Garbage Collection

Deep Ghosh
Garbage: heap-allocated records that are no longer needed

- memory occupied by garbage should be reclaimed:
  1) require programmer to explicitly "free" memory
  2) invoke run-time system garbage collection program

Compiler cannot always tell whether a heap-allocated record will be needed in future

- conservative approximation: if record not reachable from program variables by chain of pointers, then record is garbage
<table>
<thead>
<tr>
<th>Manual Garbage Collection</th>
<th>Automatic Garbage Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ High efficiency</td>
<td>+ Reduces programmer burden</td>
</tr>
<tr>
<td>+ Close programmer control</td>
<td>+ Eliminates sources of errors</td>
</tr>
<tr>
<td>− More code to maintain</td>
<td>− May hurt performance</td>
</tr>
<tr>
<td>− Correctness difficult</td>
<td>− Cannot determine all objects that won’t be used in future</td>
</tr>
</tbody>
</table>
Overview

• Reference counting garbage collection
• Mark-and-sweep garbage collection
• Copying garbage collection
Reference Count Collection

- Mark and sweep collection identifies garbage by performing DFS
- Can identify garbage directly by keeping track of how many pointers point to each record
  → reference count of record, stored in record

Given: \( x \leftarrow p \), \( x \) is program variable or record field

- Compiler emits code to perform following:
  1) increment reference count of \( p \)
  2) decrement reference count of record \( r \) that \( x \) previously pointed to
  3) if reference count of \( r = 0 \), \( r \) is put on free-list, all records pointed to by \( r \) have their reference counts decremented
Two problems:

1) Incrementing and decrementing reference counts expensive

   - Instead of generating 'x ← p', compiler must now generate

     \[
     \begin{align*}
     z & \leftarrow x \\
     c & \leftarrow z\.count \\
     c & \leftarrow c - 1 \\
     z\.count & \leftarrow c \\
     \text{if } c = 0 \text{ then putOnFreeList}(z) \\
     x & \leftarrow p \\
     c & \leftarrow p\.count \\
     c & \leftarrow c + 1 \\
     p\.count & \leftarrow c
     \end{align*}
     \]
2) Impossible to reclaim cycles of garbage

program vars

heap

a

\[ \text{count} = 1 \]

\[ \text{count} = 2 \]

b

\[ \text{count} = 1 \]

\[ \text{count} = 1 \]

these two records are garbage, but each has reference count of 1
two solutions to this problem:

- require programmer to explicitly break all cycles when done
- perform occasional mark-sweep collection to reclaim cycles of garbage
Mark and Sweep Collection

- program variables + heap-allocated records form directed graph
  - roots: program variables
  - internal nodes and leaves: heap-allocated records
  - edges: pointers
- mark phase: perform depth-first search (DFS) from each root node, marking all nodes reachable from root
  - heap-allocated record n reachable from root r if path r\rightarrow n_1\rightarrow n_2\rightarrow\ldots\rightarrow n exists
  - any node not marked must be garbage
- sweep phase: scan through entire heap, collecting onto a free-list all unmarked nodes
  - all nodes subsequently unmarked for next collection
- After garbage collection, compiled program resumes execution
- when run-time support function allocRecord called to allocate size n record,
  free-list checked for record of size n
  - if record exists, then return it
  - else replenish free-list by performing garbage collection
Example

Initial

program vars

heap

free-list

a

b
After Mark Phase

program vars

heap

1
2
3
4
5

free-list
Copying Collection

- Heap divided into two regions: **from-space** and **to-space**
  - all records allocated in from-space
    - from-space is fragmented - garbage interspersed with reachable data
  - when from-space full, copying garbage collection copies all reachable records into to-space
    - to-space copy occupies contiguous memory -> compact
  - roots point to to-space copy, entire from-space made unreachable
  - to-space becomes from-space, from-space becomes to-space
- **Forwarding**: main operation used during copying collection
  - given a pointer that points to from-space, make it point to to-space
    1) if pointer p points to from-space record that hasn’t been copied, then:
       - copy record into to-space
       - make first field of record pointed to by p (p.fi) point to copy
         ⇒ forwarding pointer
       - return pointer to to-space copy
    2) if pointer p points to from-space record that has been copied, then:
       - return p.fi (pointer to to-space copy)
- Cheney's Algorithm: performs copying collection using BFS

Example

**Before Collection**

**Roots Forwarded**
How does compiler interact with garbage collector?

1) generates code to invoke run-time support function allocRecord when record must be heap-allocated
   - allocRecord will invoke garbage collector, if necessary

2) describes locations of directed graph roots, in preparation for next garbage collection

3) describes layout of records on heap
In order to determine which heap-allocated records are reachable, collector must know size of each record + location of pointer fields

- let first word of every record be pointer to type-descriptor record

- type-descriptor record generated by compiler during semantic analysis
  → pointer to record passed as argument to allocaRecord

- Compiler must also identify all "active" program variables that are pointers
  - may be located on stack or register
  - used as roots during mark-sweep or copying garbage collection