Chapter 22

Automatic Memory Management

22.1 Motivation

In Section 22 at the end of running filter-pos, we got the answer 29. We had to then write the procedure location→list to extract the actual list. This printed the resulting list, but what about the entire content of the heap? Let’s take a look at it:

> HEAP
#1000 (num-empty
    num-cons
      -1
      0
    num-cons
      1
      1
    num-cons
      -5
      4
    num-cons
      -1
      7
    num-cons
      3
      10
    num-cons
      2
      13
    num-cons
      0
      16
  num-empty ;; location 22
This notation means \textit{HEAP} is a vector of 1000 locations; the `'unused at the end is the content of all the remaining locations.

We can see from this why the response is \texttt{(list 2 3 1)}. But what about the rest of the heap? There are 22 locations of data that aren’t part of the value returned by \textit{filter-pos}. If we don’t reclaim these locations, they become unavailable for future use, and will eventually starve the program of memory resources.

\textit{Whose responsibility is it to reclaim these locations?} It can’t be the responsibility of \textit{filter-pos}, which can’t know whether the procedure that called it needs the input list again or not. After all, that procedure may, in turn, pass the list to some other procedure, and so on.

Even if it the chain of responsibility is clear, memory reclamation is often frustrating because it interferes with the flow of control in the program, mixing high-level algorithmic description with low-level resource management. For instance, suppose \textit{filter-pos} knew for sure that the input list would not be used any longer. It could then attempt to reclaim the list’s elements as follows:

\begin{verbatim}
(define (filter-pos l)
  (cond
    [(empty? l) empty]
    [else
      (begin
        (reclaim-memory! (first l))
        (if (> (first l) 0)
          (cons (first l) (filter-pos (rest l)))
          (filter-pos (rest l))))])
)
\end{verbatim}

There is a subtle bug in this program, but let’s focus on a simpler problem with it: while this reclaims each first element, it doesn’t reclaim the \textit{conses} that constitute the input list. We might therefore try

\begin{verbatim}
(define (filter-pos l)
  (cond
    [(empty? l) empty]
    [else
      (begin
        (reclaim-memory! (first l))
        (reclaim-memory! (rest l))
        (if (> (first l) 0)
          (cons (first l) (filter-pos (rest l)))
          (filter-pos (rest l))))]))
\end{verbatim}
22.1. MOTIVATION

Unfortunately, this version duplicates the bug! Once we reclaim the first and rest of the list, we can no longer refer to those elements. In particular, in a concurrent system (and most software today is concurrent), the moment we reclaim the memory, another process might write into it, so if we access the memory we might get nonsensical output. And even otherwise, in general, if we reclaim and then perform a procedure call (in this case, a recursive one), when we return (as we do in the first branch, to perform the cons) that heap location may have since been overridden with other values. So this is a problem even in the absence of concurrency. We must therefore instead write

\[
\text{(define \(\text{filter-pos} l\))}
\text{(cond [\((\text{empty? } l) \text{ empty}\)]
\text{[else [\((\text{define result})
\text{(if (> (first l) 0)
\text{(cons (first l) (filter-pos (rest l)))
\text{(filter-pos (rest l)))}}
\text{(begin
\text{(reclaim-memory! (first l))
\text{(reclaim-memory! (rest l)) result))}))\)])]
\text{While this version is no longer susceptible to the problems we discussed earlier, it has introduced a significant new problem. Whereas earlier \(\text{filter-pos}\) was tail-recursive in cases when the list element not positive, now \(\text{filter-pos}\) is never tail recursive. In fact, the problem we see here is a common problem with loop-like programs: we must hold on to the value being passed in the recursive call so we can reclaim it after the call completes, which forces us to destroy any potential for tail-call optimizations.}
\text{In short, even when we know who is responsible for reclaiming data, we face several problems:}
\text{• The program structure may be altered significantly.}
\text{• Concurrency, or even just other function calls, can expose very subtle reclamation errors.}
\text{• Loops often lose tail-calling behavior.}
\text{• It becomes much harder to define simple abstractions. For example, we would need two versions of a \(\text{filter-pos}\) procedure, one that does and one that doesn’t reclaim its argument list. In turn, every procedure that wants to invoke \(\text{filter-pos}\) must choose which version to invoke. And so on up the abstraction hierarchy. (Can you see how the number of possible options can grow exponentially in the number of arguments?)}
\text{These problems, furthermore, assume we can even know which procedure is responsible for managing every datum, which is a very strong assumption. Sometimes, two procedures may share a common resource}
\text{\(^{1}\)Do compare it against the original version, though, and think about which you’d rather write!}
(think of the pasteboard in a typical windowing system, which is shared between multiple applications), which means it’s no single unit’s responsibility in particular.

At any rate, reasoning about these chains of ownership is hard, and making the wrong decisions leads to numerous insidious errors. Therefore, it would be better if we could make this the responsibility of the run-time system: that is, whatever is responsible for allocating memory should also be responsible for reclaiming memory when it is no longer necessary. That is, we usually prefer to program with automated memory management, colloquially referred to by the much more colorful term, garbage collection.

22.2 Truth and Provability

In the previous paragraph, we have given the garbage collector the responsibility of reclaiming memory “when it is no longer necessary”. This puts a very significant pressure on the garbage collector: the collector must know whether or not a programmer is going to use a datum again or not. In other words, garbage collection becomes an artificial intelligence problem.

This highlights a common tension that arises in computer science, and especially in programming language design: that between truth and provability. This might sound like a very profound philosophical issue—and it is—but you are already very familiar with it from math courses, where a professor asked you to prove something she knew to be true, but you were unable to construct an actual line of reasoning for it! We see this tension in several other places, too: for example, the type checker may not know whether or not a given operation will succeed, while the programmer has a complex line of reasoning that justifies it; and the optimizer in a compiler might not be able to prove that an expensive expression is equivalent to a less expensive one (you might notice that this goes back to our discussion about referential transparency).

Anyway, the garbage collector obviously cannot know a programmer’s intent, so it needs to approximate her intent as best as possible. Furthermore, this approximation must meet a few tightly intertwined properties. To understand these, let us consider a few extreme implementations of collectors.

The first collector reclaims absolutely no garbage. Obviously it runs very quickly, and it never accidentally reclaims something that it should not reclaim. However, this is obviously useless. This suggests that a collector must demonstrate utility The collector’s approximation must identify enough garbage to actually help computation continue.

Another collector avoids this problem by reclaiming all data in memory, irrespective of whether or not they are necessary for future computation. This, too, is obviously not very useful, because the computation would soon crash. Therefore, a collector must exhibit soundness The collector must never reclaim a datum that is used by a subsequent computation.

A third collector, wanting to avoid both of the above perils, halts at every datum and computes a very complex simulation of the program’s execution to determine whether or not the program will access this datum again. It has to consider all execution paths, both branches of each conditional, and so on. This, too, would be unacceptable: a collector must manifest efficiency The collector should run sufficiently quickly so that programmer do not get frustrated (and therefore turn back to manual memory management).
22.2. TRUTH AND PROVABILITY

In practice, garbage collectors reconcile these demands thanks to a very useful approximation of truth: reachability. That is, a collector begins with a set of memory locations called the root set; this typically includes all the heap references on the stack and in the current registers. From the root set, the collector sweeps the heap to determine which objects are reachable: if object \( o \) is reachable, then all objects that \( o \) refers to in its fields are also reachable—and so on, recursively. All reachable objects are called live, and survive garbage collection; the collector reclaims the storage allocated to all other objects.

With a little reflection, we realize that reachability is an excellent approximation of truth. If an object is reachable, then there is (in effect) some sequence of field dereferences and function or method invocations that can use its value. Since the programmer may have written exactly such a sequence of invocations, the collector should not reclaim the object. If, on the other hand, an object is not reachable, no sequence of dereferences and invocations can use its value. Therefore, its space can be reclaimed safely.

Reachability is, of course, not always a strong enough approximation to truth. For instance, consider the following program fragment:

```
(define v (make-vector 1000))
define k (vector-length v))
;; rest of program
```

Suppose the rest of the program never references \( v \). Then after \( k \) has been given its value the space consumed by the vector bound to \( v \) should be reclaimed; but since \( v \) is a global variable, it is always reachable, so the collector cannot reclaim it. In general, large data structures bound to global variables are invariably candidates for space leakage, which is what we call the phenomenon of a collector not reclaiming space that we know is no longer necessary. (Notice the difference between truth and provability coming into play very strongly.) Tracing space leaks is sometimes subtle, but it is often as simple as looking at values bound to global and static variables and, when those values are no longer necessary, mutating the variable to a value like \( \text{null} \) (in Java) or \( \text{void} \) (in Scheme).

Notice, by the way, the asymmetry in our justification for why tracing is a reasonable approximation to truth. Unreachable objects will not be used so they can always be reclaimed safely, whereas reachable objects may be used again so we must allow them to persist. In fact, a collector can sometimes reason about these “maybe” cases. For instance, consider the following program:

```
(local ([define v (make-vector 1000)]
       [define k (vector-length v)])
     \ldots)
```

Now suppose the body of this expression doesn’t reference \( v \). Because \( v \) is not global, as soon as the value of \( k \) has been computed, the compiler can insert code that effectively sets the value of \( v \) to a null or void value. This makes the vector formerly bound to \( v \) a candidate for reclamation immediately, instead of waiting until the end of the \( \text{local} \). Many compilers for languages that employ garbage collection do in fact perform such “safe for space” optimizations.

---

2 This assumes that the rest of the program text is known. Modern languages support features such as dynamic loading, which is the ability to extend the program during its execution.

3 This claim makes a certain important assumption about the underlying programming language that is not always valid: it applies to languages like Java and Scheme but not to C and C++. Do you see it?