Memory management

OCaml PRO – Grégoire Henry

March 9-13 2015

Outline

Uniform data representation
Garbage collection

Uniform data representation

Memory graph traversal

Expected properties for a Garbage Collector (GC):

- deallocate unused heap values as soon as possible;
- do not deallocate values required by further program execution;
- a fast allocation mechanism;
- do not block the program too long while collecting.

The GC traverses the memory graph to identify:

alive values that may be accessed from the roots (globals, stack) and potentially used later.

dead values : all others values, a subset of unused values.

A GC is:

precise if the memory graph can be traversed without ambiguity.

conservative (ambiguous), if it has to over-estimate the set of living value.

$\Rightarrow$ the GC of OCaml is precise.

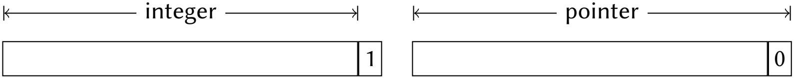
Uniform data representation

- every value is represented with a single word;
- somme immediate values fit in a word (int, char);
- bigger values are heap-allocated (a.k.a. boxed values).

This allows to compile parametric polymorphism with code sharing
There is no specialisation by default.

An OCaml value is either:

- a word-aligned pointer (Least significant bit: 0), or
- an even integer (Least significant bit: 1).



All pointers should target the OCaml heap.
Out of heap pointers are not valid immediate values, and should be properly boxed (since OCaml 4.02).

All pointers should target a block, starting with a single word header containing:

- a tag (8 bits);
- GC bits, a.k.a. a color (2 bits);
- the block size in words (22 or 54 bits).

tag	color	size	block contents

Maximum size of a block: 2^{22}words $\approx$ 16MB (32 bits), 2^{54}words $\approx$ a lot (64 bits)

Uniform data representation

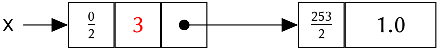
The block header tag is used by the GC to discriminate:

- Scannable block: every word in the block is an OCaml value.
  0-245 Generic block (sum type)
  246 Non-evaluated lazy value
  247 Closure
  248 Object
  249 Infix (mutually recursive closure)
  250 Forward (evaluated lazy value, internal usage)
- Non-scannable block: they must not contain pointers to the OCaml heap.
  251 Abstract value (may contains out of heap pointers)
  252 String
  253 Double
  254 Double array
  255 Custom block (may contains out of heap pointers)

This is enough information to precisely traverse
the memory graph from its roots: global value, stack frame, ...

int an immediate value
char an immediate value (8 significant bits)
bool an immediate value (only 1 significant bit!)
float[a] a boxed double (3 words)
int32[a], **int64**[a], **nativeint**[a] boxed values (custom block: 4, 6 or 4/6 words)
array a block: tag 0, $n+1$ words (when `Array.length = n`)
float array[a] a block: $2*n+1$ words (when `Array.length = n`)
tuples/record types a block: tag 0, $n+1$ words (when n is the number of fields)

```
type t = { a: int; b: float; }
let x = { a = 1; b = 1.0; }
type t2 = { a: bool; b: float; }
let x2 = { a = true; b = 1.0; }
```



[a]Compiler contains some optimisation to locally avoid boxing/unboxing.

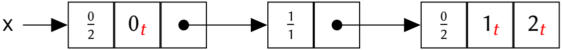
sum types

constant constructor an immediate value *i*, when the constructor is the *i*-th constant constructor in the type declaration;

non-constant constructor a block

- tag *i*, when the constructor is the *i*-th non-constant constructor in the type declaration;
- size = the number of parameter.

```
type t = A of t * t | B | C of (t * t) | D | E
let x = A (B, C (D, E))
```



string/bytes

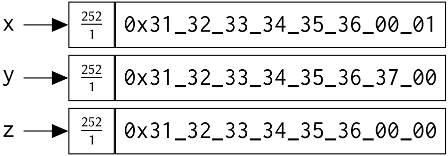
- a block of tag 252.
- size, when `String.length s = n`:

$$1+\frac{n+1}{8}$$

- the last byte of the block contains:

$$8*size-(n+1)$$

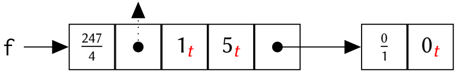
```
let x = "123456"      (* String.length x = 6 *)
let y = "1234567"     (* String.length y = 7 *)
let z = "123456\000"  (* String.length z = 7 *)
```



function (intuitively)

- a block of tag 247
- first field: code pointer
- second field: arity
- remaining fields: the captured environment (no globals)

```
let x = 3
let make y =
  let cpt = ref 0 in
  fun () -> cpt := !cpt + x + y; !cpt
let f = make 5
```



partial application a closure containing the original closure and the applied arguments

mutually recursive function a shared closure for all the functions

module a block

functor a closure

How many heap allocated words?

```
let x = [1;2;3]
let y = [1.;2.;3.]
let z = ["1";"2";"3"]

let a = [|1;2;3|]
let b = [|1.;2.;3.|]
let c = [|"1";"2";"3"|]

let t = (1, 2., '3')

let rec l = 1 :: 2 :: 3 :: l
```

Value introspection: the non-documented module `Obj`.

```
(** Not for the casual user. *)

type t

val repr : 'a -> t

val is_block : t -> bool
val is_int : t -> bool
val tag : t -> int
val size : t -> int
val field : t -> int -> t
val double_field : t -> int -> float

val new_block : int -> int -> t
val set_field : t -> int -> t -> unit
val set_double_field : t -> int -> float -> unit
```

Garbage collection

Generational GC

OCaml has a generational GC:

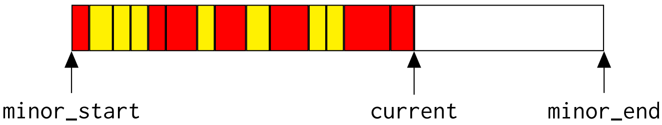
- First generation (a.k.a. **minor heap**): stop-and-copy
- Second generation (a.k.a. **major heap**):
  - incremental mark-and-sweep (most of the time)
  - stop-and-copy (when a compaction is required)

The GC is tailored for functional programming and symbolic processing :

- a lot of short-lived value;
- a lot of small values;
- few mutations.

Minor heap

Immutable blocks smaller than a constant (currently 256) are allocated on the minor heap. Allocating in the minor heap is as fast as allocation in the stack with C++!



- Allocation is simply incrementing `current`.
- When the minor heap is full, live blocks ( )
- Dead blocks ( ) "disappears" when `current` is reverted to `minor_start`.

Major heap

Mutable blocks and huge blocks are allocated in the major heap.

- Allocation by traversing the (lonely) free-list. Two strategies:
  Next-fit start the traversal from the last allocated value (fast allocation);
  First-fit start the traversal from the beginning of the list (prevents unbounded fragmentation).
- Three phases:
  Idle Do nothing!
  Marking Traverse the graph to mark live blocks.
  Sweeping Linearly scan the heap to free non-live blocks.
- Incremental: after a minor collection, execute a small part of the current phase.
- Compaction: when the fragmentation is too important (may be disabled).

Write barrier

How to handle pointer between generations ?

- Pointer from major heap to minor heap are required to detect living blocks in the minor heap.
  - traversing the whole major heap for a minor collection would be too costly;
  - they may only appear while mutating an "old" object from the major heap;

  The GC keep a list of such pointers: at every mutation in the major heap should test the "age" of the written value (still a little bit costly).

  The reverse list, is also the required to update the pointer when a block is copied from the minor heap to the major heap.

- Pointer from minor heap to major heap are "removed" while copying blocks from the minor heap to the major heap.

Tweaking & introspection

The GC has some parameters that may be tuned:

minor_heap_size current default: 256k
major_heap_increment in percent, default: 15%
space_overhead major GC eagerness
max_overhead ratio waste/live that trigger compaction, default: 500%
stack_limit default: 1M (bytecode only)
allocation_policy default: next fit
verbose

Modified by settings the OCAMLRUNPARAM environment variable, or dynamically by calling the function `Gc.set`.

```
Gc.set
  { Gc.get () with
    Gc.minor_heap_size = 1024 * 1024;
    Gc.max_overhead = 1_000_000; (* disable compaction *)
    Gc.allocation_policy = 1; (* First-fit *) }
```