Comp Sci 321: Programming Languages — Homework 8

1 Two-Space Collection

Implement a two-space copying collector, as described in lecture. Use four state variables in your collector: the allocation pointer, the active semi-space, and two pointers that you use only during collection, one pointing to the left and one to the right side of the queue for copying the data. (It is possible to do this with only three state variables, or even two, but I recommend four.)

You may assume that your heap size is a multiple of 2 and contains at least 12 cells.

2 Quality

You collector must be able to run these three programs without running out of space. Keep in mind, though, that these tests are not exhaustive; not running out of memory on these programs is necessary, but not sufficient, for getting full marks.

Each of these programs is a mutator. To run it, you should put them in their own file, and have them refer the file that implements your own garbage collector (in case it's not called `"gc.rkt"`).

```
#lang plai/gc2/mutator
(allocator-setup "gc.rkt" 400)
(define (count-down n)
  (cond
    [(zero? n) (count-down 20)]
    [else (count-down (- n 1))]))
(count-down 0)
```

```
#lang plai/gc2/mutator
(allocator-setup "gc.rkt" 400)
(define (mk-list n)
  (cond
    [(zero? n) '()]
    [else (cons n (mk-list (- n 1)))]))
(define (forever)
  (mk-list 10)
  (forever))
(forever)
```

```
#lang plai/gc2/mutator
(allocator-setup "gc.rkt" 400)
(define (proc-lst n)
  (cond
    [(zero? n) (lambda () 0)]
    [else (let ([n1 (proc-lst (- n 1))])
            (lambda () (+ (n1) n)))]))
(define (forever)
  ((proc-lst 10))
  (forever))
(forever)
```

You will want to supplement your testing with some random mutators (see `save-random-mutator`); again not sufficient, but certainly helpful.

3 PLAI Allocation: a No-No

Your garbage collector may not do any PLAI-level allocation (except stack space). This means that the functions `cons`, `vector`, `box`, `map`, and `list` (among others) are off-limits. Similarly, the constructors generated by `define-type` are off-limits (e.g., `fun`, `id`, `num`, `add`, and `sub` from earlier assignments). Closures with non-empty environments also require allocation, so you can only define functions at the top-level of the file (no `lambda` expressions in nested scopes).

The only exception: you may call `get-root-set`, which allocates at the PLAI level internally.

This means that your PLAI-level variables should be immutable and only hold flat values (integers, booleans, symbols), or should be module-level function definitions.

You may, of course, use functions that allocate in your tests to build up example heaps.

If you are unsure what primitives allocate memory, ask.

4 `plai/gc2` vs `plai`

You must use `#lang plai/gc2/collector` and `#lang plai/gc2/mutator`. The documentation explains what the primitives are and what functions you have to implement, but be careful that you read the versions with a `gc2` in the name, not `plai/mutator` or `plai/collector`.

To find the right docs, open up the docs and click the up link at the top of the page until you get to a page named "Racket Documentation". From there, click "Programming Languages: Application and Interpretation" and then either "GC Collector Language, 2" or "GC Mutator Language, 2".

5 Tips

You may need to do a lot of debugging. Here are a few tips to make your life easier.

- Write your heap checker (`validate-heap`) first, and call it often. A well-written heap checker can catch corruption shortly after it happens, and (most importantly) before it snowballs out of control.
- Bugs in your garbage collector only trigger when your garbage collector is actually run (duh). With a normal-sized heap, this can take some fairly large test cases, which do a lot of allocation. Here are two ways to get your garbage collector to fire with smaller test cases:
  - Use a smaller heap. Smaller heaps take fewer allocations to fill up, so your garbage collector will fire more often / with smaller test cases.

  - Modify your allocator to run your garbage collector every allocation, rather than only when the heap is full. In GC parlance, this is referred to as *stress mode*. This is extremely inefficient (so you don't want it turned on in production!), but it will trigger your garbage collection as often as it can. Which in turn means that pinpointing the source of any corruption should be a lot easier.

- If you insert calls to `read-line` in the middle of your collector, upon reaching that point DrRacket will pause until you hit return in the interactions window (but will still update the heap GUI). This allows you to see what is happening during collection (or allocation) and thus can help you understand what is going on.
- A very common class of bugs in garbage collectors is forgetting to traverse and/or update roots. Inspecting your code with an eye for that may find bugs.
- Most of all, when you debug, *be systematic*! Debugging by Brownian motion may kinda sorta work in some contexts, but it's a recipe for tears with any sort of memory management code (like, say, a garbage collector). When debugging, formulate hypotheses, gather data to confirm or deny them, and repeat until you find the problem. When you make a change, *always* have a theory as to *why* and *how* this will affect what you see. And if that theory doesn't hold, formulate a new one and keep going.

6 Template

Below is a list of the functions required to implement a garbage collector, along with bogus implementations. See the docs for a better description of each of these functions (but beware that you find the `gc2` versions of these functions).

```
#lang plai/gc2/collector

(define (init-allocator) (error 'this-collector-always-fails))
(define (gc:deref fl-loc) (error 'gc:deref))
(define (gc:alloc-flat fv) (error 'gc:alloc-flat))
(define (gc:cons hd tl) (error 'gc:cons))
(define (gc:first pr-loc) (error 'gc:first))
(define (gc:rest pr-loc) (error 'gc:rest))
(define (gc:flat? loc) (error 'gc:flat?))
(define (gc:cons? loc) (error 'gc:cons?))
(define (gc:set-first! pr-ptr new) (error 'gc:set-first!))
(define (gc:set-rest! pr-ptr new) (error 'gc:set-rest!))
(define (gc:closure code-ptr free-vars) (error 'gc:closure))
(define (gc:closure-code-ptr loc) (error 'gc:closure-code-ptr))
(define (gc:closure-env-ref loc i) (error 'gc:closure-env-ref))
(define (gc:closure? loc) (error 'gc:closure?))
```

7 Errors

Your garbage collector should throw errors in two kinds of situations:

- The program runs out of memory. The mutator needs more space than is available in the heap, even after garbage collection. When this happens, you should raise an error that contains the string `"out of memory"`.
- One of your functions is passed unexpected data. E.g., `gc:deref` being passed a non-`flat` value. We do not impose constraints on your error messages for such cases.

8 Handin

Hand in your garbage collector. It should begin with the line:

```
#lang plai/gc2/collector
```

PLEASE:

- Take out / comment out any debugging printing *before* you turn in; no one likes getting gigabytes of debugging output when grading.
- Comment out calls to your heap checker; otherwise, your garbage collector will likely time out during grading.
- Don't leave your garbage collector in "stress mode"; for the same reason.

Have the 8 rules from the Provost's website (see homework 1 for more details).
Submit your code via Canvas.
You do not need to submit test cases for this assignment.