Memory Management Overview

- How is memory allocated?
- Notion of equality
- How is memory deallocated?
  - When to deallocate?
  - Algorithms to deallocate

This is section 10.6 of Sethi.

Memory Allocation

• Each list can be represented as a sequence of applications of `cons`:

```
'(1 2) = (cons 1 '(2)) = (cons 1 (cons 2 '()))
```

• Operator `cons` ends up creating a cons pair, with a pointer to head and a pointer to tail:

So, what happens here?

```
(define x '(1 2 3)
```

```
(define y (cons (car x) (cdr x)))
```


Notion of Equality

Three operators: `eqv` (will not cover), `equal?` and `eq?`.

- all three have the same results when applied to symbols:

  `(eqv?/eq?/equal? 'hello 'hello)` ⇒ `#t`

- `eq?` tests that the two operators have identical pointers
- `equal?` recursively checks that the two arguments are lists with equal elements. May fail on circular data structures.

Example:

```
(equal? 'hi 'hi) ⇒ #t
(eq? 'hi 'hi) ⇒ #t
(equal? '(hi bye) '(hi bye)) ⇒ #t
(eq? '(hi bye) '(hi bye)) ⇒ #f
```

Why?

Allocation and Deallocation

- Cells not in use have to be recovered, otherwise program runs out of memory.
- A cell is no longer in use if nothing points to it
- *garbage collection* - reallocates memory without explicit instructions

Technique - have a free list - stack of cells
  - A pop returns a freshly allocated cell
  - A push returns a cell back onto the stack

Examples

```
'( 1 )


(let ((x '( 1 ))) (x))


(define x '( 1 ))
x
```


Approaches to cell deallocation

Lazy approach:
- Wait until memory runs out and only then collect dead cells.
- Advantage - if enough memory is available, the need for collecting cells may never arise
- Disadvantage - once garbage collecting starts, all other work comes to a halt. Example: emacs.

Eager approach:
- Each time a cell is reached, check whether the cell will be needed after the operation; if not, deallocate the cell by placing it on the free list.
- Advantage - work does not come to a halt
- Disadvantage - to implement it, need space.

Implementing Eager Approach

- For each cell, store *reference count* - the number of pointers to the cell. - When a new pointer appears, increment count
- When count becomes zero:
  - Decrement reference count of cell pointed to by `cdr` and continue recursively
  - Free cell

Example:

Implementation of Garbage Collection

Mark-Sweep approach
1. *Mark phase*. Start from all global variables. Mark all the cells that can be reached by following pointers.
2. *Sweep phase*. Sweep (scan) through memory, starting at "beginning", looking for unmarked cells. Unmarked cells are returned to the free list. Unlike mark phase, this starts at one end of memory and looks at every cell.

Example:

Optimization of Mark-Sweep Algorithm

This is called *copying collector*.
- Divides memory into two halves, the *working* and the *free*
- Allocates cells from working half
- When no more cells in working, copies all reachable cells into consecutive locations in the free half.
- Switches roles of working and free halves

Advantage: No need to to do sweep part of the mark-sweep algorithm. Just copy reachable cells (those marked by marking phase of the algorithm).

Very effective in a virtual memory system

Summary

Programming in untyped functional language Scheme.

- History and Introduction
- Primitive and compound expressions
- Local and Global Definitions
- Lists and their manipulation
- Recursive reasoning
- Expression evaluation
- Proofs of correctness
- Higher-order programming
- Miscellaneous stuff:
  - Association lists
  - Functions with arbitrary number of parameters
- Memory management

Functional Programming - Review

- Do away with variables and reassignments (referential transparency)
- No notion of state (in pure functional programming)
- Declarative rather than imperative approach
- Allows to analyze programs more easily

Software engineering issues:
- want to construct programs more reliably.
- OO programming: better code faster. That's good but we want BEST, i.e., no bugs.
- want to do away with testing. Testing can prove presence of bugs rather than their absence. Want to replace testing with reasoning

Functional Programming - Review (Cont'd)

Functional programming allows to develop programs which submit to logical analysis relatively easily.
- Can make assertions about programs and prove them correct.
- All datatypes have constructors and base cases, so can reason by induction
- Allows to write "better" programs
- Can write good code in imperative language (hard) or bad code in functional languages

Problems:
- Slow ("efficiency does not matter")
- Primitive user interface

Functional vs Imperative programming

The development cycle:

Imperative:
- Using informal language a specification may be open to interpretation.
- Using appropriate testing strategies we can improve confidence, but not in any measurable way.
- Mistakes/bugs are common and difficult to spot and correct.

Functional:
- Using logic, we can state the specification exactly.
- Using mathematics we may be able to prove useful properties of our programs.

Functional vs Imperative Programming (Cont'd)

The Development Language:

Imperative:
- Using structured programming or object oriented techniques we can reuse code.
- Can partition the program into more manageable chinks.

Functional:
- Using structured programming or OO techniques we can reuse code.
- Can partition the problem into easy to use chunks. These are often "higher-level" abstractions. Can write things in Scheme which are difficult or impossible to write in a traditional language

Functional vs Imperative Programming (Cont'd)

The Run-time System:

Imperative:
- The compiler can produce fast compact code taking a fixed amount of memory
- Parallel processing is not possible, in general
- Fancy GUI's may be added

Functional:
- Interpretors are available
- The memory requirements are large and unpredictable
- Parallel processing is possible
- Fancy GUI's may be added, although with great difficulty

food for slide eater