Richard WM Jones

AUGUST 6, 2009 · 5:00 PM

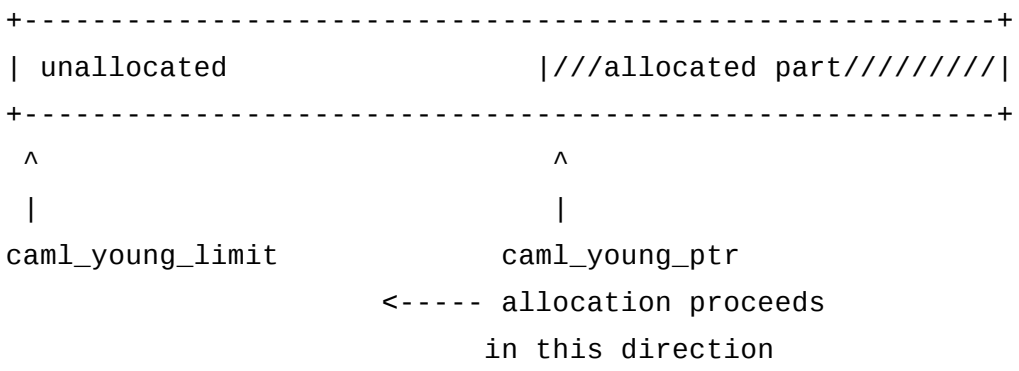
OCaml internals part 3: The minor heap

In part 1 we learned about values and in part 2 we learned about blocks. Values and blocks are the toolkit used to make every OCaml type. But where do OCaml blocks live? In this part we'll start to look at the **OCaml heap**.

Most OCaml blocks are created in the **minor (or young) heap**.

The minor heap is a very simple structure, easy to understand but also very fast when used. It is simply a block of memory, allocated using the C malloc function. The memory is 32 K-words by default, which is 128K or 256K on 32 and 64 bit platforms respectively (you can change this size if you want).

So it's just a block of memory. This is how it works:



Remember from part 2 that blocks contain a header and some words. Consider an array of two elements. The total size of this object will be three words (header + 2 words), so 12 bytes on an i386. To allocate this object is very simple — you just subtract 12 from caml_young_ptr and that gives you the address to store the new object.

Of course it's not quite so simple because you have to check that you don't run out of space, so the fast path for allocations is:

1. Subtract *size* from caml_young_ptr.
2. If caml_young_ptr < caml_young_limit, then take the slow path through the garbage collector.
3. Erm, that's it!

The fast path is just five machine instructions and no branches. This is a good thing because functional languages do a lot more allocations than imperative languages, so allocation has to be very cheap.

The OCaml compiler (ocamlopt) on the other hand compiles your code very literally. For example if you write:

```
let f (a,b) = (a,b)
```

then you are asking the compiler to pattern match `(a, b)` and construct (ie. allocate) another tuple `(a, b)`, and that's *exactly* what the compiler will do.

These sorts of "hidden" allocations are a common problem when people are trying to wring the last drop of performance from their OCaml inner loops. Even 5 instructions in an inner loop can be costly, particularly when it's unnecessary as in the function above.

You are better off writing this function like so:

```
let f (x : ('a*'b)) = x
```

which has the same type signature but will be probably an order of magnitude faster.

Tomorrow, in part 4, I'll look at what happens when the minor heap needs to be garbage collected, and introduce the major heap.

3 Responses to *OCaml internals part 3: The minor heap*

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

Pingback: OCaml internals part 4: The major heap « Richard WM Jones

Pingback: Reference counting « Richard WM Jones