Managing Memory

6.037 – Structure and Interpretation of Computer Programs

Mike Phillips

Massachusetts Institute of Technology

Lecture 8B

Expanding our horizons

So far, we've examined just a bit of the overall Scheme story.

math

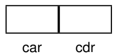
↕

electrical engineering

lambda calculus
complex data structures
primitive data structures (e.g., `cons` cells)
RAM

Implementing cons-cell memory

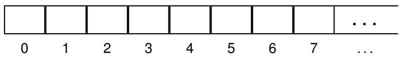
- We've been using the `cons`-cell abstraction this whole class.



- Computer memory doesn't really work like that.

Computer Memory

- Conventional memory is an array of locations, each of which has an integer address, and stores a single value.



- Addresses are sequential, so we often move around memory by adding and subtracting values from addresses.

Vectors

- We will model memory using vectors.
- Also a generally-useful data structure (similar to arrays in other languages).
- Vectors support constant-time access of an arbitrary element.

Vector Operations

- `(make-vector <size>) → <v>`
  - Returns a vector of the given size.
- `(vector-ref <v> <n>) → <elt>`
  - Return the element at index `n` of `v` (0-indexed)
- `(vector-set! <v> <n> <val>)` → *undefined*
  - Sets the element at index `n` of `v`.
- `(vector-length <v>) → <size>`

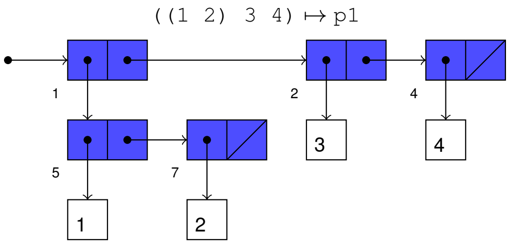
Vectors and Lists

Lists	**Vectors**
Constant-time append at the beginning	No append at all
Constant-time insert at any point (with mutation)	No insert at all
Accessing the n^{th} element takes $O(n)$	Accessing the n^{th} element takes constant time
Structure can be shared between different lists	Every vector is entirely disjoint
Rich set of built-in procedures (`map`, etc.)	Few built-ins (but you can build more)

Representing `cons` cells

- We will represent `cons` cells using two vectors, `the-cars` and `the-cdrs`.
- A `cons` cell is an index *i* into the arrays
  - Its `car` is `(vector-ref the-cars i)`
  - Its `cdr` is `(vector-ref the-cdrs i)`
- To represent other data, we'll use tagging.
  - `ni` is a number with value *i*
  - `pi` is a pointer to a pair at index *i*
  - `e0` is the special empty list

An example



Index	0	1	2	3	4	5	6	7	8	...
the-cars		p5	n3		n4	n1		n2		...
the-cdrs		p2	p4		e0	p7		e0		...

car and cdr

```
(define (gc-car pair)
  (vector-ref the-cars (pointer-value pair)))

(define (gc-cdr pair)
  (vector-ref the-cdrs (pointer-value pair)))

(define (gc-set-car! pair new-car)
  (vector-set! the-cars
               (pointer-value pair) new-car))

(define (gc-set-cdr! pair new-cdr)
  (vector-set! the-cdrs
               (pointer-value pair) new-cdr))
```


cons

```
(define (gc-cons car cdr)
  (let ((pair (gc-new-pair)))
    (gc-set-car! pair car)
    (gc-set-cdr! pair car)
    pair))
```


gc-new-pair

```
(define *free* 0)
(define (gc-new-pair)
  (let ((new-pair *free*))
    (set! *free* (+ *free* 1))
    (tag-pointer 'pair new-pair)))
```

- What's wrong?

Too many cons cells!

```
(define (find-primes n)
 (define (helper ns)
  (cons (car ns) (find-primes
   (filter (lambda (i) (not (divides? i n))) (cdr ns)))))
 (helper (cdr (integers-less-than n))))
```

- `(2 3 4 5 6 7 8 9 10 11 12 ...)`
- `(3 5 7 9 11 13 15 17 19 21 ...)`
- `(5 7 11 13 17 19 23 25 27 ...)`

Goal: Re-use storage

- Every `filter` step generates intermediate lists
- But those lists can never be accessed again!
- We can re-use that storage space

The Big Idea

- We can **simulate** a machine with infinite memory by detecting and re-using memory that can never be used again.
- How do we do that?

Reachability

- There is a set of objects (the “root set”) the program can directly access (e.g. the global environment)
- Objects can point to other objects (e.g. cons cells, the environment pointer of a lambda)
- Any object that is transitively reachable by following pointers from the root set is **live** and must be preserved.
- Anything else is **garbage** and can be reused.

First try: Reference Counting

- We could keep track of how many pointers there are to each object.
- Every time we generate a new reference to an object, we increase the reference count.
  - `define`
  - `set!`
  - `apply` a compound procedure
  - ...
- Whenever we remove a reference to an object, decrease the count.
  - `set!` (The old value)
  - After `apply`ing a compound procedure.
  - ...

Reference Counting: Problems

- Naïve refcounting leaks circular objects!

```
(define x (list 'a))
(set-cdr! x x)
(set! x 0)
```

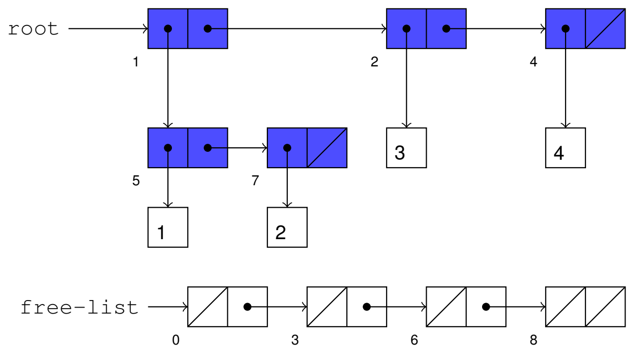
- Performance impact in many cases.
  - Every time you leave a frame, you need to walk its variables

Garbage Collection

- Describe the “root set” explicitly.
  - On real hardware, this is the “registers”
  - In `m-eval` this is (roughly) the global environment plus the current environment.
- Only objects reachable from this set by some sequence of `car` and `cdr` can ever matter.
- Any memory that is not accessible in this way is garbage, and can be reused.

mark-and-sweep

- mark-and-sweep is one of the simplest garbage collection algorithms, composed of two phases:
  1. Starting from the root set, recursively mark every reachable object.
  2. sweep all of memory, collecting every unmarked object into the free list.
- Allocation then takes place by removing new pairs from the free list.



Index	0	1	2	3	4	5	6	7	8	...
the-cars	e0	p5	n3	e0	n4	n1	e0	n2	e0	...
the-cdrs	p3	p2	p4	p6	e0	p7	p8	e0	e0	...
the-marks		#t	#t		#t	#t		#t		
	↑	↑ ↑	↑ ↑	↑	↑ ↑	↑ ↑	↑	↑ ↑	↑	↑

mark

```
(define (mark p)
  (if (and (gc-pair? p)
           (not (vector-ref
                 the-marks
                 (pointer-value p))))
      (begin
        (vector-set! the-marks
                     (pointer-value p)
                     #t)
        (mark (gc-car p))
        (mark (gc-cdr p)))))
```


sweep

```
(define (sweep i)
  (if (not (vector-ref the-marks i))
      (begin
        (vector-set! the-cars i *gc-nil*)
        (vector-set! the-cdrs i *free-list*)
        (set! *free-list* (tag-pointer 'pair i))))
  (if (> i 0)
      (sweep (- i 1))))
```


mark-and-sweep

```
(define (mark-and-sweep root)
  (clear-all-marks)
  (mark root)
  (set! *free-list* *gc-nil*)
  (sweep (- *memory-size* 1)))
```

gc-new-pair with a free list

```
(define (mark-and-sweep-new-pair)
  (if (eq? *free-list* *gc-nil*)
      (begin (mark-and-sweep root)
             (if (eq? *free-list* *gc-nil*)
                 (error "Out of memory"))))
  (let ((pair *free-list*))
    (set! *free-list*
          (gc-cdr *free-list*))
    pair))
```


mark-and-sweep: problems

- How do we keep track of state during `mark`?
- `sweep` needs to examine all of memory.
- Heap fragmentation becomes a big problem.

An alternate plan: Stop-and-copy

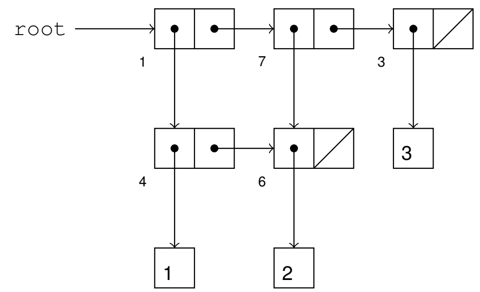
- To solve these problems, many real systems use some form of a copying garbage collector.
- Our stop-and-copy collector maintains two regions of memory, the working memory and the free memory.
- When we run out of memory, we copy live objects into the free memory, and switch the roles of the halves.

Stop-and-Copy

- We allocate pairs as we did initially with a `*free*` pointer.
- When we run out of memory, we switch the free and working memories, and we relocate `root` into the new free memory.
- We use a new pointer, `scan`, initially pointing at the start of the new free memory.
- As long as `scan` < `*free*`, we relocate the `car` and `cdr` of `scan`, and increment `scan`.

Relocation

- To relocate a pointer:
  - If the value it points to has already been copied, update it to point at the new location.
  - Otherwise, allocate a new pair, copy the pair it points to there, and update it.
    - Replace the `car` of the old pair with a tag known as a broken heart()
    - Replace the `cdr` of the old pair with the pair's new address.



root: p1

Index	0	1	2	3	4	5	6	7	8	...
the-cars		p4		n3	n1		n2	p6		...
the-cdrs		p7		e0	p6		e0	p3		...

root: p1p0

Index	0	1	2	3	4	5	6	7	8	...
thenew-cars		p4		n3	n1		n2	p6		...
thenew-cdrs		p7p0		e0p4	p6p1		e0p3	p3p2		...
free	↓	↓	↓	↓	↓	↓				↑

Index	0	1	2	3	4	5	6	7	8	...
newthe-cars	p4p1	n1	p6p3	n2	n3					...
newthe-cdrs	p7p2	p6p3	p3p4	e0	e0					...
scan	↑	↑	↑	↑	↑	↑				

stop-and-copy

```
(define (stop-and-copy)
  (define (loop scan)
    (if (< scan *free*)
        (begin
          (vector-set! new-cars scan
                       (relocate
                        (vector-ref new-cars scan)))
          (vector-set! new-cdrs scan
                       (relocate
                        (vector-ref new-cdrs scan)))
          (loop (+ scan 1)))))
  (set! *free* 0)
  (set! *root* (relocate *root*))
  (loop 0)
  (swap-spaces))
```

relocate

```
(define (relocate ptr)
  (if (gc-pair? ptr)
      (if (broken-heart? (gc-car pair))
          (gc-cdr pair)
          (let ((new-pair *free*))
            (set! *free* (+ 1 *free*))
            (vector-set! new-cars new-pair
                         (gc-car ptr))
            (vector-set! new-cdrs new-pair
                         (gc-cdr ptr))
            (gc-set-car! ptr *broken-heart*)
            (gc-set-cdr! ptr
                         (tag-pointer 'pair new-pair))
            (tag-pointer 'pair new-pair)))
      ptr))
```


Properties of stop-and-copy

- Since it moves things around, the garbage collector must know about *every* pointer into the heap.
- Compacts used memory into a single chunk
  - This means allocation is extremely efficient.
- You only get to use half of your memory.
  - But with mark-and-sweep you potentially needed that for the stack.
- Most modern GCs use something that looks more like stop-and-copy than mark-and-sweep.

Generational GC

- Think about the kinds of garbage a program creates.
- `find-primes` generated a lot of garbage, but it was very short-lived.
- In the adventure game, players, brains and items are created and destroyed, but tend to last a while first.
- This turns out to be true in general: A large amount of garbage is destroyed very quickly, whereas garbage that sticks around for a while is likely to stick around more.

Generational GC

- Big Idea: Have two (or more!) memory pools.
- Allocate everything into a small one, and scan it every time you do a GC.
- If an object survives a few garbage collections, move it into a larger pool, which is only fully scanned rarely.
- Nearly every real modern GC works roughly this way.