25 Understanding the Garbage Collector

This chapter includes contributions from Stephen Weeks and Sadiq Jaffer.

We've described the runtime format of individual OCaml variables earlier, in Chapter 24 (Memory Representation of Values). When you execute your program, OCaml manages the lifecycle of these variables by regularly scanning allocated values and freeing them when they're no longer needed. This in turn means that your applications don't need to manually implement memory management, and it greatly reduces the likelihood of memory leaks creeping into your code.

The OCaml runtime is a C library that provides routines that can be called from running OCaml programs. The runtime manages a *heap*, which is a collection of memory regions that it obtains from the operating system. The runtime uses this memory to hold *heap blocks* that it fills up with OCaml values in response to allocation requests by the OCaml program.

25.1 Mark and Sweep Garbage Collection

When there isn't enough memory available to satisfy an allocation request from the pool of allocated heap blocks, the runtime system invokes the garbage collector (GC). An OCaml program can't explicitly free a value when it is done with it. Instead, the GC regularly determines which values are *live* and which values are *dead*, i.e., no longer in use. Dead values are collected and their memory made available for reuse by the application.

The GC doesn't keep constant track of values as they are allocated and used. Instead, it regularly scans them by starting from a set of *root* values that the application always has access to (such as the stack). The GC maintains a directed graph in which heap blocks are nodes, and there is an edge from heap block `b1` to heap block `b2` if some field of `b1` is a pointer to `b2`.

All blocks reachable from the roots by following edges in the graph must be retained, and unreachable blocks can be reused by the application. The algorithm used by OCaml to perform this heap traversal is commonly known as *mark and sweep* garbage collection, and we'll explain it further now.

https://doi.org/10.1017/9781009129220.029 Published online by Cambridge University Press

25.2 Generational Garbage Collection

The usual OCaml programming style involves allocating many small values that are used for a short period of time and then never accessed again. OCaml takes advantage of this fact to improve performance by using a *generational* GC.

A generational GC maintains separate memory regions to hold blocks based on how long the blocks have been live. OCaml's heap is split into two such regions:

- A small, fixed-size *minor heap* where most blocks are initially allocated
- A larger, variable-size *major heap* for blocks that have been live longer

A typical functional programming style means that young blocks tend to die young and old blocks tend to stay around for longer than young ones. This is often referred to as the *generational hypothesis*.

OCaml uses different memory layouts and garbage-collection algorithms for the major and minor heaps to account for this generational difference. We'll explain how they differ in more detail next.

The Gc Module and OCAMLRUNPARAM

OCaml provides several mechanisms to query and alter the behavior of the runtime system. The `Gc` module provides this functionality from within OCaml code, and we'll frequently refer to it in the rest of the chapter. As with several other standard library modules, Core alters the `Gc` interface from the standard OCaml library. We'll assume that you've opened `Core` in our explanations.

You can also control the behavior of OCaml programs by setting the `OCAMLRUNPARAM` environment variable before launching your application. This lets you set GC parameters without recompiling, for example to benchmark the effects of different settings. The format of `OCAMLRUNPARAM` is documented in the OCaml manuala.

a `https://ocaml.org/manual/runtime.html`

25.3 The Fast Minor Heap

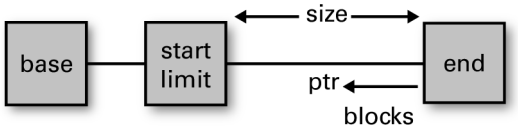
The minor heap is where most of your short-lived values are held. It consists of one contiguous chunk of virtual memory containing a sequence of OCaml blocks. If there is space, allocating a new block is a fast, constant-time operation that requires just a couple of CPU instructions.

To garbage-collect the minor heap, OCaml uses *copying collection* to move all live blocks in the minor heap to the major heap. This takes work proportional to the number of live blocks in the minor heap, which is typically small according to the generational hypothesis. In general, the garbage collector *stops the world* (that is, halts the application) while it runs, which is why it's so important that it complete quickly to let the application resume running with minimal interruption.

https://doi.org/10.1017/9781009129220.029 Published online by Cambridge University Press

25.3.1 Allocating on the Minor Heap

The minor heap is a contiguous chunk of virtual memory that is usually a few megabytes in size so that it can be scanned quickly.



The runtime stores the boundaries of the minor heap in two pointers that delimit the start and end of the heap region (`caml_young_start` and `caml_young_end`, but we will drop the `caml_young` prefix for brevity). The `base` is the memory address returned by the system `malloc`, and `start` is aligned against the next nearest word boundary from `base` to make it easier to store OCaml values.

In a fresh minor heap, the `limit` equals the `start`, and the current `ptr` will equal the `end`. `ptr` decreases as blocks are allocated until it reaches `limit`, at which point a minor garbage collection is triggered.

Allocating a block in the minor heap just requires `ptr` to be decremented by the size of the block (including the header) and a check that it's not less than `limit`. If there isn't enough space left for the block without decrementing past `limit`, a minor garbage collection is triggered. This is a very fast check (with no branching) on most CPU architectures.

Understanding Allocation

You may wonder why `limit` is required at all, since it always seems to equal `start`. It's because the easiest way for the runtime to schedule a minor heap collection is by setting `limit` to equal `end`. The next allocation will never have enough space after this is done and will always trigger a garbage collection. There are various internal reasons for such early collections, such as handling pending UNIX signals, but they don't ordinarily matter for application code.

It is possible to write loops or recurse in a way that may take a long time to do an allocation - if at all. To ensure that UNIX signals and other internal bookkeeping that require interrupting the running OCaml program still happen the compiler introduces *poll points* into generated native code.

These poll points check `ptr` against `limit` and developers should expect them to be placed at the start of every function and the back edge of loops. The compiler includes a dataflow pass that removes all but the minimum set of points necessary to ensure these checks happen in a bounded amount of time.

Setting the Size of the Minor Heap

The default minor heap size in OCaml is normally 2 MB on 64-bit platforms, but this is increased to 8 MB if you use Core (which generally prefers default settings that improve performance, but at the cost of a bigger memory profile). This setting can be

https://doi.org/10.1017/9781009129220.029 Published online by Cambridge University Press

overridden via the `s=<words>` argument to `OCAMLRUNPARAM`. You can change it after the program has started by calling the `Gc.set` function:

```
# open Core;;
# let c = Gc.get ();;
val c : Gc.Control.t =
  {Core.Gc.Control.minor_heap_size = 262144; major_heap_increment =
    15;
   space_overhead = 120; verbose = 0; max_overhead = 500;
   stack_limit = 1048576; allocation_policy = 2; window_size = 1;
   custom_major_ratio = 44; custom_minor_ratio = 100;
   custom_minor_max_size = 8192}
# Gc.tune ~minor_heap_size:(262144 * 2) ();;
- : unit = ()
```

Changing the GC size dynamically will trigger an immediate minor heap collection. Note that Core increases the default minor heap size from the standard OCaml installation quite significantly, and you'll want to reduce this if running in very memory-constrained environments.

25.4 The Long-Lived Major Heap

The major heap is where the bulk of the longer-lived and larger values in your program are stored. It consists of any number of noncontiguous chunks of virtual memory, each containing live blocks interspersed with regions of free memory. The runtime system maintains a free-list data structure that indexes all the free memory that it has allocated, and uses it to satisfy allocation requests for OCaml blocks.

The major heap is typically much larger than the minor heap and can scale to gigabytes in size. It is cleaned via a mark-and-sweep garbage collection algorithm that operates in several phases:

- The *mark* phase scans the block graph and marks all live blocks by setting a bit in the tag of the block header (known as the *color* tag).
- The *sweep* phase sequentially scans the heap chunks and identifies dead blocks that weren't marked earlier.
- The *compact* phase relocates live blocks into a freshly allocated heap to eliminate gaps in the free list. This prevents the fragmentation of heap blocks in long-running programs and normally occurs much less frequently than the mark and sweep phases.

A major garbage collection must also stop the world to ensure that blocks can be moved around without this being observed by the live application. The mark-and-sweep phases run incrementally over slices of the heap to avoid pausing the application for long periods of time, and also precede each slice with a fast minor collection. Only the compaction phase touches all the memory in one go, and is a relatively rare operation.

https://doi.org/10.1017/9781009129220.029 Published online by Cambridge University Press

25.4.1 Allocating on the Major Heap

The major heap consists of a singly linked list of contiguous memory chunks sorted in increasing order of virtual address. Each chunk is a single memory region allocated via *malloc(3)* and consists of a header and data area which contains OCaml heap chunks. A heap chunk header contains:

- The *malloc*ed virtual address of the memory region containing the chunk
- The size in bytes of the data area
- An allocation size in bytes used during heap compaction to merge small blocks to defragment the heap
- A link to the next heap chunk in the list
- A pointer to the start and end of the range of blocks that may contain unexamined fields and need to be scanned later. Only used after mark stack overflow.

Each chunk's data area starts on a page boundary, and its size is a multiple of the page size (4 KB). It contains a contiguous sequence of heap blocks that can be as small as one or two 4 KB pages, but are usually allocated in 1 MB chunks (or 512 KB on 32-bit architectures).

Controlling the Major Heap Increment

The `Gc` module uses the `major_heap_increment` value to control the major heap growth. This defines the number of words to add to the major heap per expansion and is the only memory allocation operation that the operating system observes from the OCaml runtime after initial startup (since the minor is fixed in size).

Allocating an OCaml value on the major heap first checks the free list of blocks for a suitable region to place it. If there isn't enough room on the free list, the runtime expands the major heap by allocating a fresh heap chunk that will be large enough. That chunk is then added to the free list, and the free list is checked again (and this time will definitely succeed).

Older versions of OCaml required setting a fixed number of bytes for the major heap increment. That was a value that was tricky to get right: too small of a value could lead to lots of smaller heap chunks spread across different regions of virtual memory that require more housekeeping in the OCaml runtime to keep track of them; too large of a value can waste memory for programs with small heaps.

You can use `Gc.tune` to set that value, but the values are a little counter-intuitive, for backwards-compatibility reasons. Values under 1000 are interpreted as percentages, and the default is 15%. Values 1000 and over are treated as a raw number of bytes. But most of the time, you won't set the value at all.

25.4.2 Memory Allocation Strategies

The major heap does its best to manage memory allocation as efficiently as possible and relies on heap compaction to ensure that memory stays contiguous and unfragmented.

https://doi.org/10.1017/9781009129220.029 Published online by Cambridge University Press

The default allocation policy normally works fine for most applications, but it's worth bearing in mind that there are other options, too.

The free list of blocks is always checked first when allocating a new block in the major heap. The default free list search is called *best-fit allocation*, with alternatives *next-fit* and *first-fit* algorithms also available.

Best-Fit Allocation

The best-fit allocator is a combination of two strategies. The first, size-segregated free lists, is based on the observation that nearly all major heap allocations in OCaml are small (consider list elements and tuples which are only a couple of machine words). Best fit keeps separate free lists for sizes up to and including 16 words which gives a fast path for most allocations. Allocations for these sizes can be serviced from their segregated free lists or, if they are empty, from the next size with a space.

The second strategy, for larger allocations, is the use of a specialized data structure known as a *splay tree* for the free list. This is a type of search tree that adapts to recent access patterns. For our use this means that the most commonly requested allocation sizes are the quickest to access.

Small allocations, when there are no larger sizes available in the segregated free lists, and large allocations greater than sixteen words are serviced from the main free list. The free list is queried for the smallest block that is at least as large as the allocation requested.

Best-fit allocation is the default allocation mechanism. It represents a good trade-off between the allocation cost (in terms of CPU work) and heap fragmentation.

Next-Fit Allocation

Next-fit allocation keeps a pointer to the block in the free list that was most recently used to satisfy a request. When a new request comes in, the allocator searches from the next block to the end of the free list, and then from the beginning of the free list up to that block.

Next-fit allocation is quite a cheap allocation mechanism, since the same heap chunk can be reused across allocation requests until it runs out. This in turn means that there is good memory locality to use CPU caches better. The big downside of next-fit is that since most allocations are small, large blocks at the start of the free list become heavily fragmented.

First-Fit Allocation

If your program allocates values of many varied sizes, you may sometimes find that your free list becomes fragmented. In this situation, the GC is forced to perform an expensive compaction despite there being free chunks, since none of the chunks alone are big enough to satisfy the request.

First-fit allocation focuses on reducing memory fragmentation (and hence the number of compactions), but at the expense of slower memory allocation. Every allocation scans the free list from the beginning for a suitable free chunk, instead of reusing the most recent heap chunk as the next-fit allocator does.

For some workloads that need more real-time behavior under load, the reduction in the frequency of heap compaction will outweigh the extra allocation cost.

Controlling the Heap Allocation Policy

You can set the heap allocation policy by calling `Gc.tune`:

```
# Gc.tune ~allocation_policy:First_fit ();;
- : unit = ()
```

The same behavior can be controlled via an environment variable by setting `OCAMLRUNPARAM` to `a=0` for next-fit, `a=1` for first-fit, or `a=2` for best-fit.

25.4.3 Marking and Scanning the Heap

The marking process can take a long time to run over the complete major heap and has to pause the main application while it's active. It therefore runs incrementally by marking the heap in *slices*. Each value in the heap has a 2-bit *color* field in its header that is used to store information about whether the value has been marked so that the GC can resume easily between slices.

- Blue: On the free list and not currently in use
- White (during marking): Not reached yet, but possibly reachable
- White (during sweeping): Unreachable and can be freed
- Black: Reachable, and its fields have been scanned

The color tags in the value headers store most of the state of the marking process, allowing it to be paused and resumed later. On allocation, all heap values are initially given the color white indicating they are possibly reachable but haven't been scanned yet. The GC and application alternate between marking a slice of the major heap and actually getting on with executing the program logic. The OCaml runtime calculates a sensible value for the size of each major heap slice based on the rate of allocation and available memory.

The marking process starts with a set of *root* values that are always live (such as the application stack and globals). These root values have their color set to black and are pushed on to a specialized data structure known as the *mark* stack. Marking proceeds by popping a value from the stack and examining its fields. Any fields containing white-colored blocks are changed to black and pushed onto the mark stack.

This process is repeated until the mark stack is empty and there are no further values to mark. There's one important edge case in this process, though. The mark stack can only grow to a certain size, after which the GC can no longer recurse into intermediate values since it has nowhere to store them while it follows their fields. This is known as mark stack *overflow* and a process called *pruning* begins. Pruning empties the mark stack entirely, summarizing the addresses of each block as start and end ranges in each heap chunk header.

Later in the marking process when the mark stack is empty it is replenished by *redarkening* the heap. This starts at the first heap chunk (by address) that has blocks

https://doi.org/10.1017/9781009129220.029 Published online by Cambridge University Press

needing redarkening (i.e were removed from the mark stack during a prune) and entries from the redarkening range are added to the mark stack until it is a quarter full. The emptying and replenishing cycle continues until there are no heap chunks with ranges left to redarken.

Controlling Major Heap Collections

You can trigger a single slice of the major GC via the `major_slice` call. This performs a minor collection first, and then a single slice. The size of the slice is normally automatically computed by the GC to an appropriate value and returns this value so that you can modify it in future calls if necessary:

```
# Gc.major_slice 0;;
- : int = 0
# Gc.full_major ();;
- : unit = ()
```

The `space_overhead` setting controls how aggressive the GC is about setting the slice size to a large size. This represents the proportion of memory used for live data that will be "wasted" because the GC doesn't immediately collect unreachable blocks. Core defaults this to `100` to reflect a typical system that isn't overly memory-constrained. Set this even higher if you have lots of memory, or lower to cause the GC to work harder and collect blocks faster at the expense of using more CPU time.

25.4.4 Heap Compaction

After a certain number of major GC cycles have completed, the heap may begin to be fragmented due to values being deallocated out of order from how they were allocated. This makes it harder for the GC to find a contiguous block of memory for fresh allocations, which in turn would require the heap to be grown unnecessarily.

The heap compaction cycle avoids this by relocating all the values in the major heap into a fresh heap that places them all contiguously in memory again. A naive implementation of the algorithm would require extra memory to store the new heap, but OCaml performs the compaction in place within the existing heap.

Controlling Frequency of Compactions

The `max_overhead` setting in the `Gc` module defines the connection between free memory and allocated memory after which compaction is activated.

A value of `0` triggers a compaction after every major garbage collection cycle, whereas the maximum value of `1000000` disables heap compaction completely. The default settings should be fine unless you have unusual allocation patterns that are causing a higher-than-usual rate of compactions:

```
# Gc.tune ~max_overhead:0 ();;
- : unit = ()
```

https://doi.org/10.1017/9781009129220.029 Published online by Cambridge University Press

25.4.5 Intergenerational Pointers

One complexity of generational collection arises from the fact that minor heap sweeps are much more frequent than major heap collections. In order to know which blocks in the minor heap are live, the collector must track which minor-heap blocks are directly pointed to by major-heap blocks. Without this information, each minor collection would also require scanning the much larger major heap.

OCaml maintains a set of such *intergenerational pointers* to avoid this dependency between a major and minor heap collection. The compiler introduces a write barrier to update this so-called *remembered set* whenever a major-heap block is modified to point at a minor-heap block.

The Mutable Write Barrier

The write barrier can have profound implications for the structure of your code. It's one of the reasons using immutable data structures and allocating a fresh copy with changes can sometimes be faster than mutating a record in place.

The OCaml compiler keeps track of any mutable types and adds a call to the runtime `caml_modify` function before making the change. This checks the location of the target write and the value it's being changed to, and ensures that the remembered set is consistent. Although the write barrier is reasonably efficient, it can sometimes be slower than simply allocating a fresh value on the fast minor heap and doing some extra minor collections.

Let's see this for ourselves with a simple test program. You'll need to install the Core benchmarking suite via `opam install core_bench` before you compile this code:

```
open Core
open Core_bench

module Mutable = struct
  type t =
    { mutable iters : int
    ; mutable count : float
    }

  let rec test t =
    if t.iters = 0
    then ()
    else (
      t.iters <- t.iters - 1;
      t.count <- t.count +. 1.0;
      test t)
end

module Immutable = struct
  type t =
    { iters : int
    ; count : float
    }

  let rec test t =
    if t.iters = 0
```

https://doi.org/10.1017/9781009129220.029 Published online by Cambridge University Press

```
    then ()
    else test { iters = t.iters - 1; count = t.count +. 1.0 }
end

let () =
  let iters = 1_000_000 in
  let count = 0.0 in
  let tests =
    [ Bench.Test.create ~name:"mutable" (fun () ->
          Mutable.test { iters; count })
    ; Bench.Test.create ~name:"immutable" (fun () ->
          Immutable.test { iters; count })
    ]
  in
  Bench.make_command tests |> Command.run
```

This program defines a type `t1` that is mutable and `t2` that is immutable. The benchmark loop iterates over both fields and increments a counter. Compile and execute this with some extra options to show the amount of garbage collection occurring:

```
$ dune exec -- ./barrier_bench.exe -ascii alloc -quota 1
Estimated testing time 2s (2 benchmarks x 1s). Change using '-quota'.

  Name         Time/Run   mWd/Run   mjWd/Run   Prom/Run   Percentage
 ------------ ---------- --------- ---------- ---------- ------------
  mutable        5.06ms    2.00Mw     20.61w     20.61w      100.00%
  immutable      3.95ms    5.00Mw     95.64w     95.64w       77.98%
```

There is a space/time trade-off here. The mutable version takes longer to complete than the immutable one but allocates many fewer minor-heap words than the immutable version. Minor allocation in OCaml is very fast, and so it is often better to use immutable data structures in preference to the more conventional mutable versions. On the other hand, if you only rarely mutate a value, it can be faster to take the write-barrier hit and not allocate at all.

The only way to know for sure is to benchmark your program under real-world scenarios using `Core_bench` and experiment with the trade-offs. The command-line benchmark binaries have a number of useful options that affect garbage collection behavior and the output format:

```
$ dune exec -- ./barrier_bench.exe -help
Benchmark for mutable, immutable

  barrier_bench.exe [COLUMN ...]

Columns that can be specified are:
    time       - Number of nano secs taken.
    cycles     - Number of CPU cycles (RDTSC) taken.
    alloc      - Allocation of major, minor and promoted words.
    gc         - Show major and minor collections per 1000 runs.
    percentage - Relative execution time as a percentage.
    speedup    - Relative execution cost as a speedup.
    samples    - Number of samples collected for profiling.
...
```

https://doi.org/10.1017/9781009129220.029 Published online by Cambridge University Press

25.5 Attaching Finalizer Functions to Values

OCaml's automatic memory management guarantees that a value will eventually be freed when it's no longer in use, either via the GC sweeping it or the program terminating. It's sometimes useful to run extra code just before a value is freed by the GC, for example, to check that a file descriptor has been closed, or that a log message is recorded.

What Values Can Be Finalized?

Various values cannot have finalizers attached since they aren't heap-allocated. Some examples of values that are not heap-allocated are integers, constant constructors, Booleans, the empty array, the empty list, and the unit value. The exact list of what is heap-allocated or not is implementation-dependent, which is why Core provides the `Heap_block` module to explicitly check before attaching the finalizer.

Some constant values can be heap-allocated but never deallocated during the lifetime of the program, for example, a list of integer constants. `Heap_block` explicitly checks to see if the value is in the major or minor heap, and rejects most constant values. Compiler optimizations may also duplicate some immutable values such as floating-point values in arrays. These may be finalized while another duplicate copy is being used by the program.

Core provides a `Heap_block` module that dynamically checks if a given value is suitable for finalizing. Core keeps the functions for registering finalizers in the `Core.Gc.Expert` module. Finalizers can run at any time in any thread, so they can be pretty hard to reason about in multi-threaded contexts. Async, which we discussed in Chapter 17 (Concurrent Programming with Async), shadows the `Gc` module with its own module that contains a function, `Gc.add_finalizer`, which is concurrency-safe. In particular, finalizers are scheduled in their own Async job, and care is taken by Async to capture exceptions and raise them to the appropriate monitor for error-handling.

Let's explore this with a small example that finalizes values of different types, all of which are heap-allocated.

```
open Core
open Async

let attach_finalizer n v =
  match Heap_block.create v with
  | None -> printf "%20s: FAIL\n%!" n
  | Some hb ->
    let final _ = printf "%20s: OK\n%!" n in
    Gc.add_finalizer hb final

type t = { foo : bool }

let main () =
  attach_finalizer "allocated variant" (`Foo (Random.bool ()));
  attach_finalizer "allocated string" (Bytes.create 4);
  attach_finalizer "allocated record" { foo = (Random.bool ()) };
```

https://doi.org/10.1017/9781009129220.029 Published online by Cambridge University Press

```
  Gc.compact ();
  return ()

let () =
  Command.async
    ~summary:"Testing finalizers"
    (Command.Param.return main)
  |> Command.run
```

Building and running this should show the following output:

```
$ dune exec -- ./finalizer.exe
    allocated record: OK
    allocated string: OK
   allocated variant: OK
```

The GC calls the finalization functions in the order of the deallocation. If several values become unreachable during the same GC cycle, the finalization functions will be called in the reverse order of the corresponding calls to `add_finalizer`. Each call to `add_finalizer` adds to the set of functions, which are run when the value becomes unreachable. You can have many finalizers all pointing to the same heap block if you wish.

After a garbage collection determines that a heap block `b` is unreachable, it removes from the set of finalizers all the functions associated with `b`, and serially applies each of those functions to `b`. Thus, every finalizer function attached to `b` will run at most once. However, program termination will not cause all the finalizers to be run before the runtime exits.

The finalizer can use all features of OCaml, including assignments that make the value reachable again and thus prevent it from being garbage-collected. It can also loop forever, which will cause other finalizers to be interleaved with it.

https://doi.org/10.1017/9781009129220.029 Published online by Cambridge University Press