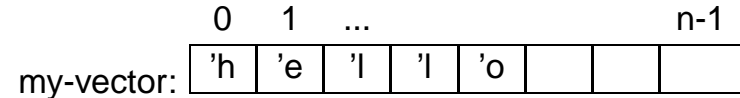


Memory management

- We could add explicit memory to our evaluators.
- Just consider problem of storing cons cells (a linear representation of box and pointer diagrams).
- What happens when we run out of memory: can we detect cells with no further use and recycle them?
- Intro to standard garbage collection techniques.

Vector abstraction of memory

- Memory normally referenced by sequential addresses: load from and store to an address.
- Sequentially addressed data structure in Scheme: vectors.



- Scheme

```
> (vector-ref my-vector 1)
e
> (vector-set! my-vector 0 'j)
```

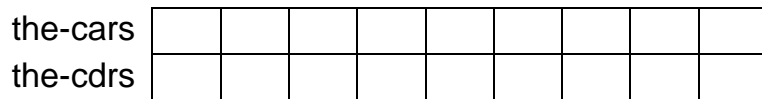
..p.1/1

..p.2/1

Vectors for cons cells

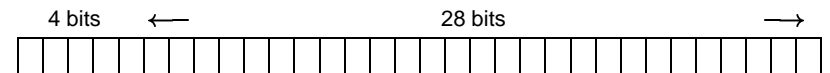
Representing primitive data

Pair of vectors



↑
Free

```
(car <exp>) => (vector-ref the-cars <exp>)
(cdr <exp>) => (vector-ref the-cdrs <exp>)
(cons <e1> <e2>) =>
  (let ((cell free))
    (vector-set! the-cars free <e1>)
    (vector-set! the-cdrs free <e2>)
    (set! free (+ free 1))
    cell)
```



- | | | | |
|-----|---------------------|-----|------------------------------|
| 0 | - Empty list | E0 | no value |
| 1 | - Cons cell pointer | P5 | pointer to memory location 5 |
| 2 | - Integer | N3 | the number 3 |
| 3 | - Boolean | ... | |
| ... | | | |
| 15 | - ... | | |

Example

```
(define a (list 4 7 6))
(define b (cons (a a)))
(define c (list 1))
(set! c (list 2 3))
```

the-cars	N6	N7	N4	P2	N1	N3	N2		
the-cdrs	E0	P0	P1	P2	E0	E0	P5		

a = 2
b = 3
c = 6
free = 7
4 no longer used

..-p.5/1

..-p.6/1

Automatic garbage collection

- The only cells in memory that matter are those that will be used in future computation (undecidable problem).
- Keep cells that are “reachable”. The rest are garbage.
- Find pointers into memory
 - from continuation (env / control model)
 - from registers, stack, globals (model of machine)Call these “roots”.
- Starting from roots, follow pointers to find all reachable memory.

Running out of memory

- free up unused memory, and continue using newly freed memory.
- In C, programmer allocates with `malloc` frees with `free`
 - + Very fine control.
 - Fail to free: “memory leak”.
 - “Free” memory still in use: bugs.
- Java, Scheme, ..., manage memory for programmer.

Mark-and-sweep algorithm

1. **Mark**
 - (a) Start at root: mark it, and put any pointers in stack.
 - (b) Visit each item in stack, looping until stack is empty.
 - i. If marked then nothing more to do: process next in stack.
 - ii. If unmarked then mark and put any pointers on stack.
2. **Sweep**
 - (a) Start at end of memory, building a free list.
 - (b) Sweep to other end of memory. Each unmarked cell is garbage: add to free list.Clean up marks.

Mark-and-sweep algorithm

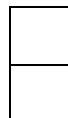
Mark-and-sweep example

```
(define (mark object)
  (cond ((not (pair? object)) #f)
        ((not (= 1 (vector-ref the-marks object)))
         (vector-set! the-marks object 1)
         (mark (cdr obj))
         (mark (car obj))))))

(define (sweep i)
  (cond ((not (= i size))
         (cond ((= 0 (vector-ref the-marks i))
                 (set-cdr! (int-to-pointer i) free)
                 (set! free (int-to-pointer i))))))
        (vector-set! the-marks i 0)
        (sweep (+ i 1))))))

(define (gc) (for-each mark roots) (sweep 0))
```

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1
marks	0	0	0	0	0	0	0	0	0



Stack:

Root = P5

Free = E0

--p.9/1

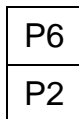
--p.10/1

Mark-and-sweep example

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1
marks	0	0	0	0	0	1	0	0	0

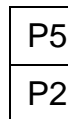
Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1
marks	0	0	0	0	0	1	1	0	0



Stack:

Root = P5

Free = E0



Stack:

Root = P5

Free = E0

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1
marks	0	0	0	0	0	1	1	0	0

P2

Stack:

Root = P5

Free = E0

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1
marks	0	0	1	0	0	1	1	0	0

P4
P0

Stack:

Root = P5

Free = E0

-- p.10/1

-- p.10/1

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1
marks	0	0	1	0	1	1	1	0	0

P0
P0

Stack:

Root = P5

Free = E0

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1
marks	1	0	1	0	1	1	1	0	0

P0

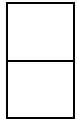
Stack:

Root = P5

Free = E0

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1
marks	1	0	1	0	1	1	1	0	0



Stack:

Root = P5

Free = E0

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1
marks	0	0	1	0	1	1	1	0	0



Stack:

Root = P5

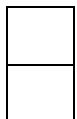
Free = E0

-- p.10/1

-- p.10/1

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1
marks	0	0	1	0	1	1	1	0	0



Stack:

Root = P5

Free = P1

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1
marks	0	0	0	0	1	1	1	0	0



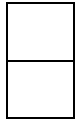
Stack:

Root = P5

Free = P1

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P1	P0	P6	P5	P3	P1
marks	0	0	0	0	1	1	1	0	0



Root = P5

Free = P3

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P1	P0	P6	P5	P3	P1
marks	0	0	0	0	0	1	1	0	0



Root = P5

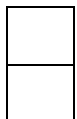
Free = P3

-- p.10/1

-- p.10/1

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P1	P0	P6	P5	P3	P1
marks	0	0	0	0	0	0	1	0	0



Root = P5

Free = P3

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P1	P0	P6	P5	P3	P1
marks	0	0	0	0	0	0	0	0	0

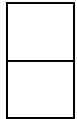


Root = P5

Free = P3

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P1	P0	P6	P5	P3	P1
marks	0	0	0	0	0	0	0	0	0



Stack:

Root = P5

Free = P7

-- p.10/11

Mark-and-sweep example

Memory	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N2	P1
the-cdrs	E0	E0	P4	P1	P0	P6	P5	P3	P7
marks	0	0	0	0	0	0	0	0	0



Stack:

Root = P5

Free = P8

-- p.10/11

Mark-and-sweep features

- Mark only looks at reachable memory, but sweep looks at all memory.
- Getting cells from free list takes several instructions.
- Looks like you need potentially double the space, to keep stack, but can actually maintain a stack by reversing pointers during marking.
(Deutch-Schorr-Waite pointer reversal algorithm)
- + Needs minimal compiler support as roots do not change.

Can guess at roots, guess what might be a pointer, and put mark-and-sweep garbage collection in C. The Boehm-Demers-Weiser conservative garbage collector: Know that it exists. Use it.

Stop-and-copy algorithm

Use 2 spaces (Current, Next). When one fills up, copy to the other.

1. Set Free to beginning of Next space. Copy root pair, to first position in next space. Leave "Redirect" marker and pointer to new position. Set Scan to beginning of Next space.
2. Starting at the Scan pointer... If car or cdr is a pointer: If redirected, fix pointer in Next space. If not redirected, copy to Next space, redirect, and bump Free pointer.
3. Increment Scan and loop back to step 2 until Scan pointer catches up with Free pointer.
4. Update root. Swap Current and Next spaces.

Stop-and-copy algorithm

Stop-and-copy example

```
(define (gccopy scan)
  (if (not (= scan free))
      (begin
        (set-car! scan (forward (vector-ref the-cars scan)))
        (set-cdr! scan (forward (vector-ref the-cdrs scan)))
        (gccopy (+ scan 1))))))

(define (forward object)
  (cond ((pair? object)
        (let ((oldcar (car object)))
          (if (forward? oldcar)
              (int-to-object oldcar) ; redirected: fix pointer
              (let ((newptr (int-to-ptr free))) ; else move
                (set-car! newptr oldcar)
                (set-cdr! newptr (cdr object))
                (set! free (+ free 1))
                (set-car! object (int-to-pointer newptr))
                newptr))))))
        (else object))) ; just copy non-pair
```

Curr	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	P2	N2	N3	P1
the-cdrs	E0	E0	P4	P5	P0	P6	P5	P3	P1

Next	0	1	2	3	4	5	6	7	8
the-cars									
the-cdrs									
	F								

Root = P5

--p.13/1

--p.14/1

Stop-and-copy example

Stop-and-copy example

Curr	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	P0	N3	N5	R	N2	N3	P1
the-cdrs	E0	E0	P4	P5	P0	p0	P5	P3	P1

Curr	0	1	2	3	4	5	6	7	8
the-cars	N3	N4	R	N3	N5	R	R	N3	P1
the-cdrs	E0	E0	p1	P5	P0	p0	p2	P3	P1

Next	0	1	2	3	4	5	6	7	8
the-cars	P2								
the-cdrs	P6								
	S	F							

Next	0	1	2	3	4	5	6	7	8
the-cars	p1	P0	N2						
the-cdrs	p2	P4	P5						
		S		F					

Root = P5

Root = P5

Stop-and-copy example

Curr	0	1	2	3	4	5	6	7	8
the-cars	R	N4	R	N3	R	R	R	N3	P1
the-cdrs	p3	E0	p1	P5	p4	p0	p2	P3	P1

Next	0	1	2	3	4	5	6	7	8
the-cars	p1	p3	N2	N3	N5				
the-cdrs	p2	p4	P5	E0	P0				
			S			F			

Root = P5

--p.14/1

Stop-and-copy example

Curr	0	1	2	3	4	5	6	7	8
the-cars	R	N4	R	N3	R	R	R	N3	P1
the-cdrs	p3	E0	p1	P5	p4	p0	p2	P3	P1

Next	0	1	2	3	4	5	6	7	8
the-cars	p1	p3	N2	N3	N5				
the-cdrs	p2	p4	p0	E0	P0				
				S		F			

Root = P5

--p.14/1

Stop-and-copy example

Curr	0	1	2	3	4	5	6	7	8
the-cars	R	N4	R	N3	R	R	R	N3	P1
the-cdrs	p3	E0	p1	P5	p4	p0	p2	P3	P1

Next	0	1	2	3	4	5	6	7	8
the-cars	p1	p3	N2	N3	N5				
the-cdrs	p2	p4	p0	E0	P0				
				S		F			

Root = P5

Stop-and-copy example

Curr	0	1	2	3	4	5	6	7	8
the-cars	R	N4	R	N3	R	R	R	N3	P1
the-cdrs	p3	E0	p1	P5	p4	p0	p2	P3	P1

Next	0	1	2	3	4	5	6	7	8
the-cars	p1	p3	N2	N3	N5				
the-cdrs	p2	p4	p0	E0	p3				
						F			
						S			

Root = P5

Stop-and-copy example

Stop-and-copy Features

Next	0	1	2	3	4	5	6	7	8
the-cars	R	N4	R	N3	R	R	R	N3	P1
the-cdrs	p3	E0	p1	P5	p4	p0	p2	P3	P1

Curr	0	1	2	3	4	5	6	7	8
the-cars	p1	p3	N2	N3	N5				
the-cdrs	p2	p4	p0	E0	p3				
						F			

Root = p0

- + Only examines locations reachable from root.
- + Free pointer easy to manage.
- Need double the space that you are using for reachable data. (Pay in space, get time.)
- Needs compiler support to find and update roots (in registers, on stack).
- Breadth-first copy: Changes relative positions of data, whatever implications may be on future cache behavior.

-- p.14/1

-- p.15/1

Reference counting

Variations

- Attach a counter to each cons cell in memory.
- When a new pointer is pointed to that cons cell increment the counter.
- When a pointer that was pointing at the cons cell stops pointing there, decrement the counter.
- If counter goes to 0, release any pointers in the car or cdr of the cell and put the cell on the free list.
- + Can package all this up in a C++ `reference_counted` class.
- Extra overhead on each pointer change.
- If there is a loop in the box-and-pointer structure, then the counts for the cells in the loop never fall to 0.

- Barely scraped surface:
- Multi-generational copying collectors.
 - Compacting Mark-and-sweep.
 - Support for threads.
 - Support for multiprocessors.
 - Support for hard real-time.
 - Some tutorial papers:
 - Paul Wilson "Uniprocessor Garbage Collection Techniques", 1992
 - Henry Baker's columns in Sigplan Notices
 - Research topic at Microsoft, Sun, various universities.