CS 461

Programming Language Concepts

Gang Tan
Computer Science and Engineering
Penn State University

The Heap

- Two routines provided by a memory manager
  - p=new(n): allocate a memory region of size n
  - delete(p): return the memory region to the manager
- The mem manager tracks
  - what regions have been allocated and their sizes
  - what regions are available for allocation
    - free list
- *Heap overflow* occurs when a call to *new* occurs and the heap does not have a large enough block available to satisfy the call
  - May happen due to fragmentation

Memory Management in PLs

- C/C++: manual memory management
  - programmers manage memory: malloc/free, new/delete
  - pros: efficient/flexible
  - Cons: error prone; dangling pointers/free twice; memory leak; security risks
- Java/Lisp/Scheme: automatic memory management (GC)
  - No "delete" operations for programmers
    - GC collects memory of objects that are no longer used
  - pros: programmer convenience (programming at a high level)
    - easier to get the program right; no problem such as dangling pointers
  - cons: inflexible; GC may not be suitable for real-time systems

Scheme Memory Cells

Scheme S-Expressions

- Symbolic expressions: either an atom or a list
- Atoms
  - symbols, e.g., A, append, name, etc.
  - numeric literals, e.g., 1, 3.14, etc.
  - Booleans, #t or #f
- Lists: elements within parenthesis
  - e.g., '(A B C D)
  - Can be nested; e.g., '(A (B C (D E) F) (G H))

Internal Representation of S-Expressions (all pointers)

- Everything represented by cells
- Atom cells

atom	a

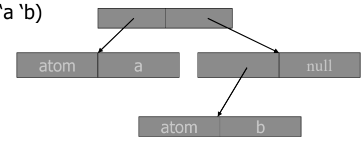
- Cons Cells: for general dotted pairs

car	cdr

  - both car and cdr are pointers to other cells
  - there is a special value for null pointers

Examples

- Atoms and lists represented by cells
  - '(a b) = (list 'a 'b)



Storage Allocation for Lists

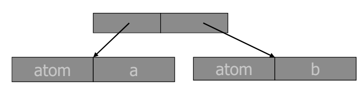
- In Scheme, storage allocation is implicit
- For atoms: when Scheme encounters an atom , it allocates an atom cell
- Storage allocation for (cons e1 e2)
  - allocate a new cons cell c
  - allocate cells for e1 and e2 to represent them in mem
  - set the car part of c to point to the representation of e1
  - set the cdr part of c to point to the representation of e2
  - return a pointer to c
  - example: '(a b) = (cons 'a (cons 'b '()))

Implementation of List Operations

- Implementations of car, cdr, null?, pair?
  - (car x): if x points to a cons cell, return the car part; otherwise error
  - (cdr x): if x points to a cons cell, return the cdr part; otherwise error
  - null?: returns true if it's a null pointer
  - pair?: returns true if it points to a cons cell
  - atom? = not pair? and not null?

The representations can represent more than lists

- (cons 'a 'b): not a list, but can still be represented
  - The representation can in general represent binary trees



- This is in general called dotted pairs
- This can be used to represent, for example, trees
- list? decides whether something is a proper list: keep applying cdr will always returns a list
  - (list? (cons 'a 'b)) = #f

Garbage Collection (Ch 8.5.3)

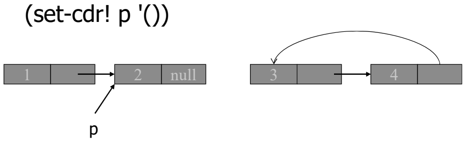
Garbage Collection

- Garbage:

  At a given point in the execution of a program *P*, a memory location *m* is *garbage* if no continued execution of *P* from this point can access location *m*.
- Scheme: automatic garbage collection
  - memory: a list of cells
    - A free list tracks what the unused cells are
  - When the running program needs new cells, they are from the free list
  - When the free list is below some threshold, the run-time system performs GC to reclaim garbage cells
    - Detect garbage during program execution

A Running Example

(define p '(1 2 3 4))
(define p (cdr p))
(set-cdr! (cddr p) (cdr p))
(set-cdr! p '())



14

Mark-and-Sweep Algorithm (BFS or DFS)

- Need
  - One tag bit for each cell
  - List of memory cells directly accessible to the program
    - Called the root set
- Algorithm
  - Initialization: set all tag bits to 0.
  - First pass (mark): start from each cell in the root set; follow all pointers; change tag bits of reachable cells to 1
  - Second pass (sweep): place all cells with tag = 0 on the free list
- Pros: straightforward
- Cons: two passes; need to stop the program from running; poor real time performance

Reference Counting

- Each cell has a reference count (RC).
  - It records the number of references that point to the cell
- Initialize the reference counts when a data structure is first created
- For an operation involving pointers, like (define q p)
  - The reference count for p's node is increased by 1
  - the reference count for q's node is decreased by 1
    - If it becomes 0, reclaim it to the free list and decrease the reference count of anything it points to; maybe a snowball effect
  - q's value get p's value

But not all garbage is collected…

- Cons
  - Cannot reclaim circular data structures
  - Space overhead
  - Runtime overhead for reference account adjustments
- Pros
  - GC work is divided
    - Occurs whenever mem is allocated or during pointer assignments
  - better real-time performance

Copy Collection

- Heap partitioned into two halves
  - From space, to space
  - Only the from space is active
- All allocation is performed in the from space
- When GC starts,
  - Reachable nodes in the from space copied to the to space
  - Swap the from and the to space
  - Note: The accessible nodes are packed, orphans are returned to the free_list, and the two halves reverse roles.
- Pros: Faster than mark-sweep; reduce storage fragmentation
- Cons: reduces the size of the heap space

Garbage Collection Summary

- Modern algorithms are more elaborate.
  - Most are hybrids/refinements of the above three.
- Functional languages have garbage collection built-in
  - Rely on GC to destroy old values
- In Java, garbage collection is built-in.
  - runs as a low-priority thread.
  - Also, System.gc may be called by the program
- C/C++ default garbage collection to the programmer.