Warm-up Notes on Garbage Collection
CS536, Fall 2002

Kathi Fisler

October 27, 2002

Note: These notes bridge our class on compilers with Paul Wilson's excellent survey on uniprocessor garbage collection techniques (linked to the lectures page). After the bridge, these notes continue with the following sections from Paul's article: 1, 2 (except 2.5), 4.1, 4.5, 4.6 (scan), 6.1, 6.2, 7.1, 7.2, and 9. I refer to some of these subsections specifically in the notes. You are responsible for all of the sections listed above with respect to homeworks and exams.

We will take two weeks to cover this material: the first week, we will do an overview, reference counting, mark-and-sweep, and stop-and-copy collection. The second week, we will cover generational and conservative collection, as well as manual memory management.

Refining the Memory Model

Recall the *filter-pos* program that we were working on last week. If we load the list -2 3 0 -1 9 into memory and run the program, we get back 20 as the answer. What is 20? It's an address in the heap where the answer is stored (it is **not** a list!). If we print out the heap, we see that it has the following contents (formatted with line-breaks for readability):

```
heap: #('empty
        'cons 9 0
        'cons -1 1
        'cons 0 4
        'cons 3 7
        'cons -2 10
        'empty
        'cons 9 16
        'cons 3 17  [location 20 is the 'cons on this line]
        0 ...)
```

How do we find the actual output to check whether it is correct? The output starts in address 20, which is a 'cons. So the first item on the list is in address 21 and the rest is in address 22. The list must therefore contain 3 and the contents stored at address 17. Address 17 is also a cons with a first of 9 and a rest at address 16. Address 16 contains empty. So, the program produced a list containing 3 and 9 as the answer, which is correct.

You could write a simple program to print out the list starting from a given address as follows:

```
;; print-heap-list : heap-index[number] → void
;; for debugging: prints linked list in heap at given addr
(define (print-heap-list addr)
  (cond [(empty?/heap addr) (printf "()")]
        [(cons?/heap addr)
         (begin (printf "~a -> " (first/heap addr))
                (print-heap-list (rest/heap addr)))]
        [else (error 'print-heap-list
                     (format "address ~a holds non-list data" addr))]))

> (print-heap-list 20)
3 -> 9 -> ()
```

Notice how this program relies on the type tags that we put into the heap. Type tags are very useful, because they indicate what kind of data we are looking at. Assume I asked for the contents of heap-location 12:

> (*vector-ref the-heap* 12)
7

What do we know about the type of this result (assuming we didn't have the whole heap to look at?) Is it a number? Clearly. Does it *represent* a number in our program though? We don't know. Right now, we are using numbers for two things: numeric constants and addresses. Ideally, we need a way to distinguish these two uses of numbers.

One option would be to type-tag the numbers, but that will get expensive (two cells per constant instead of only 1). Instead, we'll set aside a separate area in memory for constants. We will call numeric constants *immediate* values and pointers *references*.

Look at the handout from class on converting the *filter-pos* program to use immediates. What do we see?

- A separate area *base-constant-memory*
- Helper functions for manipulating immediates and references
- Recognizers for immediates versus references (any number above 1000 is a reference; those below 1000 are immediates)

If we run *filter-pos* with this new memory model, we get address 1018 as the answer and the following constant area and heap contents:

```
constant 0 represents value empty
constant 1 represents value -2
constant 2 represents value 3
constant 3 represents value 0
constant 4 represents value -1
constant 5 represents value 9

heap: #('cons 5 0
        'cons 4 1000
        'cons 3 1003
        'cons 2 1006
        'cons 1 1009
        'cons 5 0
        'cons 2 1015
        'unused ...)
```

This heap looks rather different from the previous heap: now it has references (addresses) in all of the heap positions that should correspond to lists (the *rest* position of each trio of cells for a *cons*). A few questions:

- What happened to the 'empty markers in the old heap?

  Since 'empty is a constant, it resides in the constant area (at address 0). Anywhere the heap has a 0 in the *rest* position of a list, the reference points to the 'empty tag.

- Where did the negative numbers go?

  Same idea as we just discussed for 'empty.

- This heap is a lot shorter than 1018 (the address the program produced for the answer). Does this mean the output isn't really in the heap?

  The output is in the heap, we've just scaled the addresses so we can distinguish immediates and references. Look at the primitive *reference-location* for mapping reference addresses to actual heap addresses: it substracts 1000 (the size of the constant area) from the reference address. If we use *cons?/heap* to check whether the data at address 1018 is a list, we will actually look in index 18 of the heap vector.

- What list is sitting in address 18? It looks like `2 -> 5 -> 'empty`, but that's not the right answer. Remember that numbers less than 1000 do not represent numbers: they are pointers into the constant area. So the real list sitting in here is `3 -> 9 -> 'empty`, as desired.

We will use separate immediate and reference areas throughout the garbage collection lectures.

Monitoring Allocation Space

Assume we started with an initial heap of 32 cells and tried to run *filter-pos* on the list -1 1 8 3 -4 5 9 -2. We get the message:

```
vector-set!: index 32 out of range [0, 31] for vector:
#32(cons 8 0 cons 7 1000 ... cons 4)
```

What happened? The program ran off the end of the heap when it tried to store something in index 32 (which doesn't exist – remember, vectors are 0-based in their indexing). Even worse, look at the end of the heap: it has a 'cons and a 4: that means it managed to put part of a *cons* in memory, but not all of it. That's quite bad – if we allocate a datum, either the whole datum or none of it should go into memory.

To prevent partial allocation, we can write a helper function to check whether the heap contains enough space before we move data into the heap. The key question, though, is what should that function do if the heap is out of space? We have a few options:

- Increase the heap size, copy the old heap over to the new one (retaining indices), and set the-heap to refer to the new heap. Obviously, this approach only works if you have sufficient memory to keep growing the heap – this isn't always feasible.
- Report that the heap is full and abort the program. This is also a bad idea for the obvious reasons.
- Check to see if any memory isn't actually in use, then re-use it.

We're interested in the last option. Look once more at the heap contents at the end of running *filter-pos* on the list -2 3 0 -1 9:

```
constant 0 represents value empty
constant 1 represents value -2
constant 2 represents value 3
constant 3 represents value 0
constant 4 represents value -1
constant 5 represents value 9

heap: #('cons 5 0
        'cons 4 1000
        'cons 3 1003
        'cons 2 1006
        'cons 1 1009
        'cons 5 0
        'cons 2 1015
        'unused ...)
```

The answer starts in address 18. What is all of the data at the beginning of the heap? It was the space we used to represent the input list. But once we have the answer, we don't need the input list anymore (assuming we aren't saving it to run through another program, of course). We should be able to reuse that space.

Many languages provide operators for freeing memory for precisely this reason (*free* in C, *delete* in Pascal, etc) : when you (the programmer) are done with a piece of data, you tell the run-time system to free the memory. Languages like Scheme and Java take a different approach: rather than ask the *programmer* to decide when to free memory, they

include a special program called a *garbage collector* to find memory that isn't in use and to make it available for re-use.

Over the next two classes, you'll learn how to implement garbage collectors.

Wilson's article does a great job of arguing why garbage collection is important, so I won't reproduce his argument here. At this point, you should read Section 1 of Wilson's article.

How Not Having GC Affects Program Structure

Having read Section 1, let's make sure you understand his argument about why the lack of garbage collection affects how you write programs. Recall quicksort:

```
(define (quicksort L)
  (cond [(empty? L) empty]
        [(cons? L)
         (let ([pivot (first L)])
           (append (quicksort (get-smaller-nums pivot (rest L)))
                   (list pivot)
                   (quicksort (get-larger-nums pivot (rest L)))))]))
```

Written this way, the structure of the algorithm is very clear. Let's see what happens to the program structure if we have to issue the commands to free the memory.

After the call to *append*, it's clear that we won't use the lists of smaller and larger numbers, and they could be reclaimed. If we had to free this memory manually, how would we do it? We don't have names for those lists. If we had to manually free the memory, we'd need to introduce names for those lists, as follows:

```
(define (quicksort L)
  (cond [(empty? L) empty]
        [(cons? L)
         (let ([pivot (first L)])
           (let ([smaller-elts (get-smaller-nums pivot (rest L))]
                 [larger-elts (get-larger-nums pivot (rest L))])
             (append (quicksort smaller-elts)
                     (list pivot)
                     (quicksort larger-elts))))]))
```

Now we have the names (*smaller-elts* and *larger-elts*), but where do we free them? We can't free them until after the calls to *quicksort*. But the calls to *quicksort* are nested inside the *append*, which is the return value of the program. We need to hold onto the return value so we can free the memory:

```
(define (free x) "Warning: free is not a real Scheme operation")

(define (quicksort L)
  (cond [(empty? L) empty]
        [(cons? L)
         (let ([pivot (first L)])
           (let ([smaller-elts (get-smaller-nums pivot (rest L))]
                 [larger-elts (get-larger-nums pivot (rest L))])
             (let ([result (append (quicksort smaller-elts)
                                   (list pivot)
                                   (quicksort larger-elts))])
               (begin
                 (free smaller-elts)
                 (free larger-elts)
                 result))))]))
```

Does this take care of freeing all of the unused memory? No, because once we do the append, we don't need the results of (*quicksort smaller-elts*) and (*list pivot*) because *append* rebuilds all but the last of its argument lists. So to really free the unused memory, we'd need to write:

```
(define (free x) "Warning: free is not a real Scheme operation")

(define (quicksort L)
  (cond [(empty? L) empty]
        [(cons? L)
         (let ([pivot (first L)])
           (let ([smaller-elts (get-smaller-nums pivot (rest L))]
                 [pivot-list (list pivot)]
                 [larger-elts (get-larger-nums pivot (rest L))])
             (let ([sorted-smaller (quicksort smaller-elts)])
               (let ([result (append sorted-smaller
                                     pivot-list
                                     (quicksort larger-elts))])
                 (begin
                   (free smaller-elts)
                   (free larger-elts)
                   (free pivot-list)
                   (free sorted-smaller)
                   result))))))]))
```

Some notes:

- This code is getting both harder to read and more cluttered with run-time commands (as opposed to program logic).
- To fully reclaim memory in this case, you had to know how *append* is implemented! (If you accidentally free the sorted larger elements, you could overwrite part of your output data).
- If you accidentally free the pivot, you could also corrupt your output.
- The programmer must choose a convention: either the caller of a function frees the data or the called function (callee) frees the data. Mixed conventions can lead to memory errors.

Garbage collection frees the programmer from worrying about any of these issues.

How Data Becomes Free: A Lower-Level Look

The *quicksort* example helps give us an intuition about when memory becomes re-claimable in a program, but doesn't give us insight as to how garbage collection operates at the level of our stack and heap vector representation. If we are going to write garbage collectors, we need intuition at that level as well.

Figure 1 shows a snapshot of the stack and heap contents while running the have-eye-color program from the homework. This snapshot helps reinforce the presence of continuations in our compiler. Note that even though Mary is at the root of the tree, Mary's data is no longer accessible in the running program. At the stage shown, the program has finished processing Mary's mother's (Ann) tree, and has moved on to processing her father's (Fred) tree. Only Fred and his ancestors are still accessible in the tree, as is the partially formed output list.

Implementing Garbage Collection

The *quicksort* example is fine for building some intuition about where garbage comes from in a Scheme program, but it doesn't help us figure out how to implement a garbage collector. To implement a garbage collector, we have to be able to identify when data is garbage, and we need a method for reclaiming the space used by that data (see Wilson, Section 1.2).

```
heap: #(info 1985 blue            [0]
        info 1960 green           [3]
        info 1941 green           [6]
        unknown                   [9]
        unknown                   [10]
        person jane 6 9 10        [11]
        info 1939 blue            [16]   <-- (from stack)
        unknown                   [19]
        unknown                   [20]
        person ed 16 19 20        [21]
        person ann 3 11 21        [26]
        info 1961 blue            [31]   <-- (from stack, loc 56, =inforeg=)
        info 1930 brown           [34]   <-- (from loc 50)
        info 1908 brown           [37]   <-- (from loc 44)
        unknown                   [40]   <-- (from loc 45)
        unknown                   [41]   <-- (from loc 46)
        person hilda 37 40 41     [42]   <-- (from loc 51)
        unknown                   [47]   <-- (from loc 52)
        person rose 34 42 47      [48]   <-- (from stack, loc 57)
        unknown                   [53]   <-- (from stack, loc 58)
        person fred 31 48 53      [54]   <-- (from =treereg=)
        person mary 0 26 54       [59]
        empty                     [64]
        empty                     [65]
        empty                     [66]   <-- (from stack)
        empty                     [67]   <-- (from loc 70)
        cons ed 67                [68]   <-- (from stack)
        0 ...)                    [71]

stack: #(#(struct:bottom-rec)
         #(struct:father-cons-rec mary 0 68 blue)
         #(struct:match-rec fred 31 48 53 blue)
         #(struct:father-cons-rec ed 16 66 blue))

=treereg= is 54
=inforeg= is 31
```

Figure 1: A snapshot of the heap and the stack while running the have-eye-color program. The numbers in square brackets in the heap indicate the heap index of the first cell in the line. The arrows in the right part of the figure indicate which cells are reachable and the path by which they are reachable. The snapshot was taken when Fred was the person being processed. As expected, the program has finished processing Mary and Ann (and Ann's ancestors), so none of the info for Mary, Ann, or her ancestors is still reachable.

What is garbage? From a garbage collector's perspective, *garbage* is any data that is no longer accessible in the program. What does *accessible* mean? A datum is accessible if no chain of pointers in the running program can reach the datum. Figure 1 illustrates chains of pointers reaching data objects. The simple story is *if your program can provide a way to access a datum, you can't garbage collect it.*

Consider this statement in the context of programming in C or C++. What are its implications? In simplest terms, we can't write garbage collectors for these languages because pointer arithmetic leaves *every* location accessible in the program. We'll revisit this question next week though and see how people do try to use garbage collection with C++.

Note that garbage collection does **not** try to predict whether a value will be used again in the future. If a program has a reference to a value, the garbage collector will leave the value in the heap, even if the program never again follows that reference. In other words, garbage collection will **never** throw away data that the program might access, but it might retain some data that won't be used again. That's okay though: garbage collection is a formal analysis, not an AI problem.

Now, onto the collectors!

Reference Counting

If you've heard of only one approach to garbage collection algorithm, it's likely to be reference counting. The basic idea is that each cell maintains a count of how many references point to it, and the cell can be reclaimed when the count goes to 0. See Wilson's survey for details, advantages, and disadvantages of reference counting (Section 2.1). Let's consider some small examples. For readability, I'll do these in regular Scheme, rather than in our compiled language.

```
(let ([x1 (cons 2 (cons 3 empty))])
  (+ (let ([x2 (cons 1 x1)])
       (first x2))
     (let ([x3 (rest x1)])
       (first x3))))
```

The reference counts must change as the program executes. The following table shows the reference counts on each *cons* cell in the program, with - denoting that the *cons* hasn't yet been created:

Expression	(*cons* 3 *empty*)	(*cons* 2 (*cons* 3 *empty*))	(*cons* 1 *x1*)
(**let** ([*x1* . . .] . . .)	1	1	-
(**let** ([*x2* . . .] . . .)	1	2	1
(*first x2*)	1	2	1
(**let** ([*x3* . . .] . . .)	2	1	0
(*first x3*)	2	1	0
(+ . . .)	1	1	0
end of evaluation	0	0	0

As another example, in the snapshot from Figure 1, the reference counts on each typed cell match the number of source locations in parentheses after the arrows.

Mark and Sweep

Mark-and-sweep is the simplest real garbage collection algorithm. See Wilson, Section 2.2 for details. Figures 2 and 3 show how our heap snapshot would look after the marking and sweeping stages, respectively.

Stop-and-Copy

For this algorithm, refer to Wilson Section 2.4 and the diagrams drawn during class to illustrate this algorithm. Figure 4 shows one possible reconfigured heap after running stop-and-copy on the snapshot from Figure 1.

```
heap: #(info 1985 blue              [0]
        info 1960 green             [3]
        info 1941 green             [6]
        unknown                     [9]
        unknown                     [10]
        person jane 6 9 10          [11]
        Markedinfo 1939 blue        [16]   <-- (from stack)
        unknown                     [19]
        unknown                     [20]
        person ed 16 19 20          [21]
        person ann 3 11 21          [26]
        Markedinfo 1961 blue        [31]   <-- (from stack, loc 56, =inforeg=)
        Markedinfo 1930 brown       [34]   <-- (from loc 50)
        Markedinfo 1908 brown       [37]   <-- (from loc 44)
        Markedunknown               [40]   <-- (from loc 45)
        Markedunknown               [41]   <-- (from loc 46)
        Markedperson hilda 37 40 41 [42]   <-- (from loc 51)
        Markedunknown               [47]   <-- (from loc 52)
        Markedperson rose 34 42 47  [48]   <-- (from stack, loc 57)
        Markedunknown               [53]   <-- (from stack, loc 58)
        Markedperson fred 31 48 53  [54]   <-- (from =treereg=)
        person mary 0 26 54         [59]
        empty                       [64]
        empty                       [65]
        Markedempty                 [66]   <-- (from stack)
        Markedempty                 [67]   <-- (from loc 70)
        Markedcons ed 67            [68]   <-- (from stack)
        0 ...)                      [71]
```

Figure 2: The heap snapshot from Figure 1 after marking.

```
heap: #(unused unused unused        [0]
        unused unused unused        [3]
        unused unused unused        [6]
        unused                      [9]
        unused                      [10]
        unused unused unused        [11]
        unused unused               [14]
        info 1939 blue              [16]   <-- (from stack)
        unused                      [19]
        unused                      [20]
        unused unused unused        [21]
        unused unused               [24]
        unused unused unused        [26]
        unused unused               [29]
        info 1961 blue              [31]   <-- (from stack, loc 56, =inforeg=)
        info 1930 brown             [34]   <-- (from loc 50)
        info 1908 brown             [37]   <-- (from loc 44)
        unknown                     [40]   <-- (from loc 45)
        unknown                     [41]   <-- (from loc 46)
        person hilda 37 40 41       [42]   <-- (from loc 51)
        unknown                     [47]   <-- (from loc 52)
        person rose 34 42 47        [48]   <-- (from stack, loc 57)
        unknown                     [53]   <-- (from stack, loc 58)
        person fred 31 48 53        [54]   <-- (from =treereg=)
        unused unused unused        [59]
        unused unused               [62]
        unused                      [64]
        unused                      [65]
        empty                       [66]   <-- (from stack)
        empty                       [67]   <-- (from loc 70)
        cons ed 67                  [68]   <-- (from stack)
        0 ...)                      [71]
```

Figure 3: The heap snapshot from Figure 1 after sweeping.

```
heap: #(info 1939 blue              [0]
        empty                       [3]
        info 1961 blue              [4]
        person rose 12 15 20        [7]
        info 1930 brown             [12]
        person hilda 21 24 25       [15]
        unknown                     [20]
        info 1908 brown             [21]
        unknown                     [24]
        unknown                     [25]
        unknown                     [26]
        empty                       [27]
        cons ed 27                  [28]
        person fred 4 7 26          [31]

stack: #(#(struct:bottom-rec)
         #(struct:father-cons-rec mary 0 28 blue)
         #(struct:match-rec fred 4 7 26 blue)
         #(struct:father-cons-rec ed 0 3 blue))

=treereg= is 31
=inforeg= is 4
```

Figure 4: One possible heap snapshot from Figure 1 after copying in stop-and-copy.