GC IV: Improving allocation

David Bremner

April 1, 2024

Allocation speed

- In `https://www.cs.unb.ca/~bremner/teaching/cs4613/lectures/lecture20` we saw mark-and-sweep GC
- This had the naive approach of scanning the entire heap for every allocation.
- Two standard improvements are keeping a bitmap of free / allocated locations, and keeping a list of free records

Allocation bitmap

1	1	0						flat	1	?

```
;(heap-size)>=(bitmap-words)+block-width*(bitmap-words)
(define (block-width) 8)
(define (bitmap-words)
  (quotient (heap-size)
            (add1 (block-width))))

(define (init-allocator)
  (for ([i (in-range (bitmap-words))])
    (heap-set! i 0)))
```

Malloc is mostly the same

```
(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)))
  (unless second
    (error 'alloc "out of memory"))
  (update-bits! second n #t) ;; CHANGED
  second)
```

Updating the bitmap

```
(define (ones k) (sub1 (expt 2 k)))
(define (update-bits! loc how-many set?)
  (define (flip bits)
    (bitwise-xor bits (ones (block-width))))
  (let* ([addr (- loc (bitmap-words))]
         [block (quotient addr (block-width))]
         [index (- addr (* block (block-width)))]
         [diff
          (arithmetic-shift (ones how-many) index)]
         [current (heap-ref block)])
    (heap-set!
     block
     (if set? (bitwise-ior current diff)
         (bitwise-and current (flip diff))))))
```

Bitmap as numbers

	0	1	2	3	4	5	6	7	8	9
0	127	7	'clos	cons-tes	0	'flat	1	'flat	2	#f
10	'cons	5	7	#f	#f	#f	#f	#f	#f	#f

cons

```
(allocator-setup "bitmapped-fits.rkt" 20)

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

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

- bitmap is byte / word addressable
- 0 is empty block
- otherwise we can (pre)compute the biggest space in block

Optimization I: skipping full blocks

```
(define (find-free-space n)
  (define (loop i)
    (define bits (heap-ref i))
    (cond
      [(> n (block-width))
       (error 'find-free-space
              "allocation > ~a" (block-width))]
      [(>= i (bitmap-words)) #f]
      [(>= (max-gap bits) n)
       (+ (* (block-width) i)
          (first-fit bits (ones n)))]
      [else (loop (add1 i))]))
  (define offset (loop 0))
  (and offset (+ (bitmap-words) offset)))
```

Optimization II: skipping empty blocks

```
(define (for/bitmap/proc action)
  (for ([block (in-range 0 (bitmap-words))]
        #:unless (zero? (heap-ref block)))
    (define start (+ (bitmap-words)
                           (* block (block-width))))
    (define (loop loc)
      (define index (- loc start))
      (cond
        [(>= index (block-width)) (void)]
        [(bitwise-bit-set?
          (heap-ref block) index)
         (action loc loop)]
        [else (loop (add1 loc))]))
    (loop start)))
```

(Pre)-calculating gaps

```
;; use dynamic programming to find longest run of 0s
(define (ending-at bits pos acc best)
  (cond
    [(>= pos (block-width)) best]
    [(bitwise-bit-set? bits pos)
     (ending-at bits
                (add1 pos) 0
                (max acc best))]
    [else
     (ending-at bits
                (add1 pos) (add1 acc)
                (max (add1 acc) best))]))

;; memoize the gap finding
(define max-gap
  (let ([gap-table (make-vector (expt 2 (block-width))
```

Using for/bitmap I/II

```
(define (mark-white!)
  (for/bitmap (loc loop)
    (case (heap-ref loc)
      [(cons)
       (heap-set! loc 'white-cons)
       (loop (+ loc 3))]
      [(flat)
       (heap-set! loc 'white-flat)
       (loop (+ loc 2))]
      [(clos)
       (heap-set! loc 'white-clos)
       (loop (+ loc 3 (heap-ref (+ loc 2))))]
      [else (error 'mark-white!
                   "unexpected tag: ~a" loc)])))
```

Using for/bitmap II/II

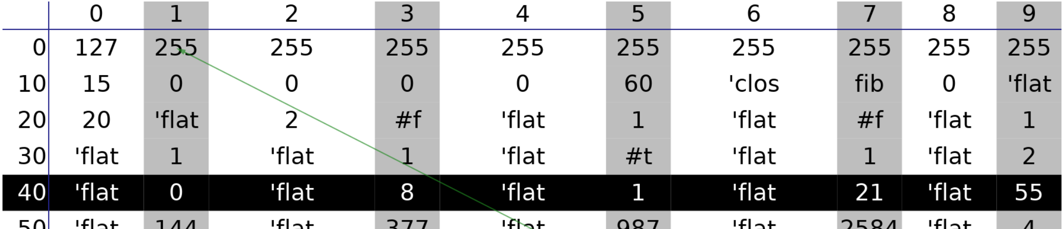
```
(define (free-white!)
  (for/bitmap (loc loop)
    (define (free! width)
      (update-bits! loc width #f) (loop (+ loc width)))
    (case (heap-ref loc)
      [(white-clos) (free! (+ 3 (heap-ref (+ loc 2))))]
      [(clos) (loop (+ loc 3 (heap-ref (+ loc 2))))]
      [(white-flat) (free! 2)]
      [(flat) (loop (+ loc 2))]
      [(white-cons) (free! 3)]
      [(cons) (loop (+ loc 3))]
      [else (error 'free-white!
                   "bad tag at ~a" loc)])))
```

Fibonacci Example

fib

```
(allocator-setup "bitmapped-fits.rkt" 144)
(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	127	255	255	255	255	255	255	255	255	255
10	15	0	0	0	0	60	'clos	fib	0	'flat
20	20	'flat	2	#f	'flat	1	'flat	#f	'flat	1
30	'flat	1	'flat	1	'flat	#t	'flat	1	'flat	2
40	'flat	0	'flat	8	'flat	1	'flat	21	'flat	55
50	'flat	144	'flat	377	'flat	987	'flat	2584	'flat	4

Sum Example

sum-ms

```
(allocator-setup "bitmapped-fits.rkt" 54)
(define (sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst) (sum (rest lst)))]))

(sum '(1 2 3 4 5))
```

	0	1	2	3	4	5	6	7	8	9
0	127	255	255	255	127	0	'clos	sum	0	'flat
10	1	'flat	2	#f	'flat	3	'flat	4	'flat	5
20	'flat	empty	'cons	18	20	'cons	16	22	'flat	9
30	'cons	14	25	'cons	11	30	'flat	12	'cons	9

Pros and cons of bitmap allocators

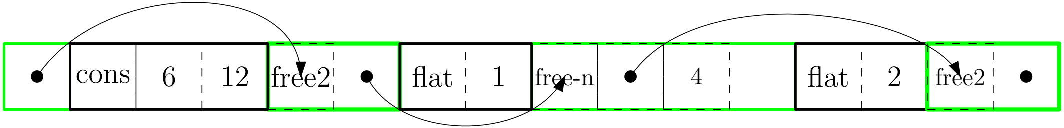
Pros

- Faster than naive linear scan
- Coalescing is automatic
- Re-uses non-contiguous free space, compared to bump-pointer
- Bitmap compactness is good for cache

Cons

- Only a constant factor faster
- more complex implementation
- heap overhead for metadata
- handling multi-block allocations

Defining a free list in the heap



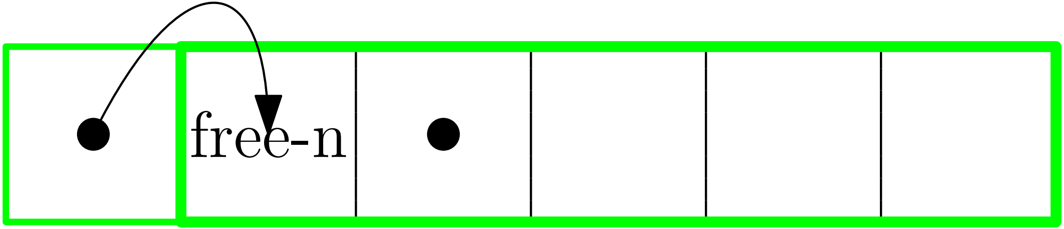
```
(define (fl:set-next! prev loc)
  (when prev (fl:check prev))
  (heap-set! (if prev (+ prev 1) FREE-LIST) loc))

(define (fl:init! loc size next)
  (case size
    [(2) (heap-set! loc 'free-2)]
    [else
     (heap-set! loc 'free-n)
     (fl:set-length! loc size)])
  (fl:set-next! loc next))
```

Initializing the free list

```
(define (init-allocator)
  (heap-set! FREE-LIST HEAP-START) ; head of free list
  (fl:init! HEAP-START (- (heap-size) HEAP-START) #f))
```

msfl

```
(with-heap (make-vector 6 #f)
  (init-allocator)
  (test (current-heap)
        #(1 free-n #f 5 #f #f)))
```

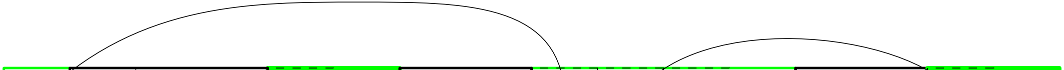


Finding free space: main loop

```
(case (heap-ref start)
  [(free-2)
   (cond
     [(= size 2) (delete-current!) start]
     [else (loop (fl:next start) start)])]
  [(free-n)
   (define length (heap-ref (+ start 2)))
   (cond
     [(= size length) (delete-current!) start]
     [(< size length)
      (split-current! (+ start size)
                      (- length size))
      start]
     [else (loop (fl:next start) start)])]
  [else (error 'find-free-space "wrong tag ~s at ~s"
               (heap-ref start) start)])
```

Find free space details

```
(define (find-free-space size)
  (define (loop start prev)
    (define (split-current! loc free-size)
      (case free-size
        [(1) (delete-current!) (heap-set! loc 'free)]
        [else
         (fl:init! loc free-size (fl:next start))
         (fl:set-next! prev loc)]))
    (define (delete-current!)
      (fl:set-next! prev (fl:next start)))

     #;(⋮))
(let ([head (heap-ref FREE-LIST)])
  (and head (loop head #f))))
```



Freeing garbage: main loop

```
(define (free-white!)
  (define (loop loc prev last-start spaces-so-far)
    (define (tag-of len)
      (case len [(1) 'free] [(2) 'free-2]
            [else 'free-n]))
    (define (write-free-record! where next)
      (heap-set! where (tag-of spaces-so-far))
      (when (>= spaces-so-far 2)
        (heap-set! (+ 1 where) next))
      (when (>= spaces-so-far 3)
        (heap-set! (+ 2 where) spaces-so-far))
        (fl:set-next! prev last-start))

    #;(⋮))
  (loop HEAP-START #f #f #f))
```

Freeing garbage: loop body I/II

```
(define merging (and last-start spaces-so-far
                     (> spaces-so-far 1)))
(cond
  [(>= loc (heap-size))
   (when merging (write-free-record! last-start #f))]
  [else
   (define length (object-length loc))
   #;(⋮)])
```

Freeing Garbage: loop body II/II

```
(case (heap-ref loc)
  [(flat cons clos)
   (when merging (write-free-record! last-start #f))
   (loop (+ loc length)
         (if merging last-start prev) #f #f)]
  [(white-flat white-cons white-clos
               free free-2 free-n)
   (cond
     [(and last-start spaces-so-far)
      (loop (+ loc length) prev last-start
            (+ spaces-so-far length))]
     [else (loop (+ loc length)
                 prev loc length)])]
  [else (error 'free-white! "wrong tag at ~a" loc)])
```

Example mutator

sum-ms

```
(allocator-setup "mark-sweep-free-list.rkt" 160)
(define (sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst) (sum (rest lst)))]))

(sum '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18))
```

Heap state

	0	1	2	3	4	5	6	7	8	9
0	108	'clos	sum	0	'flat	1	'flat	2	'flat	3
10	'flat	4	'flat	5	'flat	6	'flat	7	'flat	8
20	'flat	9	'flat	10	'flat	11	'flat	12	'flat	13
30	'flat	14	'flat	15	'flat	16	'flat	17	'flat	18
40	'flat	empty	'cons	38	40	'cons	36	42	'cons	34
50	45	'cons	32	48	'cons	30	51	'cons	28	54
60	'cons	26	57	'cons	24	60	'cons	22	63	'cons
70	20	66	'cons	18	69	'cons	16	72	'cons	14
80	75	'cons	12	78	'cons	10	81	'cons	8	84
90	'cons	6	87	'cons	4	90	'flat	156	'flat	161
100	'flat	165	'flat	168	'flat	170	'flat	171	'free-n	#f
110	52	#f	'white-flat	#f	'white-flat	#f	'white-flat	#f	'white-flat	#f
120	'white-flat	#f	'white-flat	#f	'white-flat	#f	'white-flat	#f	'white-flat	#f
130	'white-flat	#f	'white-flat	#t	'white-flat	0	'white-flat	18	'white-flat	35
140	'white-flat	51	'white-flat	66	'white-flat	80	'white-flat	93	'white-flat	105

Acknowledgements / References

- Dynamic programming is covered in any (decent) algorithms book. For example Cormen et all, look for longest (something) subsequence.
- For more about allocation, see Chapter 7 of the Garbage Collection Handbook
- free list handling based on code from the Master's Thesis of Yixi Zhang `https://github.com/yixizhang/plai-gc/`