Garbage Collection Basics

Freeing memory is a pain

- Need to decide on a protocol (who frees what when?)
- Pollutes interfaces
- Errors hard to track down
- Remember 211 / 213?
- ... but lets try an example anyway
  (fire isn't hot until it burns ***me***)

Freeing memory is a pain

```
void removeSome (node *head) {
    node *current;
    while (current) {
        if (shouldRemove(current->payload)) {
            current->prev->next = current->next;
            current->next->prev = current->prev;
        }
        current = current->next;
    }
}
```

We're not freeing anything; leaking like a sieve.

Freeing memory is a pain

```
void removeSome (node *head) {
    node *current;
    while (current) {
        if (shouldRemove(current->payload)) {
            free(current->payload);
            current->prev->next = current->next;
            current->next->prev = current->prev;
        }
        current = current->next;
    }
}
```

Better, but still leaking.

Freeing memory is a pain

```
void removeSome (node *head) {
    node *current, *next;
    while (current) {
        if (shouldRemove(current->payload)) {
            free(current->payload);
            free(current);
            current->prev->next = current->next;
            current->next->prev = current->prev;
        }
        current = current->next;
    }
}
```

Now we're freeing too soon.

Freeing memory is a pain

```
void removeSome (node *head) {
    node *current, *next;
    while (current) {
        if (shouldRemove(current->payload)) {
            free(current->payload);
            current->prev->next = current->next;
            current->next->prev = current->prev;
            free(current);
        }
        current = current->next;
    }
}
```

Still too soon.

Freeing memory is a pain

```
void removeSome (node *head) {
    node *current, *next;
    while (current) {
        next = current->next;
        if (shouldRemove(current->payload)) {
            free(current->payload);
            current->prev->next = next;
            next->prev = current->prev;
            free(current);
        }
        current = next;
    }
}
```

Finally got it right. I think...

But what a mess! Logic and memory management tangled up!

Automatic storage management

PLs come with their own implementation of allocation; why not freeing too?

When can we free an object?

- When we can guarantee that it won't be used again in the computation (ground truth)
- ... when it isn't reachable (conservative approximation); this is garbage collection

Garbage Collection

Garbage collection: a way to know whether a record is *live* i.e., accessible

- 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

The World According to Garbage Collectors

Running programs are divided into two parts:

- **The collector:** manages the heap, allocates memory, collects garbage to free space
    - That's the part we'll be interested in
- **The mutator:** asks the collector for memory, does the work the program is supposed to do
    - In general programming, that's what we write
    - Now, it's secondary; mostly used for test cases

The collector provides functions, called by the mutator:

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

Learning Garbage Collectors, the PLAI Way

Two languages for learning garbage collectors:

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

Collectors implement a specific API, for use by mutators.

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

Collectors use an API provided by the collector language to access the heap and roots

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

Learning Garbage Collectors, the PLAI Way

Two languages for learning garbage collectors:

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

The mutator language ***transforms*** mutators to keep track of roots, make allocations explicit, and use the collector API

Mutators are regular* PLAI programs

- No need to use the (low-level) collector API directly!
- * some small differences, see the docs.

Learning Garbage Collectors, the PLAI Way

Two languages for learning garbage collectors:

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

(I can't stress enough how nice this is compared to the traditional way of learning GC.)

Rules of the game

- Our heap is a big vector, mapping addresses to values
- All values need to be allocated in the heap
- All values need to be ***tagged*** (to remember their type)
- Atomic values (fit in one cell in memory) include numbers (a lie), symbols (a less bad lie), booleans, and the empty list
- Compound values (require multiple cells in memory) include pairs and closures
- If an operation may allocate, its arguments must be in *roots* so we don't accidentally collect them.

A non-collecting collector

- Put the allocation pointer at address 0
- 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`**

A non-collecting collector

```
#lang plai/gc2/collector

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

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

A non-collecting collector, cont'd

```
(define (gc:flat? addr)
  (equal? (heap-ref addr) 'flat))

(define (gc:alloc-flat x)
  (define addr (alloc 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 @ ~a" addr))
  (heap-ref (+ addr 1)))
```

A non-collecting collector, cont'd

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

(define (gc:cons? addr)
  (equal? (heap-ref addr) 'cons))
```

A non-collecting collector, cont'd

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

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

(define (check-pair addr)
  (unless (equal? (heap-ref addr) 'cons)
    (error 'check-pair "not a pair @ ~a" addr)))
```

A non-collecting collector, cont'd

```
(define (gc:set-first! addr v)
  (check-pair addr)
  (heap-set! (+ addr 1) v))

(define (gc:set-rest! addr v)
  (check-pair addr)
  (heap-set! (+ addr 2) v))
```

A non-collecting collector, cont'd

```
(define (gc:closure code-pointer free-vars)
  (define addr (alloc (+ 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)
```

A non-collecting collector, cont'd

```
(define (gc:closure? addr)
  (equal? (heap-ref addr) 'clos))

(define (gc:closure-code-ptr addr)
  (unless (gc:closure? addr)
    (error "not a closure @ ~a" addr))
  (heap-ref (+ addr 1)))

(define (gc:closure-env-ref addr i)
  (unless (gc:closure? addr)
    (error "not a closure @ ~a" addr))
  (heap-ref (+ addr 2 i)))
```

Testing a collector

We can use `with-heap` to test a collector. The expression

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

expects `h-expr` to evaluate to a vector and then it uses that vector as the memory that `heap-ref` and `heap-set!` refer to while evaluating the `body-exprs`.

Testing our non-collecting collector

```
(let ([h (vector 'x 'x 'x 'x 'x)])
  (test (with-heap h
                   (init-allocator)
                   (gc:alloc-flat #f)
                  h)
        (vector 3 'flat #f 'x 'x)))
```

Testing our non-collecting collector

```
(let ([h (vector 'x 'x 'x 'x 'x 'x 'x 'x 'x)])
  (test (with-heap
         h
         (init-allocator)
         (gc:cons
          (simple-root (gc:alloc-flat #f))
          (simple-root (gc:alloc-flat #t)))
         h)
        (vector 8 'flat #f 'flat #t 'cons 1 3 'x)))
; (Of course, this is not enough testing.)
```

Testing with mutator programs

```
#lang plai/gc2/mutator
(allocator-setup "mygc.rkt" 200) ; heap size

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

; point to the same location
(test/location=? (rest c2) c1)
; produce the same value
(test/value=? (rest c1) '(3))
```

We can also use random testing to generate mutators.

A **`plai`** library generates code that makes interesting heap structures (randomly), and then makes up a traversal of them.

The next three slides give three example random mutators and the calls into the library that generated them.

Random mutators

```
#lang racket
(require plai/random-mutator)
(save-random-mutator "tmp.rkt" "mygc.rkt" #:gc2? #t)
```

```
#lang plai/gc2/mutator
(allocator-setup "mygc.rkt" 200)
(define (build-one)
  (let* ((x0 'x)
         (x1 (cons #f #f))
         (x2 (cons x1 #f))
         (x3
          (lambda (x)
            (if (= x 0)
              x0
              (if (= x 1)
                x2
                (if (= x 2) x2 (if (= x 3) x2 (if (= x 4) x1 x2)))))))
         (x4
          (lambda (x)
            (if (= x 0)
              x1
              (if (= x 1)
                x0
                (if (= x 2)
                  x1
                  (if (= x 3)
                    x2
                    (if (= x 4)
                      x1
                      (if (= x 5)
                        x2
                        (if (= x 6)
                          x3
                          (if (= x 7) x0 (if (= x 8) x0 x3))))))))))
         (x5 (lambda (x) (if (= x 0) x3 x0)))
         (x6 (lambda (x) x1)))
    (set-first! x1 x4)
    (set-rest! x1 x2)
    (set-rest! x2 x5)
    x4))
(define (traverse-one x4)
  (symbol=? 'x ((rest ((first ((first ((first (x4 4)) 0)) 0)) 3)) 1)))
(define (trigger-gc n)
  (if (zero? n) 0 (begin (cons n n) (trigger-gc (- n 1)))))
(define (loop i)
  (if (zero? i)
    'passed
    (let ((obj (build-one)))
      (trigger-gc 200)
      (if (traverse-one obj) (loop (- i 1)) 'failed))))
(loop 200)
```

Random mutators

```
#lang racket
(require plai/random-mutator)
(save-random-mutator "tmp.rkt" "mygc.rkt" #:gc2? #t)
```

```
#lang plai/gc2/mutator
(allocator-setup "mygc.rkt" 200)
(define (build-one) (let* ((x0 0)) x0))
(define (traverse-one x0) (= 0 x0))
(define (trigger-gc n)
  (if (zero? n) 0 (begin (cons n n) (trigger-gc (- n 1)))))
(define (loop i)
  (if (zero? i)
    'passed
    (let ((obj (build-one)))
      (trigger-gc 200)
      (if (traverse-one obj) (loop (- i 1)) 'failed))))
(loop 200)
```

Random mutators

```
#lang racket
(require plai/random-mutator)
(save-random-mutator "tmp.rkt" "mygc.rkt" #:gc2? #t)
```

```
#lang plai/gc2/mutator
(allocator-setup "mygc.rkt" 200)
(define (build-one)
  (let* ((x0 empty)
         (x1
          (lambda (x)
            (if (= x 0)
              x0
              (if (= x 1)
                x0
                (if (= x 2)
                  x0
                  (if (= x 3)
                    x0
                    (if (= x 4)
                      x0
                      (if (= x 5)
                        x0
                        (if (= x 6) x0 (if (= x 7) x0 x0))))))))))
    x1))
(define (traverse-one x1) (empty? (x1 0)))
(define (trigger-gc n)
  (if (zero? n) 0 (begin (cons n n) (trigger-gc (- n 1)))))
(define (loop i)
  (if (zero? i)
    'passed
    (let ((obj (build-one)))
      (trigger-gc 200)
      (if (traverse-one obj) (loop (- i 1)) 'failed))))
(loop 200)
```