Lecture 22: Garbage Collection

10:00 AM, Mar 16, 2018

Contents

1 What is Garbage Collection? .......................... 1
   1.1 A Concrete Example .................................. 1
   1.2 A Second Example .................................. 2
   1.3 Defining Garbage Formally ......................... 3

2 How Does Garbage Collection Work? ............... 3
   2.1 Mark-and-sweep Collection ......................... 4

3 How Much Does Mark-and-Sweep Cost? .............. 4

4 Creating More Garbage ................................. 5

Objectives

By the end of these notes, you will know:

• What garbage collection is and how it works

By the end of these notes, you will be able to:

• Organize your code to enable more garbage to be collected

1 What is Garbage Collection?

As programs run, they allocate objects on the heap. Over time, memory can fill up with objects that programs are no longer using. This becomes a problem if memory gets sufficiently full that new data cannot be created because there is no place to put it. When memory starts to get full, the system that runs programs either has to find some unused memory or halt the program.

Garbage collection is the process of finding objects (or any data, if you are in a non-OO language) that aren’t being used and freeing up the memory they occupied for allocating new objects. In most modern languages, the execution system performs garbage collection automatically, though there are still some languages in regular use (most notably C) in which programmers must remove garbage themselves.

Today, we focus on understanding, and working with, automatic garbage collection.
1.1 A Concrete Example

Assume that we have a `Book` class and a `Main` class that creates a couple of `Book` objects:

```scala
class Book (val title: String, val author: String) {}

object Main extends App {
  new Book("The Hobbit", "Tolkien")
  val b = new Book("The Cat in the Hat", "Seuss")
  println(''end of main'')
}
```

What does the heap look like just before executing the `println`? It has two `Book` objects. One of them is associated with a known name, but the other is not. If we wanted to access the fields of the “Hobbit” object, we wouldn’t have a way to do it, because there is no way to “get to” the object from within the program. That suggests that the “Hobbit” object is garbage because it isn’t associated with a known name.

1.2 A Second Example

Let’s consider a slightly more complicated example to dig deeper into the nature of garbage:

```scala
object Library {
  def orderCopy(b : Book) = {
    val testCopy = new Book(b.title, b.author)
    // point A
    new Book(b.title, b.author)
  }
}

object Main extends App {
  val b1 = new Book("A Brief History of Time", "Hawking")
  val holdings = List(b1,
    Library.orderCopy(b1),
    Library.orderCopy(new Book("The Martian", "Weir")))
    // point B
  println(holdings)
}
```

Take a moment and draw the heap and stack at the first time execution reaches “Point A”. What is garbage at this point? Nothing, because both `Book` objects are associated with names. Notice that one object has two names (`b1` from `Main` and `b` from `orderCopy`).

Now, update the heap and stack to how they are when execution has finished the first call to `orderCopy` (while constructing the list in `Main`): there are more objects now, but fewer names:
Finally, what does memory look like when we reach “Point B”? There are even more objects, and a list that refers to some objects. The ones with green circles are reachable through names; the ones without green circles are garbage.

This example illustrates ways in which garbage arises in programs. At point A, execution is inside the `orderCopy` function, so the stack (i.e., the known names space) contains names `b` and `testCopy`. At point B, the function has finished executing, so neither of those names is still active (on the stack).

### 1.3 Defining Garbage Formally

With these two examples in mind, we can now define when a piece of data is considered garbage:

> An object or datum on the heap is garbage if there is no way to reach it from the stack.\(^1\)

Our examples have illustrated two concrete ways that data become garbage:

- they were never given names or stored in other named data within a program
- they were referenced solely inside a function which has finished executing

**Question to think about:** Our second example calls `orderCopy` twice. Does that mean that the `testCopy` object is only garbage after the second call to `testCopy`?

\(^1\)under the hood, there is actually one more place that can have references to heap data. Computers also store some data in quick-access locations called *registers*; you’ll learn about those if you take cs33.
2 How Does Garbage Collection Work?

We now know what garbage is, and we have a sense of how to identify it (follow references from the stack). But how does a language implementation track garbage, and more interestingly, how does it “take out the trash”, as it were? There are several different algorithms for taking out the trash: we’ll look at one (called mark and sweep), to give you a sense of what is involved.

2.1 Mark-and-sweep Collection

In a mark-and-sweep collector, when the runtime system is out of room for creating new objects, it temporarily stops executing the program, marks all the non-garbage (a.k.a, the live data), then walks across memory looking for garbage (non-marked data). It builds a list of memory locations that can be reused, which gets searched every time you create new data (with new, List, Map, etc).

Let’s see this in action on our second example program. Assume that execution is at line point B when garbage collection gets triggered. The garbage collector will traverse the stack (it’s just a data structure that can be iterated over, after all), marking live data. If it finds a reference to a datum that itself references data, such as a list, it marks those referenced data as well. The garbage collector will then walk all of memory (well, all that is set aside for the heap) to find the memory that can be reused.

To see this in action, watch this portion of the lecture capture video.

3 How Much Does Mark-and-Sweep Cost?

So how expensive is mark-and-sweep? Well, we have to walk the stack and touch all data that you can reach from the stack to mark it as live. The marking phase potentially reaches every item in memory, so the worst-case time of marking is proportional to the amount of memory that the runtime system uses for the stack and heap combined. The sweeping phase always visits the entire heap, so its runtime is proportional to the amount of memory used for the heap. On a modern machine, that usually amounts to several GB. What were the inventors of mark-and-sweep thinking???

Mark-and-sweep was invented in 1962. Back then, memory was on the order of a couple of hundred kilobytes (roughly the space needed to store your .java files for showdown). Walking all of memory isn’t that bad when you don’t have a lot of memory. On today’s systems, mark-and-sweep is prohibitively expensive. Fortunately, we’ve made a lot of progress in 50+ years, and there are much faster algorithms that effectively balance space and time complexity. To learn more, take cs173 (Programming Languages), which devotes a couple of lectures to modern garbage collection algorithms.

Modern garbage collection uses sophisticated algorithms to achieve a strong balance between time and space complexity. Garbage collection is efficient and powerful. Unless you are in a highly specialized context with very critical space or time requirements, garbage collection is a good solution to managing allocation and reclamation of memory.
The Cost of Acquisition  Despite its time overhead, mark-and-sweep is also good for illustrating another very important point about garbage collection (which is also true in modern systems). We don’t only incur costs of collecting garbage – the runtime also has to spend effort finding an open memory location when you create new data. Remember that mark-and-sweep created this list of open memory locations? The runtime walks this list to find a space that is large enough for the datum you are trying to store. The upshot is that you don’t just pay to take out garbage, you also incur costs to acquire new data (a point reiterated in many self-improvement articles, but that’s another issue ...).

Summary  The core idea of garbage collection is straightforward: when an object on the heap can no longer be reached through a name (on the stack), the space for that object can be reused for something else. Garbage collection and memory allocation go hand in hand, as the allocation of space needs to know what space is free, and the garbage collector determines which spaces are free. Modern collectors do NOT walk all of memory as mark-and-sweep does. We showed that algorithm because it is a simple initial presentation of the interplay between collection and allocation though.

4  Creating More Garbage

Understanding how data becomes garbage is critical to you as a programmer, because the way you structure your code determines when a datum becomes garbage. If you are working with limited memory or large data, this can be the difference between your program finishing execution or crashing with an out-of-memory exception. So now, let’s turn to the programmer side of things.

Here is a sample program for managing catalogs of items (this is for an online store, but it could just as well be a music library, a collection of webpages, etc). This program reads a catalog in from a csv file, filters out a subset of the catalog, then performs multiple computations to analyze the subset. After filtering out the subset, the program doesn’t consult the complete catalog again.

Look at the code in figure 1. Ask yourself:

* When do each of the inventory and subset of games become garbage according to our technical definition of garbage?

* When are each of the inventory and subset of games effectively garbage relative to the computation we are trying to perform (meaning the point at which we know we won’t use it again)?

To program well against garbage collection, you want the answers to these two questions to align for large pieces of data (like the inventory). As this code is written, the inventory never becomes garbage in the eyes of Scala, even though it is effectively garbage after the games list is created. Could we have written it differently to align these two answers better?

Think back to earlier in the lecture: what are ways that programs create garbage? With our second example, we saw garbage created when we exited the orderCopy function. Can we move the inventory creation into a function, so that it becomes garbage when the function finishes?

We already create the inventory inside the readCatalog function, so simply creating data inside a function isn’t the whole picture. We have to finish using large data inside a function to get it to appear as garbage to the runtime system.
Figure 1: An initial program to analyze a subset of a catalog
// one function to perform the read and the filter
def readCatalogSet(filename : String, format : String): List[Item] = {
  val inventory = readCatalog(filename)
  inventory.filter(i => i.inFormat(format))
}

With this code organization, which runs the same computations in the same order as our original program, the inventory stops being referenced once readCatalogSet finishes. The value returned from readCatalogSet is just the filtered list, not the inventory. So inventory isn’t held onto in the main program, and hence can be reclaimed as garbage as needed once the list of games has been filtered out.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback).