Mark-and-Sweep Garbage Collection

sk and dbtucker

November 17, 2000

We have been discussing how to manage memory through two different data structures: the stack and the heap. Take a quick look at yesterday's handout. How much memory were we using when the program finished? We certainly couldn't have been using any stack space, since the stack represents the computation left to be done. However, we were using 27 cells of memory in the heap. How many of these cells did we really need? We need only 9 of them—3 cells for each *cons* pair in the answer.

Today we'll study how to reclaim unneeded memory cells—a process called *garbage collection*. Garbage collection is a necessary part of the run-time system for which a compiler generates code—modulo some assumptions about the platform. You must also provide a GC when writing an interpreter in a language that doens't have one (for example, when writing a Scheme interpreter in C).

Look at today's handout. On the left-hand side of page 1, we create a small heap and mark all the cells as unused:

```
(define Heap-Page-Size 20)
(define Heap (make-vector Heap-Page-Size 'Unused))
(define Heap-Ptr 0)
```

Instead of allocating a new page when we run out of heap space, we'll declare that we're out of space:

```
(define (ensure-heap-space n)
  (when (> (+ Heap-Ptr n) Heap-Page-Size)
    (error 'ALLOCATION "OUT OF HEAP SPACE")))
```

Now when we run the program, we see that we've run out of space on the heap. Notice that the last two elements on the heap ('Cons and 8) represent two-thirds of a *cons* pair. We really shouldn't break up data in this way, so on the right-hand side of page 1 we add (*ensure-heap-space* 3) to the function *cons/prim*. This check ensures that we don't put partial *cons* pairs on the heap.

We are still running out of space, though. Instead of declaring an error and aborting, we should throw out everything in heap that we don't need. How can we tell which heap cells we will access in the future? The answer is we can't. However, we *can* prove that some pieces of memory will never be used.

The key is to find the values we have references to. If we have a reference, then it's possible to use the value in the future. However, if there are no references to the value, then there is no way of getting back to it. This property

is true in Scheme, but not in C. Since C allows you to do arbitrary pointer arithmetic, you can always access any part of the heap. Hence, there is no way to garbage collect C programs. To do GC effectively, you must get rid of pointer arithmetic.

In order to find all referenced values, we will use depth-first search (DFS). We start with the values on the stack, then mark all memory as either used or unused depending on whether its reachable. Turn to page 2 of the handout. You can see that the algorithm is implemented in two phases:

1. The *mark* function performs the DFS starting from some address of the heap. If the value in that cell is a 'Cons tag, *mark* changes the tag to 'MarkedCons, and recurs on the components of the pair. If the value is not a 'Cons tag, it leaves it unchanged. (The *mark* function needs stack space itself in order to run DFS. We'll ignore this detail in this course.)

2. The *sweep* function reclaims unused memory by traversing the entire heap and examing each value. If the value is an Immediate, it never gets garbage collected, so *sweep* leaves it unchanged. If the value is a 'MarkedCons, then it was reached during the *mark* phase, so *sweep* changes it back to a regular 'Cons. If the value is a 'Cons, it is no longer reachable, so its three cells are marked as 'Unused.

This algorithm is called *mark & sweep*, and has the following three properties:

1. It touches all of memory during the *sweep* process.

2. It handles circularity.

3. Nothing ever moves in memory.

Look at the function *alloc-bytes/gc*. Notice that it only tries to do allocation twice—once before garbage collection, and once after. Why doesn't it try to do GC again if the second allocation fails? The reason is that doing GC a second time (on the same heap) won't find any new unused cells.

There's a problem with this implementation, though. The result heap contains only one *cons* pair, and we know there must be at least three for the correct result. Where did the implementation go wrong? The answer is that the *mark* phase was not started with all the live references. The set of all pointers to start from is called the *root set*. In this example, we added all references on the stack to the root set, but we failed to include the local variables in *filter/split/Push-cons-first*.

The right solution is to use register form, which allocates all local variables on the stack. The solution presented on page 3 is a bit of a hack—we pass these local variables as extra arguments to *cons/prim*, which then passes them along to *alloc-bytes/gc*. This implementation does work though, and we see that the result heap contains the values necessary to construct the final answer.

An aside on reference counting

One student suggested that we use reference counting to implement garbage collection. The idea is that each cell keeps track of the number of references pointing to it; when that count reaches zero, the cell can be reclaimed. Here are the reasons why reference counting is terrible algorithm:

1. The well-known property is that it can't handle circular references.
2. Every time you add a new reference to a cell, you have to update its count.
3. The counters take up space.
4. It causes bad paging patterns: when the reference count reaches zero, you have to touch a page for a variable you will never use again.
5. It takes care of the easy cases, and leaves the hard cases to the user.