MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1998

Lecture Notes – Dec. 1, 1998

STORAGE MANAGEMENT

Suppose we assume that all of the heap is linked together in one big list of cons cells that are unused and that a variable `free-list` in the global environment has the list as its value.

Then we could write `cons` as follows:

```
(define (cons x y)
  (let ((cons-cell free-list))
    (set! free-list (cdr free-list))
    (set-car! cons-cell x)
    (set-cdr! cons-cell y)
    cons-cell))
```

This is fine until we run out of unused cons cells. So let us assume that in that case we can call a procedure `gc` which finds all the unused cons cells and puts them in the `free-list` again. Then we can rewrite `cons` to be:

```
(define (cons x y)
  (if (null? free-list)
      (gc))
  (let ((cons-cell free-list))
    (set! free-list (cdr free-list))
    (set-car! cons-cell x)
    (set-cdr! cons-cell y)
    cons-cell))
```

Now how can `gc` be implemented? We will need to think about how memory is organized. We will think of it as a collection of vectors, and use two procedures `vector-ref`, and `vector-set!` to reference and modify elements of vectors.

Elements in Lisp, such as cons cells, numbers, and the empty list will be represented internally as having two parts. They have a *tag* which encodes their type and can be one of `P`, `N`, or `E` (for pair, number, and empty list). They have a *contents* which is a number which is their index into the memory vectors for pairs, the number itself for a Scheme number, and zero for the empty list. We will use `change-tag` to take some object and return a new object with the same contents, but a different tag.

Let us demand that `vector-ref` and `vector-set!` can take any object as an index, and just use the contents field to index into a vector.

Now suppose we have three vectors:

```
the-cars        a vector of all the cars of the cons cells
the-cdrs        a vector of all the cdrs of the cons cells
the-marks       a vector of single bits, one for each cons cell
```

Now we can define `set-car!` and `set-cdr!` in terms of our new primitives:

```
(define (set-car! pair value)
  (vector-set! the-cars pair value))
(define (set-cdr! pair value)
  (vector-set! the-cdrs pair value))
```

The *mark and sweep* garbage collector recursively traces down all the list structure accessible from the `root`. It then sweeps through all of memory collecting the unmarked cons cells into a free list, and at the same time zeroing all the marks for the next time around.

The algorithm can be coded as:

```
(define (gc)
  (define (mark object)
    (cond ((not (pair? object)) false)
          ((not (= 1 (vector-ref the-marks object)))
           (vector-set! the-marks object 1)
           (mark (car object))
           (mark (cdr object)))))
  (define (sweep i)
    (cond ((not (= i size))
           (cond ((= 1 (vector-ref the-marks i))
                  (vector-set! the-marks 0))    ;for next gc
                 (else (set-cdr! i free-list)   ;i is a number!!!
                       (set! free-list (change-tag i 'p))))
           (sweep (+ i 1)))))
  (mark root)
  (sweep 0))
```

For the *stop and copy* garbage collection algorithm we partition the heap into two half size consing areas, `from` space, and `to` space. We always allocate new cons cells in `from` space until we are out of memory. We then copy the good stuff over to `to` space, swap the roles of the two half spaces, and continue consing into the new `from` space.

The algorithm needs an extra legal type of pointer, a forwarding pointer, one whose tag is `F`.

The essential points of the method can be captured by:

1. set `free` and `scan` to beginning of new memory (`to` space)
2. move `root` pairs to new memory
3. adjust root pointers to point to new location
4. increment `free` as necessary
5. mark old pairs by putting a forwarding pointer to new memory
6. basic cycle:
   - trace pointers in `car` and `cdr` in cell pointed to by `scan` back to old memory
   - relocate each one: if a forwarding pointer, use forwarding address to update pointers, and if not a forwarding pointer, copy into `free` pair, then increment `free`, store a forwarding pointer, and update pointers to new pair
   - increment `scan`
7. stop when `scan` catches up to `free`

For this GC `cons` becomes:

```
(define (cons x y)
  (if (>= free fromend)
      (gc))
  (let ((cons-cell (change-tag free 'p)))
    (set! free (+ 1 free))
    (set-car! cons-cell x)
    (set-cdr! cons-cell y)
    cons-cell))
```

And we can now write out the whole garbage collector:

```
(define (gc)
  (set! free tostart)
  (set! root (forward root))
  (gccopy tostart)
  (let ((oldtostart tostart)
        (oldtoend toend))
    (set! tostart fromstart)
    (set! toend fromend)
    (set! fromstart oldstart)
    (set! fromend oldtoend)))
(define (gccopy scan)
  (cond ((/= scan free)
         (set-car! scan (forward (vector-ref the-cars scan)))
         (set-cdr! scan (forward (vector-ref the-cdrs scan)))
         (gccopy (+ scan 1)))))
(define (forward pointer)
  (cond ((pair? pointer)
         (let ((oldcar (car pointer)))
           (if (forward? oldcar)
               (change-tag oldcar 'p)
             (let ((newptr (change-tag free 'p)))
               (set-car! newptr oldcar)
               (set-cdr! newptr (cdr pointer))
               (set! free (+ 1 free))
               (set-car! pointer (change-tag newptr 'f))
               newptr))))))
```