is because the garbage collector needs to know not to follow integers.

Reply

Pingback: OCaml internals part 3: The minor heap « Richard WM Jones

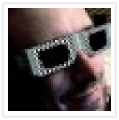
## Carl

August 15, 2009 at 7:39 pm

Is there a way to get around the integer performance hit when 32/64 bits are needed for crypto alogrithms?

Reply

### rich



August 15, 2009 at 8:09 pm

If the algorithm uses arrays of 32/64 bit numbers, then you can use a Bigarray.

But really if it's a performance critical crypto algorithm then you're far better off using an existing package, which is probably written in hand-vectorized assembler.

See also things like liboil …

Reply

Pingback: OCaml internals part 5: Garbage collection « Richard WM Jones

Pingback: Pattern matching costs | arlen