Garbage Collection

Garbage Collection, Outline

- Mark and Sweep Garbage Collectors.
- Copying Garbage Collectors.
- Generational Garbage Collectors.
- Reference Counting.

Garbage Collection

Scheme, Java, Smalltalk, Miranda, and CLP(R) all are *garbage collected*.

Garbage collection: system manages memory for you:

- No need to free memory.
- System detects inaccessible memory and frees it.
  (Consider the recursive algorithm)

Mark-and-Sweep

When memory is exhausted:

- Traverse and *mark* all *reachable* memory.
- *Sweep* through memory freeing everything not marked.

What is reachable memory?

- Every object in call stack is reachable.
- Every object pointed to by reachable object is reachable.

Example

Given definitions:

```
(define x (cons 5 '()))
(define y '(a b))
(define z '(4))
(set-cdr! z z)
```

Which `cons` cells are pointed to by variables in the stack?
(the stack now is just the global environment)

```
(set! x 10)
(set! y (cdr y))
```

What does the *heap memory* look like now?
Perform the marking algorithm.

Mark-and-Sweep Discussion

Slight problem: mark-and-sweep is recursive, needs stack space, but we are out of space!

Each allocated piece of memory must have a *mark* bit.

Runtime of algorithm is linear in the heap size (Mark and Sweep touches all allocated memory!).

Memory gets fragmented.

Copying Collectors

Divide memory in to halves: *old* and *new*.

Current heap is *old*.

When we are out of space in *old*:

- Traverse all reachable memory copying it into *new*.
- Swap which half is *old* and *new*.

Details: When traversing reachable object *obj*:

- Copy *obj* to next available slot in *new*.
- Replace *obj* in *old* with *forwarding pointer* to *obj* in *new*.
- Examine all of *obj*'s pointers:
  - If pointer to other object, recursively traverse it.
  - If pointer to forwarding pointer, then adjust to point directly to object that forwarder points to.

Example

Given definitions:

```
(define x (cons 5 '()))
(define y '(a b))
(define z '(4))
(set-cdr! z z)
```

Which `cons` cells are pointed to by variables in the stack?

```
(set! x 10)
(set! y (cdr y))
```

What does the *heap memory* look like now?
Run the copying collector.

Copying Collector Discussion

No need for mark bit.

Runs in time proportional to amount of *reachable* memory.

Only half of memory is available for use at one time.

Gets *compaction* for free.

Works well with *virtual memory*.

Generational Collectors

Observation: Memory is used in many different ways

- Some memory is used briefly, then discarded.
  E.g. Nodes in queue (linked list implementation).
- Some memory is in use for a long time.
  E.g. A hashtable.

Can we take advantage of this?

Divide heap memory into regions for generations: *young*, ..., *old*.

Perform garbage collections on *young* more often than *old*.

If an item survives a certain number of garbage collections, promote it to the next generation.

Generational GC Notes

In pure functional languages, objects can only contain pointers to older objects.

Thus, any pointer that passes over a memory region boundary goes from a younger region to an older region.

Incremental Garbage Collectors

All GCs discussed interrupt program to collect garbage.

This is undesirable in real-time applications.

Incremental GCs collect garbage as the program is running.

Sometimes they are run on a second processor.

Reference Counting

Reference counting can also be used for garbage collection.

It is inherently *incremental*.

- Each object in memory keeps track of how many references there are to it.
- When reference count gets to 0, we can free the object.

Example

Given definitions:

```
(define x (cons 5 '()))
(define y '(a b))
(define z '(4))
(set-cdr! z z)
```

What is the reference count of each `cons` cell?

Simulate running the following using reference counting:

```
(set! x 10)
(set! y (cdr y))
```

Any problems?

Reference Counting Notes

Adds overhead to every pointer operation.

Does not correctly collect cyclic data structures.

It is incremental.

Can by implemented by the programmer.

GC Conclusions

Advantages:

- Lets programmer not worry about memory management.
- Cleaner code.
- Easier to share data.

Disadvantages:

- Programmer can write more efficient memory management code than GC.
- GC disrupts program.
- Must be able to distinguish pointer data from non-pointer data.