Richard WM Jones

AUGUST 5, 2009 · 4:00 PM

OCaml internals part 2: Strings and other types

In part 1 we saw how integers and some integer-like things are represented at runtime in OCaml.

Objects which are large or more complex than simple integers are stored in OCaml blocks. An **OCaml block** consists of a header followed by an array of words. **Word** in this context means a 4 or 8 byte quantity (for 32 and 64 bit platforms respectively) which can contain a value or something else.

Actually if you have a value which is a pointer to an OCaml block, then (for performance reasons) it points to the zeroth element of the array, and the header is at a negative offset, -4 or -8.

```
+---------------+---------------+---------------+- - - -
| header        | word[0]       | word[1]       | ....
+---------------+---------------+---------------+- - - -
                  ^
                  |
              pointer (a value)
```

What we've described so far (values and blocks) turns out to be a complete description of how everything is stored in an OCaml program. [OK, not 100% true: you can also have pointers to C structs]

So how do all the OCaml types map to values and blocks? At the end of this part you'll find a comprehensive table showing the mapping for every type. Let's start with some easy ones first.

An OCaml string is just stored as a byte array, with enough words allocated for the required size of the string:

```
+---------------+----------------+------------------+
| header        | 'a' 'b' 'c' 'd' 'e' 'f' '\O' '\1' |
+---------------+----------------+------------------+
                  ^
                  |
              an OCaml string
```

(There's a bit of extra cleverness with strings described here).

An array is just stored as an array of values, with each word corresponding to one value:

```
+----------------+----------------+----------------+- - - - -
| header         | value[0]       | value[1]       | ....
+----------------+----------------+----------------+- - - - -
                   ^
                   |
              an OCaml array
```

As I described in the previous part, simple variants are just stored as integers, but if a variant has any parameter, then it's stored as a block:

```
+----------------+----------------+
| header         | arg[0]         |
+----------------+----------------+
                   ^
                   |
              a variant with one arg
```

Let's take a closer look at the header:

```
+----------------+----------------+-----------+--+--+----------------+
| size of the block in words                  | col | tag byte       |
+----------------+----------------+-----------+--+--+----------------+
 ^                                            <- 2b-><--- 8 bits --->
 |
offset -4 or -8
```

The size part is simple enough – it just records the number of words in the block, eg. the length of the array. Note on 32 bit platforms, this is 22 bits long, which is the reason for the annoying 16 MByte limit on strings (16 MBytes = 4 M-words = 2^{22} words). Solution: use a 64 bit platform for serious OCaml.

The tag byte is multipurpose. For example, in the variant-with-parameter example above, it tells you which variant it is. In the string case, it contains a little bit of runtime type information (it's a string). In other cases it can tell the garbage collector that it's a lazy value, or opaque data that the garbage collector shouldn't scan.

"col" (color) is two bits used by the garbage collector which I won't go into, but you can find out about coloring algorithms in any good book on garbage collection.

One interesting little optimization that OCaml does is for arrays of floating point numbers. So that OCaml does well at numeric programs, arrays of floats are stored inline like this:

```
+--------------+--------------+--------------+- - - -
| header       | float[0]                    | ....
+--------------+--------------+--------------+- - - -
                ^
                |
            an OCaml float array
```

Where do these OCaml blocks live? That's the subject of part 3 …

Reference table

Chapter 18 of the OCaml manual describes the representation of blocks in some detail, but not very clearly, so here is a table showing how OCaml objects map into values and blocks.

OCaml	Representation
any int, char	stored directly as a value, shifted left by 1 bit, with LSB = 1
`()`, `[]`, `false`	stored as OCaml int 0 (native int 1)
`true`	stored as OCaml int 1 (native int 3)
variant `type t = Foo | Bar | Baz` (no parameters)	stored as OCaml int 0, 1, 2
variant `type t = Foo | Bar of int`	The variants with no parameters are stored as OCaml int 0, 1, 2, etc. counting from the leftmost, counting just the variants that have no parameters. The variants with parameters are stored as blocks, with tag 0, 1, 2, etc. counting from leftmost, counting just the variants with parameters. The parameters are stored as words in the block itself. Note there is a limit around 240 variants with parameters that applies to each type, but no limit on the number of variants without parameters you can have. This limit arises because of the size of the tag byte and the fact that some of the high numbered tags are reserved.
list `[1; 2; 3]`	This is represented as `1 :: 2 :: 3 :: []` where `[]` is a value OCaml int 0, and `h :: t` is a block with tag 0 and two parameters. This representation is exactly the same as if list was a variant.
tuples, `struct` and `array`	These are all represented identically, as a simple array of values. The tag is 0. The only difference is that an array can be allocated with variable size, but structs and tuples always have a fixed size.
struct or array where every element is a float	These are treated as a special case, as described above. The tag has the special value `Double_array_tag (254)` so that the garbage collector knows how to deal with these. Note this exception does *not* apply to tuples that contain floats. Beware anyone who would declare a vector as (1.0, 2.0).

any string	Strings are byte arrays in OCaml, but they have quite a clever representation to make it very efficient to get their length, and at the same time make them directly compatible with C strings. The tag is set to `String_tag (252)`.

For full details of blocks, read the comments in the <caml/mlvalues.h> C header file that comes with OCaml.

Share this:

- Reddit
- Twitter
- Email
- Print

6 Responses to *OCaml internals part 2: Strings and other types*

Pingback: A beginners guide to OCaml internals « Richard WM Jones

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

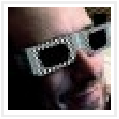
khigia
August 25, 2009 at 12:48 am

Many thanks for this very useful series of posts!

Little question about "cleverness with string" (in the link you included): the formula for string length calculation seems a bit weird, or at least I can't figure out how this works.
number_of_words_in_block * sizeof(word) + last_byte_of_block – 1

Given the padding to word boundary, the way I understand it would make more sense if formula was:
number_of_words_in_block * sizeof(word) – last_byte_of_block – 1
(note the minus before the padding value)
Am I getting something wrong?

Reply



rich
August 25, 2009 at 8:51 am

If you look at the source (byterun/str.c) you can see how the length is really calculated:

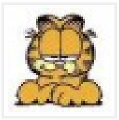
```
mlsize_t
caml_string_length(value s)
{
  mlsize_t temp;
  temp = Bosize_val(s) - 1;
  Assert (Byte (s, temp - Byte (s, temp)) == 0);
  return temp - Byte (s, temp);
}
```

temp = number_of_words_in_block * sizeof (word) – 1

The returned value is temp – last_byte_of_block.

So it looks like you're right.

Reply



khigia
August 25, 2009 at 1:26 pm

Thanks rich!
One day I'll dig into OCaml source code … it's on the infinite TODO list

Mykola
June 28, 2011 at 1:28 pm

Thanks a lot, Richard. Especially, for note about float arrays. It really, really helpful.

Reply