GC III: Mark & Sweep

David Bremner

April 1, 2024

(Mark & Sweep) Garbage Collection Algorithm

- Color all records white
- Color records referenced by registers gray
- Repeat until there are no gray records:
  - Pick a gray record, r
  - For each white record that r points to, make it gray
  - Color r black
- Deallocate all white records

(see animation in Lecture 18)

Heap model

- We'll use tags to indicate both color and kind of data
- Unqualifed tags will denote the black color
- E.g., `'flat` vs `'white-flat` vs `'gray-flat`
- cells are marked `'free` individually.

```
; init-allocator : -> void?
(define (init-allocator)
  (for ([i (in-range 0 (heap-size))])
    (heap-set! i 'free)))
```

API providing code is mostly same as null-gc.

```
; gc:cons : root? root? -> location?
(define (gc:cons v1 v2)
  (define address (malloc 3 v1 v2))
  (heap-set! address 'cons)
  (heap-set! (+ address 1) (read-root v1))
  (heap-set! (+ address 2) (read-root v2))
  address)
```

Need to store more info in “object header”

```
(define (gc:closure code-ptr free-variables)
  (define address
    (malloc (+ 3 (length free-variables))
       free-variables))
  (heap-set! address 'clos)
  (heap-set! (+ address 1) code-ptr)
  (heap-set! (+ address 2) (length free-variables))
  (for ([i (in-range 0 (length free-variables))]
        [f (in-list free-variables)])
    (heap-set! (+ address 3 i) (read-root f)))
  address)
```

- Needed for e.g. marking heap.

Allocations look almost the same.

cons-ex2

```
(define (cons-test)
    (cons 1 2))

(define the-cons (cons-test))
```

	0	1	2	3	4	5	6	7	8	9
0	'clos	cons-test	0	'flat	1	'flat	2	'cons	3	5
10	'free	'free	'free	'free	'free	'free	'free	'free	'free	'free

Malloc tries to re-use gaps

```
(define (malloc n . extra-roots)
  (define initial (find-free-space n))
  (unless initial
    (collect-garbage extra-roots))
  (define second (or initial (find-free-space n)))
  (if second second
      (error 'alloc "out of memory")))
```

- `extra-roots` are passed to gc
- `find-free-space` scans heap
- convention of returning `#f` on failure

Brute force allocator I/II

```
(define (find-free-space n) ;  size -> (or/c addr #f)
  (define (loop start)
    (and
     (< start (heap-size))
     (case (heap-ref start)
       [(flat) (loop (+ start 2))]
       [(cons) (loop (+ start 3))]
       [(clos) (loop
                (+ start 3 (heap-ref (+ start 2))))]
       [(free) (if (n-free-blocks? start n)
                   start
                   (loop (+ start 1)))]
       [else (error 'find-free-space
                    "unexpected tag ~a" start)])))
  (loop 0))
```

Brute force allocator, II/II

```
; n-free-blocks? : location? integer? -> boolean?
(define (n-free-blocks? start n)
  (or
   (<= n 0)
   (and
    (< start (heap-size))
    (equal? (heap-ref start) 'free)
    (n-free-blocks? (+ start 1) (- n 1)))))
```

Testing the allocator

ms

```
(module+ test
  (with-heap (make-vector 6 #f)
    (init-allocator)
    (test (malloc 4) 0)
    (heap-set! 0 'flat)
    (heap-set! 1 42)
    (test (current-heap)
          #(flat 42 free free free free))
    (test (malloc 2) 2)))
```

Yes, we know this is ridiculously inefficient

- A proper mark and sweep collector would use a data structure like
  - a free list
  - a bitmap
- Or an external allocator
- See Chapter 7 of the GC Handbook for more.

Top level collector

```
; collect-garbage : roots? -> void?
(define (collect-garbage . extra-roots)
  (validate-heap)
  (mark-white!)
  (traverse/roots (get-root-set))
  (traverse/roots extra-roots)
  (free-white!)
  (validate-heap))
```

Marking the entire heap as white

```
(define (mark-white!)
  (let loop ([i 0])
    (when (< i (heap-size))
      (case (heap-ref i)
        [(cons)
         (heap-set! i 'white-cons) (loop (+ i 3))]
        [(flat)
         (heap-set! i 'white-flat) (loop (+ i 2))]
        [(clos) (heap-set! i 'white-clos)
                (loop (+ i 3 (heap-ref (+ i 2))))]
        [(free) (loop (+ i 1))]
        [else (error 'mark-white! "bad tag: ~a" i)]))))
```

Traversing a root set

```
(define (traverse/roots roots)
  (cond
    [(list? roots)
     (for-each traverse/roots roots)]
    [(root? roots)
     (traverse/loc (read-root roots))]
    [else
     (error 'traverse/roots
            "unexpected roots: ~a" roots)]))
```

Traverse from a single location (pointer)

```
(define (traverse/loc loc)
  (case (heap-ref loc)
    [(flat gray-flat cons gray-cons clos gray-clos)
       (void)]
    [(white-flat) (heap-set! loc 'flat)]
    [(white-cons) (heap-set! loc 'gray-cons)
     (traverse/loc (heap-ref (+ loc 1)))
     (traverse/loc (heap-ref (+ loc 2)))
     (heap-set! loc 'cons)]
    [(white-clos) (heap-set! loc 'gray-clos)
     (for ([i (in-range 0 (heap-ref (+ loc 2)))])
       (traverse/loc (heap-ref (+ loc i 3))))
     (heap-set! loc 'clos)]
    [else (error 'traverse/loc
                 "unexpected tag: ~a" loc)]))
```

Freeing the still white nodes I/II

```
(define (free-white!)
  (let loop ([i 0])
    (when (< i (heap-size))
      (case (heap-ref i)
        [(cons) (loop (+ i 3))]
        [(flat) (loop (+ i 2))]
        [(clos) (loop (+ i 3 (heap-ref (+ i 2))))]
        [(free) (loop (+ i 1))]
        [(white-flat) (heap-set! i 'free)
                      (heap-set! (+ i 1) 'free)
                      (loop (+ i 2))]
        [(white-cons) (heap-set! i 'free)
                      (heap-set! (+ i 1) 'free)
                      (heap-set! (+ i 2) 'free)
                      (loop (+ i 3))]
        [(white-clos) (heap-set! i 'free)
```

Freeing the still white nodes II/II

```
[(white-flat) (heap-set! i 'free)
              (heap-set! (+ i 1) 'free)
              (loop (+ i 2))]
[(white-cons) (heap-set! i 'free)
              (heap-set! (+ i 1) 'free)
              (heap-set! (+ i 2) 'free)
              (loop (+ i 3))]
[(white-clos) (heap-set! i 'free)
              (heap-set! (+ i 1) 'free)
              (define size (heap-ref (+ i 2)))
              (for ([x (in-range 0 size)])
                (heap-set! (+ i 3 x) 'free))
              (heap-set! (+ i 2) 'free)
              (loop (+ i 3 size))]
```

Our friend fib

fib

```
(allocator-setup "mark-sweep.rkt" 160)
(define (fib n)
  (cond
    [(<= n 1) 1]
    [else (+ (fib (- n 1)) (fib (- n 2)))]))

(fib 20)
```

	0	1	2	3	4	5	6	7	8	9
0	'clos	fib	0	'flat	20	'flat	2	'flat	6765	'flat
10	2	'flat	1	'flat	18	'flat	377	'flat	1	'flat
20	12	'flat	#t	'flat	1	'flat	3	'flat	2	'flat
30	2	'flat	1	'flat	#f	'flat	1	'flat	1	'flat

Mark & Sweep Pros & Cons

Cons

- Cost of collection proportional to (entire) heap
- Bad locality (and fragmentation!)
- Our M&S also has a terrible allocator on top of that

Pros

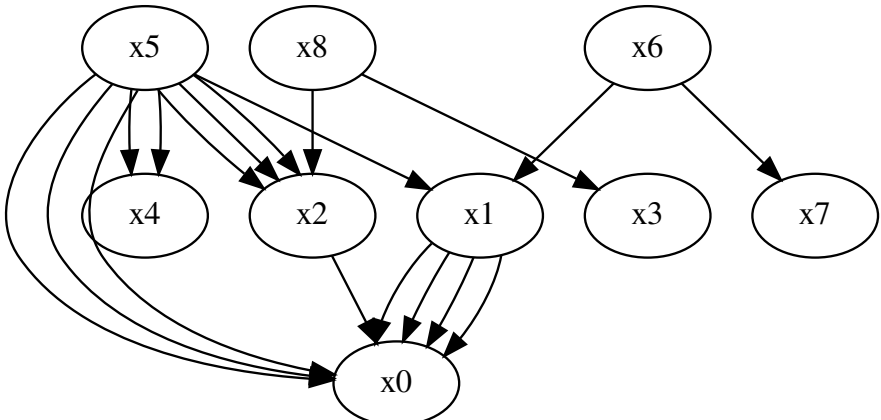
- allocator is fixable
- use of full heap (compared to 2space)
- follows closely the generic GC algorithm
- non-moving collectors work for “conservative collectors”

Generating random mutators

- A library `plai/random-mutator` generates random code.
- Generate a random (directed) graph, then turn it into code.

gen1
```
(require plai/random-mutator)
(save-random-mutator "mutator2.rkt" "mark-sweep.rkt"
                     #:gc2? #t #:iterations 10)
```

Generated Mutator



Heap State

	0	1	2	3	4	5	6	7	8	9
0	198	'clos	build-one	'clos	traverse-o	'clos	trigger-g	'clos	loop	'flat
10	10	'flat	#f	'flat	0	'clos	x1	13	'clos	x2
20	13	'flat	#t	'flat	#t	'clos	x5	23	18	15
30	13	'flat	#f	'cons	36	15	'flat	'y	'cons	18
40	21	'flat	200	'flat	#f	'cons	41	41	'flat	1
50	'flat	199	'flat	#f	'cons	50	50	'flat	1	'flat
60	198	'flat	#f	'cons	59	59	'flat	1	'flat	197
70	'flat	#f	'cons	68	68	'flat	1	'flat	196	'flat
80	#f	'cons	77	77	'flat	1	'flat	195	'flat	#f
90	'cons	86	86	'flat	1	'flat	194	'flat	#f	'cons
100	95	95	'flat	1	'flat	193	'flat	#f	'cons	104
110	104	'flat	1	'flat	192	'flat	#f	'cons	113	113
120	'flat	1	'flat	191	'flat	#f	'cons	122	122	'flat
130	1	'flat	190	'flat	#f	'cons	131	131	'flat	1
140	'flat	189	'flat	#f	'cons	140	140	'flat	1	'flat
150	188	'flat	#f	'cons	149	149	'flat	1	'flat	187

Acknowledgements / references

- https://docs.racket-lang.org/plai/gc2-collector.html
- https://docs.racket-lang.org/plai/Generating_Random_Mutators.html
- Lecture 20 based in part on slides by Vincent St. Amour.