Memory Management - Background

- Most programs allocate memory as they run
  - Scheme: cons, lambda (allocates closure)
  - Java, Smalltalk: new
- Programs may run out of space if memory is not reused.
- Even if program does not run out of space, programs using compact space run faster (due to virtual memory and cache)

Memory Management - Background

- Memory management:
  mechanism to claim + release memory
- Claiming/allocating memory is obvious:
  when program says to (e.g. malloc, new, cons, ...)
- Releasing memory is less obvious:
  when is it safe to release memory?

Memory Management - Background

- When is it <u>safe</u> to release memory?
  - when the object is no longer useful
  - when the program will never access the object again in the future

- Note: Determining if an object will be accessed again in the future is <u>undecidable</u>; "perfect" GCs don't exist

```
(val x '(1 2 3))
(if (f 0)            ; can x be reclaimed before calling f
    (... x ...)
    #f)
```

Memory Management – Background

- <u>Manual</u> Mem. Mgmt. (e.g. C)
  - reclaim space for local variables, when execution leaves the function/block } stack
  - reclaim space for heap objects when programmer requests (e.g. free) } heap
- <u>Automatic</u> Mem. Mgmt. (e.g. Java, Standard ML)
  - reclaim space for heap objects when language implementation determines it is <u>safe</u> to do so. (using a conservative approximation)
- <u>Semi-Automatic</u> Mem. Mgmt (e.g. Rust, C++ (idioms))
  - bulk free of heap objects at well-defined scope

Memory Management - Background

- Manual
  - + easy for lang. implementer
  - + programmer in full control (agressive optimization)
  - − bugs, bugs, and more bugs
    - memory leak: forget to call free
    - double free: call free twice on some address
    - use after free: access object after calling free (it wasn't safe to reclaim)
    - use after free: access object after calling free and the memory has been reused for a new object
  - − security implications

Memory Management - Background

- Automatic
  - Garbage collection: language implementation automatically reclaims unused memory

  − harder for lang. implementer

  − programmer has little/no control

  + gives the illusion of infinite memory

  + relieves programmer of mem. mgmt. burden

  + no mem. mgmt. bugs (?)

  − some performance overheads

Garbage Collection - Reachability

- Use conservative approx. of
  when an object will never be accessed again

→ object cannot be accessed ⇒ object will not be accessed

→ object not reachable ⇒ object can not be accessed

- Reachability (specification)
  - globals (top-level bindings, static-fields) are reachable
  - local variables from function calls that haven't returned are reachable (i.e., the stack is reachable)
  - any object referred to / pointed to by a reachable object is reachable
  - nothing else is reachable

Garbage Collection - Reachability

- Reachability (implementation)
  - "crawl" the globals and stack to get roots
  - recursively follow all fields of reachable objects, but don't recur on objects already seen

- Devil is in the details
  - crawling stack and following fields requires intimate knowledge of / help from the lang. implementation } GC cannot be implemented as a library
  - garbage collectors must be efficient (time + space); utilize various "tricks"

Garbage Collection - Space Leaks

- in manual mem. mgmt., space leak refers to "unreachable heap objects that were not reclaimed" (and, unreachable implies will never be reclaimed)
- a GC reclaims (all) unreachable objects, so many say "a lang w/ GC cannot have space leaks"

? agree or disagree ?

- technically true w/ above defn of space leak but a bit misleading

and false for a broader defn of space leak

Example: store a (pointer to a) huge data structure in a static field of a Java class. Never access that field again.

Garbage Collection - Space Leaks

Example: store a (pointer to a) huge data structure
in a static field of a Java class.
Never access that field again.

In general, a GC won't reclaim any reachable object,
(and static fields are globals).

Options

- ignore the issue; rare in practice
  (but, I spent days in Fall fixing space leaks in MLton)
- set fields to null (a form of manual mem. mgmt.)
- be careful to not let "permanent" data get too big
- use "weak pointers"

Garbage Collection - Performance Metrics

- When evaluating GCs,
  many aspects affect performance:

- pause time
  - ↳ stop program for mem. mgmt. tasks
    - → soft deadlines: UIs, games, ...
    - → hard deadines/realtime: medical, air-traffic, nuclear, ...

- heap size (H)
  - ↳ total memory being used by mem. mgmt.
  - → live data (L) - reachable objects
  - → available space (H - L) - mem. for new objs. before GC
  - → ratio: $\gamma = H/L$ (note: $\gamma \geq 1$)

Garbage Collection - Performance Metrics

- allocation cost
  - ↳ time to allocate a new object (i.e., time to find + use available space)
- overhead
  - ↳ cost added to program for mem. mgmt.
    - time: alloc. cost, GC. cost, ...
    - space: memory needed for mem. mgmt.

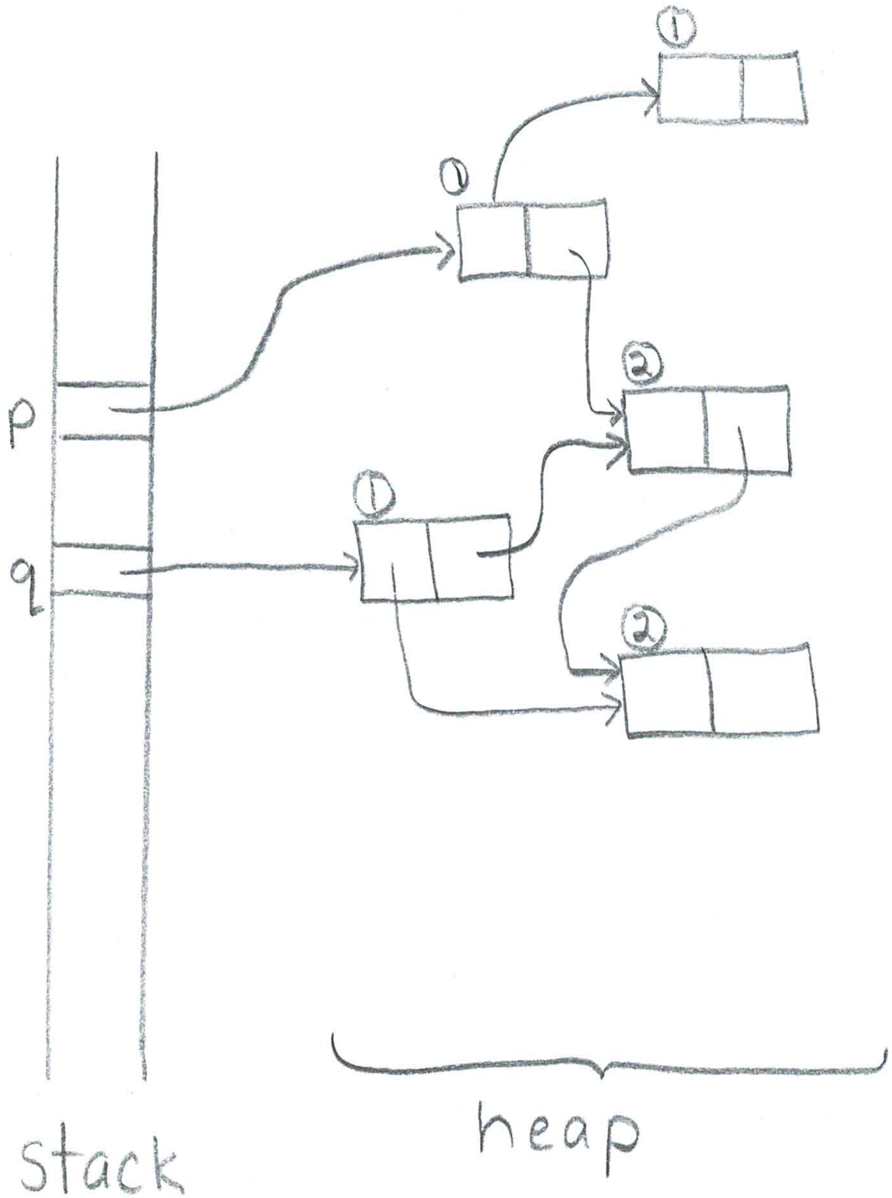
      often, need metadata for each object

      ↳ data used by mem. mgmt, but hidden from programmer

Reference Counting

- (not considered a "true GC" by some)
- uses a slightly different conservative approx.
  - → if no references/pointers to an object, then the object can not be accessed and can be reclaimed
- associate a "<u>reference count</u>" with each object
  - ↳ count of references/pointers to object (from other objects and locals/globals)
- when reference count becomes 0, then reclaim the object.

Reference Counting



What happens when program executes these operations:

- dec(p);
  p = new();

- inc(q);
  dec(p);
  p = q;

  } why would dec(p); inc(q); be incorrect?

- dec(p);
  p = NULL;

  } when does this happen implicitly?
  Ans: when p goes out of scope, like at fn ret.

Compiler inserted operations to maintain ref counts.

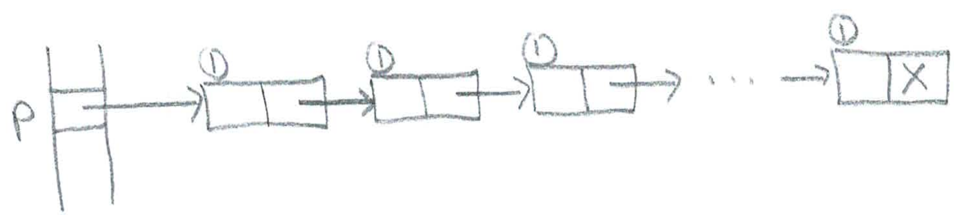
Reference Counting - Pseudo-code

- inc(p) { p→refcnt = p→refcnt + 1; }

- dec(p) {
```
dec(p) { p→refcnt = p→refcnt - 1;
         if (p→refcnt == 0) {
           foreach f in fields(p) {
             dec(p→f);
           }
           free(p);
         }
       }
```

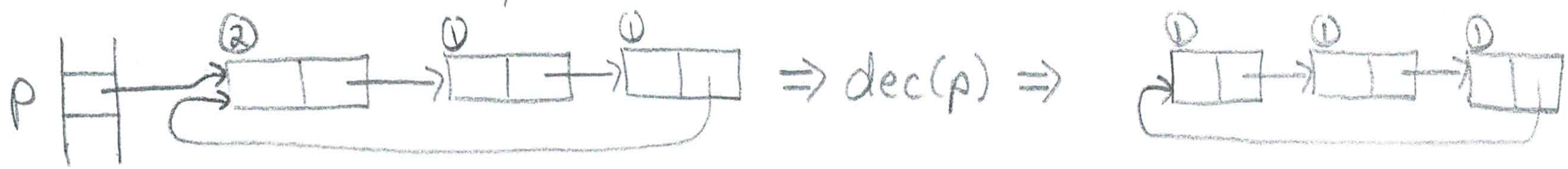
Reference Counting - Analysis

- What is allocation cost?
  - moderate, usually via a "free list" (b/c objects freed individually)
- Per object overhead:
  - refcnt field
  - must be able to enumerate (pointer) fields
- What is running time of dec(p)?
  - best: O(1)
  - worst: O(r) where r is size of objects reachable from p

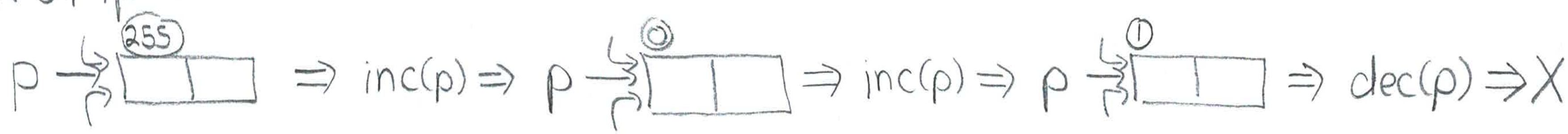


Reference Counting - Analysis

- Limitations
  - cannot collect cycles



  - recursive dec function consumes stack space
    - GC shouldn't use (lots of) space
  - refcnt field can overflow
    - example: 8-bit refcnt field



    - solution: make max. refcnt "sticky"
      - ↳ inc or dec of 255 keeps value at 255
      - ↳ never reclaims such objects or objects reachable from such objects