Richard WM Jones

AUGUST 8, 2009 · 5:00 PM

OCaml internals part 5: Garbage collection

In part 1 and part 2 I talked about values and blocks, and in part 3 and part 4 I discussed where those are stored in the OCaml heap.

In this part I want to talk about garbage collection, but actually not about how the heap is scanned or colouring algorithms because there are much better books out there on the subject.

Instead I want to talk about some peculiarities of the OCaml garbage collector that might affect the performance of your OCaml programs.

The refs list

Update Although the code calls this the refs list, it's more commonly known in GC terminology as the remembered set.

The first is mutable references which point "backwards" from the major to minor heap. In a functional language which doesn't allow any mutable types, there's one guarantee you can make which is there could never be a pointer going from the major heap to something in the minor heap. The reason should be fairly clear: You can't create a new object that points to a future object that's not been created yet, and because the language doesn't allow mutability you couldn't update the old object when the new object gets created. So when an object in an immutable language graduates from the minor heap to the major heap, it is fixed forever (until it becomes unreachable), and can't ever point back to the minor heap.

Of course OCaml *allows* mutable objects:

```
type a = { mutable mb : b }
and b = { m : int }

let set a b =
  a.mb <- b
```

So in OCaml we can indeed imagine a situation where an object `a` has migrated to the major heap, and we call `set` to point its `mb` field at an object `b` on the minor heap.

What happens next? If the minor heap collection worked exactly as I described it in part 4, then the outcome wouldn't be good. `b` isn't pointed at by any local root, so it would be "unreachable" and would disappear, leaving a dangling pointer.

One solution would be to check the major heap, but that would be massively time-consuming: minor collections are supposed to be very quick.

What OCaml does instead is to have a separate “refs” list. This contains a list of pointers that point from the major heap to the minor heap. During a minor heap collection, the refs list is consulted for additional roots (and after the minor heap collection, the refs list can be started anew).

The refs list however has to be updated, and it gets updated potentially every time we modify a mutable field in a struct, such as `a` above. The code calls the C function `caml_modify` which both mutates the struct and decides whether this is a major->minor pointer to be added to the refs list.

Needless to say, if you use mutable fields then this is much slower than a simple assignment.

The compiler helps out somewhat. Mutable integers are OK and don’t trigger the extra call. Of course non-mutable objects don’t pay any penalty (but on the other hand, if you have to “modify” one, then that entails a complete copy). You can also mutate fields yourself, eg. from C functions or using Obj, provided you can guarantee that this won’t generate a pointer between the major and minor heaps.

Slicing and dicing

The OCaml garbage collector doesn’t collect the major heap in one go. It spreads the work over small **slices**, and slices are grouped into whole **phases** of work.

A slice is just a defined amount of work, normally calculated automatically by the garbage collector based on how fast it thinks the program is allocating (and other factors).

The phases are mark and sweep, and some additional subphases dealing with weak pointers and finalization.

Finally there is a compaction phase which is triggered when there is no other work to do and the estimate of free space in the heap has reached some threshold.

There is a penalty to compacting the heap. If you don’t want to pay that, you can disable it completely using a tunable in the Gc module. Alternatively you can schedule when to compact the heap — while waiting for a keypress or between frames in a live simulation.

There is also a penalty for doing a slice of the major heap. This cannot be avoided so easily — for example if the minor heap is exhausted, then some activity in the major heap is unavoidable. However if you make the minor heap large enough, and schedule time to do some slice of work on the major heap, you can completely[?] control when GC work is done. You can also move large structures out of the major heap entirely (thus they won’t be scanned) using my ancient heap module.

That concludes our discussion of OCaml internals. Part 6 to follow tomorrow will supply further reading, interesting links, and a guide to the source code if you want to explore further.

4 Responses to *OCaml internals part 5: Garbage collection*

Masha
September 1, 2009 at 11:27 pm

Where is the Part 6 ?

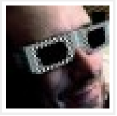
Reply

Tobu
March 31, 2011 at 12:44 am

More importantly, Ancient needs a homepage! The link from the Caml hump has bitrotted.

Reply

rich
March 31, 2011 at 9:30 pm



The source is here:
http://git.annexia.org/?p=ocaml-ancient.git;a=summary

Reply

bob
November 10, 2011 at 1:05 am

where is part 6? thanks

Reply