Garbage Collection

David Bremner

March 26, 2025

Automatic memory management

- PLAI2 chapter 11
- Garbage Collection Handbook
- https://docs.racket-lang.org/plai/gc2-collector.html
- https://docs.racket-lang.org/plai/gc2-mutator.html

The argument for automatic storage management

Manual is hard 2253 / 2263 / C programming in general

Manual is error prone Both security bugs and memory leaks are common with manually managed storage.

When can we automatically free an object

- When we can guarantee that it won't be used again in the computation (ground truth).
- this is too hard.

Two conservative approximations

Reference counting
: when number of references reaches zero (leave for later)

Garbage collection
: when an object is not reachable from *roots*

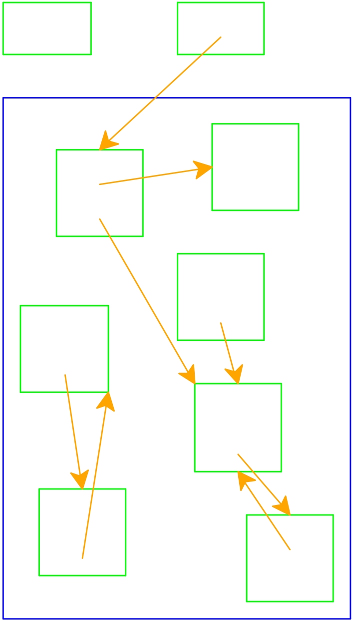
Garbage Collection

- Values reachable directly (without pointers) are live (the roots)
  E.g., values on the stack and in registers
- A record referenced by a live record is also live
- A program can only possibly use live records, because there is no way to get to other records
- A garbage collector frees all records that are not live
- Allocate until we run out of memory, then run a garbage collector to get more space

Garbage Collection Algorithm

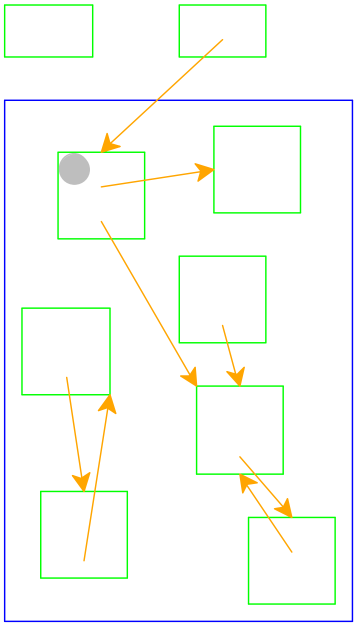
- Color all records white
- Color records referenced by registers gray
- Repeat until there are no gray records:
  - Pick a gray record, r
  - For each white record that r points to, make it gray
  - Color r black
- Deallocate all white records

Garbage Collection



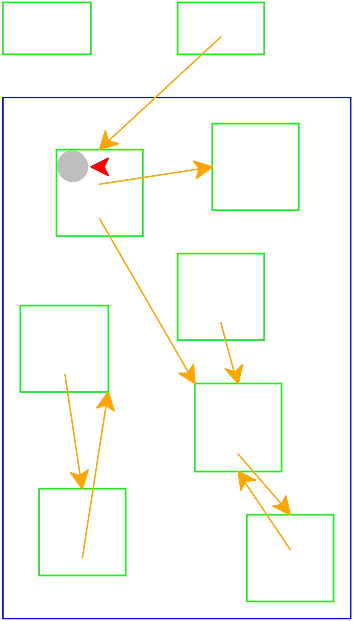
- All records are marked white

Garbage Collection



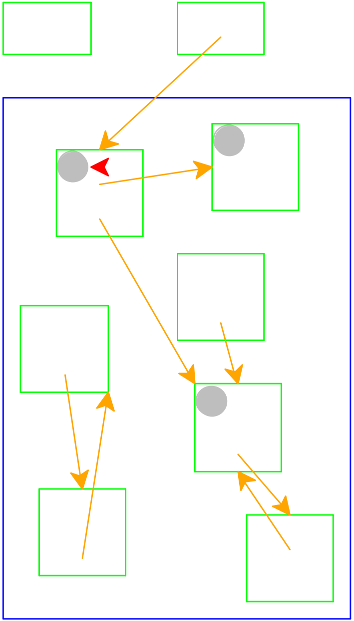
- Mark records referenced by registers as gray

Garbage Collection



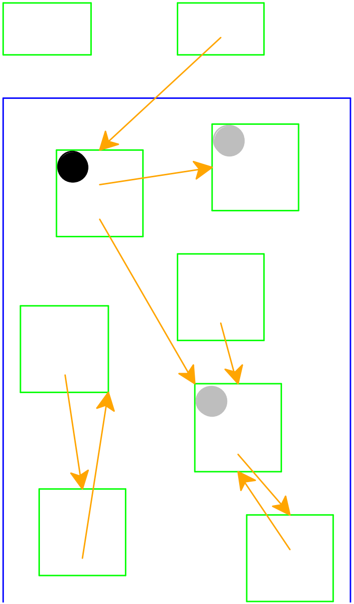
- Need to pick a gray record
- Red arrow indicates the chosen record

Garbage Collection



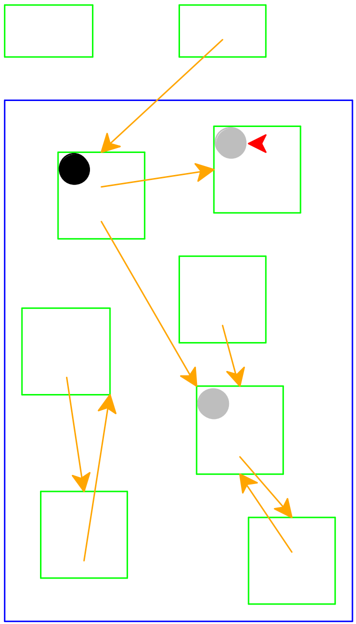
Mark white records referenced by chosen record as gray

Garbage Collection



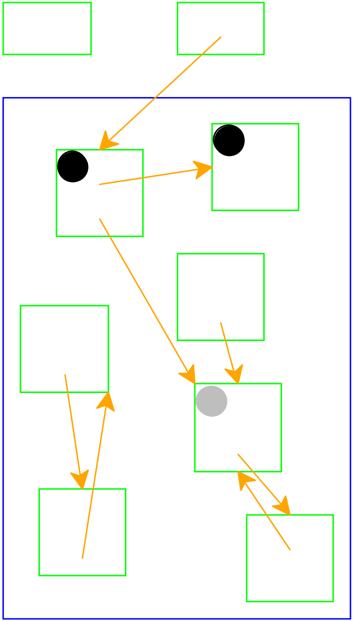
Mark chosen record black

Garbage Collection



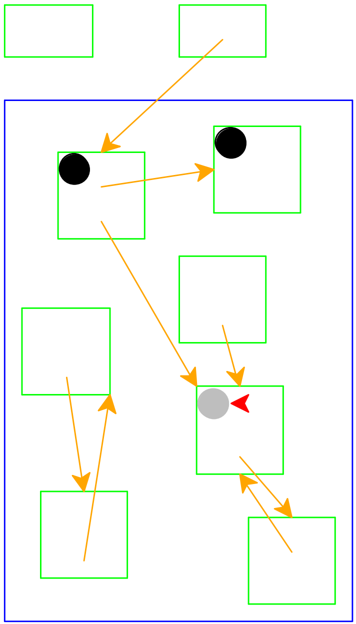
Start again: pick a gray record

Garbage Collection



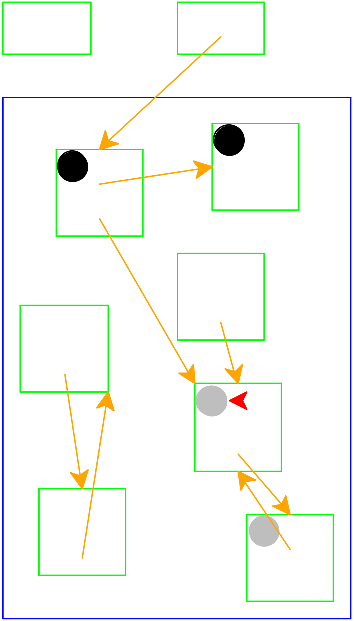
No referenced records; mark black

Garbage Collection



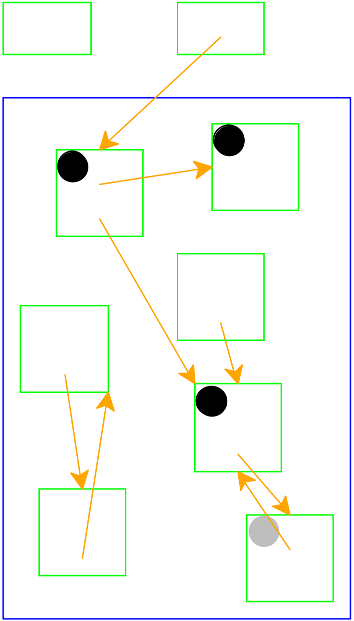
Start again: pick a gray record

Garbage Collection



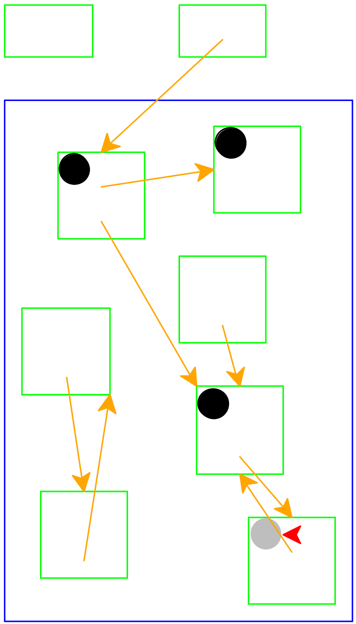
Mark white records referenced by chosen record as gray

Garbage Collection



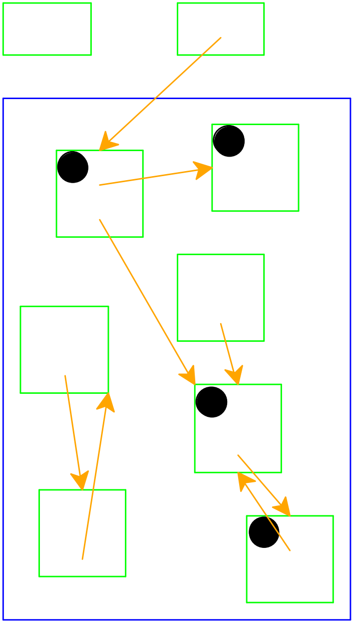
Mark chosen record black

Garbage Collection

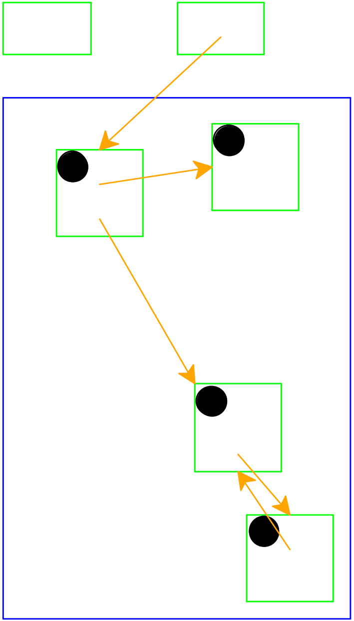


Start again: pick a gray record

Garbage Collection



No referenced white records; mark black



Garbage Collection

- No more gray records; deallocate white records
- Cycles do not break garbage collection (spoiler re: Reference counting)

Mutators and Collectors(s)

Programs divided into two parts

collector
: manages the heap, allocates memory, collects garbage to free space

mutator
: asks the collector for memory, does the work the program is supposed to do
  - from now on mostly used for test cases

collector API called by mutator

- Allocate a number,
- Allocate a pair,
- Give me the first element of that pair, ...

PLAI GC language(s)

Two languages

- `#lang plai/gc2/collector`
- `#lang plai/gc2/mutator`

Collectors implement a specific API

- See the docs: search for `init-allocator`

Collectors use an API provided by the collector language

- See the docs: search for `heap-ref`

PLAI GC language(s)

Two languages

- `#lang plai/gc2/collector`, `#lang plai/gc2/mutator`

The mutator language transforms mutators to

- keep track of roots
- make allocations explicit
- use the collector API

Mutators are (mostly) regular PLAI (racket) programs

- No need to use the (low-level) collector API directly!

Heap Model

- Like Lectures 10 - 11, but with symbols for tags.
- Heap is a vector of values
- Collector and mutator language are dynamically typed, allowing non-homogeneous heap.
- All values need to be allocated in the heap
- All values need to be tagged (to remember their type)

Atomic and compound values

- Atomic values include, numbers, symbols, booleans, and the empty list.
- Conceptually these fit in one cell; this is somewhat of a lie.
- Compound values include pairs and closures

Roots and compound values

Mutator creates roots to avoid race condition.

cons-ex

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

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

- i.e. 1 and 2 are on the heap, but nothing references them until the cons is allocated.

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

A non-collecting collector

- Put the allocation pointer at address 0 (visible)
- Allocate all constants in the heap, tag them with 'flat
- Allocate all conses in the heap, tag them with 'cons
- Allocate all closures in the heap, tag them with 'clos

Low level allocation

```
(define (init-allocator)
  (heap-set! 0 1))

(define (malloc n)
  (define addr (heap-ref 0))
  (unless (<= (+ addr n) (heap-size))
    (error 'allocator "out of memory"))
  (heap-set! 0 (+ addr n))
  addr)
```

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

Atomic values

```
(define (gc:alloc-flat x)
  (define addr (malloc 2))
  (heap-set! addr 'flat)
  (heap-set! (+ addr 1) x)
  addr)

(define (gc:deref addr)
  (unless (equal? (heap-ref addr) 'flat)
    (error 'gc:deref "not a flat at ~a" addr))
  (heap-ref (+ addr 1)))
```

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

Allocating pairs

```
(define (gc:cons f r)
  (define addr (malloc 3))
  (heap-set! addr 'cons)
  (heap-set! (+ addr 1) (read-root f))
  (heap-set! (+ addr 2) (read-root r))
  addr)
```

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

Accessing pairs

```
(define (gc:first addr)
  (check-pair addr)
  (heap-ref (+ addr 1)))

(define (gc:rest addr)
  (check-pair addr)
  (heap-ref (+ addr 2)))
```

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

Allocating closures 1/2

closure-ex

```
(let ([x 100]
      [y 1000])
  (lambda () (+ x y)))
```

	0	1	2	3	4	5	6	7	8	9
0	9	'flat	100	'flat	1000	'clos	#<proc>	1	3	#f
10	#f	#f	#f	#f	#f	#f	#f	#f	#f	#f

Allocating closures 2/2

```
(define (gc:closure code-pointer free-vars)
  (define addr (malloc (+ 2 (length free-vars))))
  (heap-set! addr 'clos)
  (heap-set! (+ addr 1) code-pointer)
  (for ([i (in-range 0 (length free-vars))]
        [v (in-list free-vars)])
    (heap-set! (+ addr 2 i)
               (read-root v)))
  addr)
```

	0	1	2	3	4	5	6	7	8	9
0	9	'flat	100	'flat	1000	'clos	#<proc>	1	3	#f
10	#f	#f	#f	#f	#f	#f	#f	#f	#f	#f

Testing a collector without a mutator

```
(with-heap h-expr body-exprs ...)
```

- `h-expr` must evaluate to a vector
- that vector is used for `heap-ref` and `heap-set!`
- `body-exprs` can (must) use the collector API.

Testing the non-collecting collector

null-gc

```
(module+ test
  (with-heap (vector 'x 'x 'x 'x 'x)
    (init-allocator)
    (gc:alloc-flat #f)
    (test (current-heap) (vector 3 'flat #f 'x 'x))))
```

Testing our non-collecting collector

null-gc

```
(module+ test
  (with-heap (vector 'x 'x 'x 'x 'x 'x 'x 'x 'x)
    (init-allocator)
    (gc:cons
     (simple-root (gc:alloc-flat #f))
     (simple-root (gc:alloc-flat #t)))
    (test (current-heap)
          (vector 8 'flat #f 'flat #t 'cons 1 3 'x))))
```

Testing with mutator programs

cons2

```
(allocator-setup "null-gc.rkt" 20) ; heap size

(define c1 (cons 2 (cons 3 empty)))
(define c2 (cons 1 c1))

(test/location=? (rest c2) c1) ; point to same location

(test/value=? (rest c1) '(3)) ; produce same value
```

	0	1	2	3	4	5	6	7	8	9
0	18	'flat	2	'flat	3	'flat	empty	'cons	3	5
10	'cons	1	7	'flat	1	'cons	13	10	#f	#f

Our friend fib

fib

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

(fib 5)
```