Memory Management and Garbage Collection

In this section we discuss the general concepts of memory management and garbage collection.

Memory Management

One key characteristic of our store-based interpreter so far is that the store (heap) keeps growing in size, never releasing unnecessary memory locations. This will cause us to run out of memory sooner or later. We need a mechanism by which we can release no longer needed "locations". In other words we need a way to "free up" memory.

Freeing up memory comes in two pieces: The one is actually removing unneeded entries from the store. The other is to reuse memory locations that are no longer in use.

There are broadly speaking two kinds of approaches: manual memory management and automatic memory management, affectionately called garbage collection.

In either case, the two main tools for the task are some forms of malloc and free. malloc allocates a given amount of memory, while free deallocates the memory pointed to by a reference. The difference between the two systems is whether the programmer uses those or whether the system uses those and does not expose them to the programmer.

In programming languages like C with manual memory management the programmer is responsible for calling malloc and free.

malloc and free

Let us discuss what needs to happen for memory allocation and freeing.

- Every time malloc is called, a specific amount of memory needs to be reserved, and a pointed to that memory returned.
- Every time free is called, the specific chunk of memory indicated by the pointer is returned.
- Since the allocation and deallocation of chunks does not have to follow a specific order, we encounter the problem of "fragmentation": As long as we allocate memory we can keep doing so contiguously, but when we have to free some memory we may create a "gap" somewhere earlier. It will be desirable to fill that gap at a later time. Often we end up with numerous such gaps, and have to decide which one to reuse at any given time.
- If we think of our simplistic "locations" implementation, this means that in addition to the counter we keep incrementing we also keep a list of "unused locations", and new−loc can use one of these locations rather than incrementing the counter.

In practice we actually maintain multiple lists of freed locations, each with different standard sizes, so that when a new memory location is needed we can choose one of suitable size (and if none is available we can split a larger location in two smaller ones.

An interesting question arises as to how we keep track of what memory locations are available, literally *where* we store that information? It turns out that we do it in the freed locations themselves! We turn them into a linked list, each location containing as its "value" a pointer to the next location. When a location is freed, its contents are replaced with a link to the first free location of that size, and the newly freed location becomes the start of the linked list. When a location is to be allocated, it is taken from the front of the list.

A simplistic implementation of all this, using our locations infrastructure, could be as follows:

```
(define free-list null)
(define (free loc)
  (set! free-list (cons loc free-list)))
(define (malloc)
  (if (null? free-list)
      (new-loc)
      (let ([loc (car free-list)])
        (begin (set! free-list (cdr free-list))
               loc))))
```

Reference counts

Manual memory allocation puts an undue burden on the programmer. A common question that programmers need to think about in C and C++ is the question of whether *they* are the ones responsible for freeing a particular memory location, or whether some library or framework is responsible instead. This becomes part of the documentation of that library/framework, but it is still a considerable amount of mental overhead.

One approach to help with this is *reference counting*. The idea is that each reference/location has a count associated with it. Each time that a new link to that location is formed, the count is incremented. Each time a link is broken, the count is decremented. If the count reaches 0, that means there are no links left to that memory location and that therefore the location can be freed.

Let us imagine for example the following OCAML code snippet:

```
let r1 = ref 0 in
let r2 = r1
in r2
```

- The first line creates a new location with the value stored being 0, and `r1` is a reference to that location. We therefore would start with a "reference count" of 1.
- When the *aliasing* `r2 = r1` occurs, a second reference to the same location is formed, and the reference count would increment to 2.

- When the second let statement ends, it returns the reference r2. At this point r1 goes out of scope, so the reference count would drop back down to 1.
- The reference count will stay at 1 as long as r2 is still in scope and hasn't been bound to something else.

One tricky bit that reference counting isn't handling very well is cycles and self references. For instance consider the following code:

```
(let ([b (box 0)]
      [c (box b)])
  (begin (set-box! b c)   ;; boxes b and c refer to each other
         10))
```

Here the box b contains a link to the box c and vice versa. Now let us think of the reference count. Box b is created and contains 0 to begin with. The corresponding location has a reference count of 1. Similarly box c is created and the corresponding location has a reference count of 1. The value contained in this box c is a reference to the location stored in box b, so that location must now have a reference count of 2, as we just established a new reference to it. Lastly, we put the value of c in box b, thus incrementing the reference count of the location in box b to 2.

Then we return. At this point the b and c variables go out of scope, hence the corresponding reference counts are decremented by 1. But this still leaves the reference counts at 1, non-zero, because the two locations still have references to each other! What is worse, we no longer have access to them and have no way to influence them. So this is a cycle in memory that will never be cleared by reference counting.

The inability to recover from reference cycles is one of the main drawbacks of the reference-counting approach.

Another somewhat related approach, popular with C++, is called RAII[1]. You can learn a lot about it, and its benefits and limitations, in many online resources.

Garbage Collection

Ideal programmers would do just fine with manual memory management. But ideal programmers, that never make mistakes, are rare. Also the mental overhead required to maintain proper memory management is unnecessary. Many systems therefore, including Java as well as most functional programming languages, have transitioned to various automated memory management systems, affectionately called "garbage collection". We will discuss some of these systems in this section.

The general idea in garbage collection is simple:

- The program allocates memory as needed, and makes no efforts at freeing any memory.
- At a certain time, the normal program interaction is interrupted, and the "garbage collector" program runs.

[1] https://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

- The garbage collector determines which parts of the memory are still needed, and frees the rest.
- The program resumes.

In some instances, the garbage collector can run on a separate thread, without interfering with the normal program execution, but these cases are the exception. In most languages, normal program execution must stall while the garbage collection process is underway.

This is a main drawback of garbage-collected systems: The program execution stops at unpredictable times, and for an unpredictable amount of time. In certain domains with real-time constraints[2], this is simply not acceptable.

Finding non-free locations

The first difficulty with garbage collection is how to determine which memory is free and which memory is not free. We have here a similar classification as for type-checking:

- A *sound* memory management system never frees up a location that will be referenced in the future.
- A *complete* memory management system frees all locations that will never be referenced.

As before, we have a familiar problem: It is not possible to have an automated memory management system that is guaranteed to terminate, and that is both sound and complete. And as before, soundness is much more important than completeness: An unsound system is likely to corrupt a memory location that will be used, and when that location is used then unpredictable behavior will happen. In other words the system will on occasion choose not to free up a memory location that ends up never being used, because it could not determine that it will not end up being used. A garbage collection system is therefore very *conservative*.

The main issue is how the system determines what memory locations may be used and what memory locations are definitely not going to be used. So we want to determine which memory locations are *reachable*. The main idea is the following:

- There are two main starting points: A "global" environment that any part of the program has access to, and also all the environments that are in the evaluation stack. This deserves some explanation: Environments change when we evaluate the body of a function call, but when that function call is completed we return to a previous environment. If our garbage collector is called in the middle of a function call, then it must keep in mind not just the current environment but also all environments that we might return to, corresponding to function calls that have not been completed. This means that our interpreter will need to keep a list of the evaluation environments that it may need to "return to".

[2] https://en.wikipedia.org/wiki/Real-time_computing

- The previous part gives us a starting list of reachable locations. We can “follow” those locations to find other reachable locations. There are two new kinds of locations we might run into:
- the value stored at a location might be a reference to another location. In that case this other location is reachable (and might further lead us to more reachable locations).
- the value stored at a location might be a function closure, which therefore contains an environment. Every location in that environment is also reachable, as the function might be called at some point in the future.
- There are no other reachable locations.

The function closures show us why the general problem of determining precisely which memory locations to free is undecidable. There are locations in the function closure, and they will only be used if the function is called, and possibly not even then. We cannot determine if the function will be called at some time in the future, therefore we have to be conservative about those memory locations.

The above process though describes how we find which locations are reachable. The question still remains how precisely we will “free” the rest. There are lots of different approaches we could take, and we will discuss those in the subsequent sections.

Mark and Sweep

The most straightforward algorithm is called “mark and sweep”:

- It marks all reachable locations following the outline in the previous part.
- Then it sweeps through all memory locations, and frees those that are not marked.

In terms of our implementation, we could imagine all memory locations carrying a flag, so they would be represented as a mutable cons cell where one entry is a boolean while the other entry is the integer that we increment. Then the mark and sweep algorithm simply uses the first entry to mark the locations that should not be removed, then frees all others.

The mark and sweep algorithm is probably the simplest one to implement. It has however one main drawback: Its runtime is proportional to the size of the memory that has been allocated, and not to the size of the reachable part of the memory. If most of the allocated memory is to be freed, then this algorithm is going to have to do a whole lot of work that some of the algorithms we will describe next can avoid. For instance if there have been a total of 2GB of memory allocated so far, and only 10MB are still reachable with the remaining 1990MB primed for freeing, the mark and sweep algorithm still has to scan the entire 2GB, rather than only the 10MB. This can be particularly bad for languages that create a lot of “intermediate” values, values that were used along the way but can then be released.Functional programming languages for instance belong to this category.

Copying algorithms

The copying algorithm aims to reduce the time it takes to release memory when only a small part of it is actually reachable. It works as follows:

- The memory is divided in two equal parts. One part is the “active” one, where memory is allocated. The other is the “inactive” one that stays dormant.
- When the garbage collector starts its work, and as it traverses the reachable locations, it also *copies* these locations from the active section to the inactive one, in the process also “compacting” them (no fragmentation any more). As it does so, it leaves a link at the original location that points to the new location.
- The collector needs to also update all locations to expressions and environments, and replace them with the new locations. In our current implementation this would be fairly tricky to do.
- Once the collector has traversed the reachable locations, it simply “swaps” the roles of the two parts. The inactive part becomes active and the active part becomes inactive.

This way we never have to manually go through every spot in the active part to see if it is available or not. Once we have copied all the reachable locations, *everything* in the active part is effectively free. When that part becomes active again (it will go through an inactive phase first) all these locations will be available.

There is a variation of this algorithm, often called “mark and compact”, that tries to do the copying in-place, essentially attempting to defragment the memory.

Generational Garbage Collection

Generational garbage collection algorithms are based on what is known as the *generational hypothesis*, that essentially says the following:

> The most recently created objects are those most likely to become unreachable. Objects that have survived some GC cycles are likely to remain reachable.

This is especially the case for functional programming languages, where the heavy use of higher-order functions like map and filter results in numerous ephemeral results.

This suggests a garbage collection technique that is similar to the copying algorithms, but further divides the objects in 2 or more “generations”.

- The 1st generation contains the youngest objects. It has two parts, active and inactive, and performs copying when needed. This is the part of the memory that undergoes GC most frequently.

- The 2nd generation contains older objects, those that have survived a number of GC passes. These were 1st generation objects that remained reachable after multiple “passes”, so they have been promoted, or matured, to this second generation. These are again divided into active and inactive. This part of the memory undergoes GC less frequently.
- We might have 3rd and 4th and 5th generations, and so on.

The idea is that if an object has been reachable for some time now, then we should expect it to be reachable in the future, so there’s no reason to keep checking. We check most frequently on objects that are more recent, for which we still don’t know if they’re “here to stay” or not. Older objects don’t need our attention. This way GC cycles can run a lot faster, as they only work on a smaller part of the memory.