

CSc 520

Principles of Programming Languages

39: Garbage Collection — Mark and Sweep

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2005 Christian Collberg

—Spring 2005—39

[1]

Garbage Collection Issues

520—Spring 2005—39

[2]

Finding the Object Graph

Finding the roots: The dynamic objects in a program form a **graph**. Most GC algorithms need to traverse this graph. The **roots** of the graph can be in

1. global variables
2. registers
3. local variables/formal parameters on the stack.

Hence, the compiler must communicate to the GC which registers/variables contain roots.

—Spring 2005—39

[3]

Finding the Object Graph...

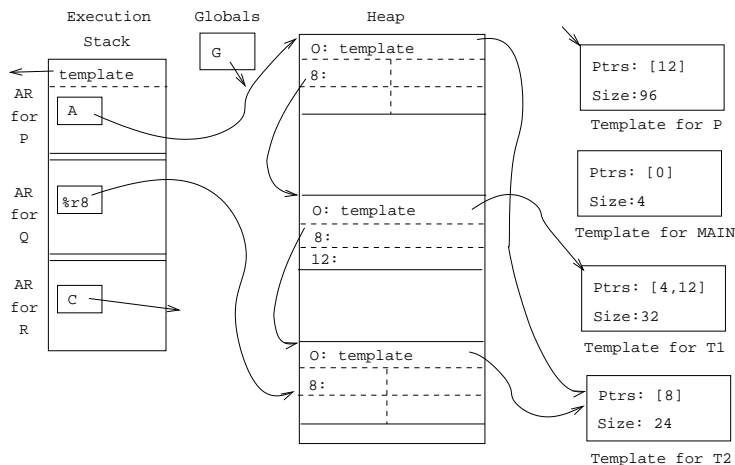
Finding internal pointers: Structured variables (arrays, records, objects) may contain internal pointers. These must be known to the GC so that it can traverse the graph. Hence, the compiler must communicate to the GC the **type** of each dynamic object and the internal structure of each type.

Finding the beginning of objects: What happens if the only pointer to an object points somewhere in the middle of the object? We must either be able to find the beginning of the object, or make sure the compiler does not generate such code.

520—Spring 2005—39

[4]

Finding the Object Graph...



Pointer Maps

- The internal structure of activation records & structured variables is described by run-time templates.
- Every run-time object has an extra word that points to a *type descriptor* (or *Template*), a structure describing which words in the object are pointers. This map is constructed at compile-time and stored statically in the data segment of the executable.

Pointer Maps...

- When the GC is invoked, registers may also contain valid pointers. The compiler must therefore also generate (for every point where the GC may be called) a *pointer map* that describes which registers hold live pointers at this point. For this reason, we usually only allow the GC to run at certain points, often the points where **new** is called.
- We must also provide pointer maps for every function call point. A function *P* may call *Q* which calls **new** which invokes the GC. We need to know which words in *P*'s activation record that at this point contain live pointers.

Pointer Maps...

- How does the GC look up which pointer map belongs to a particular call to procedure *P* at a particular address *a*? The pointer maps are indexed by the return address of *P*! So, to traverse the stack of activation records, the GC looks at each frame, extracts the return address, finds the pointer map for that address, and extracts each pointer according to the map.

The Mark-and-Sweep Algorithm

Algorithm: Mark and Sweep

- The basic idea behind **Mark-and-Sweep** is to traverse and mark all the cells that can be reached from the **root cells**.
- A **root cell** is any pointer on the stack or in global memory which points to objects on the heap.
- Once all the live cells (those which are pointed to by a global variable or some other live cells) have been marked, we scan through the heap and separate the live data from the garbage.
 - If we are dealing with equal size objects only (this is the case in LISP, for example) then we scan the heap and link all the unmarked objects onto the free list. At the same time we can unmark the live cells.

Algorithm: Mark and Sweep...

- ● If we have cells of different sizes, just linking the freed objects together may result in heap fragmentation. Instead we need to compact the heap, by collecting live cells together in a contiguous memory area on the heap and doing the same with the garbage cells in another area.

Algorithm: Mark and Sweep...

Marking Phase:

1. Mark all objects **unmarked**.
2. Find all roots, i.e. heap pointers in stack, regs & globals.
3. Mark reachable blocks using a **depth first search** starting at the roots.
 - (a) DFS may run out of stack space!
 - (b) Use non-recursive (Deutsch-Schorr-Waite) DFS.

Scanning Phase:

- same-size-cells** Scan heap and put un-marked (non-reachable) cells back on free-list.
- different-size-cells** Compact the heap to prevent fragmentation.

Marking Phase

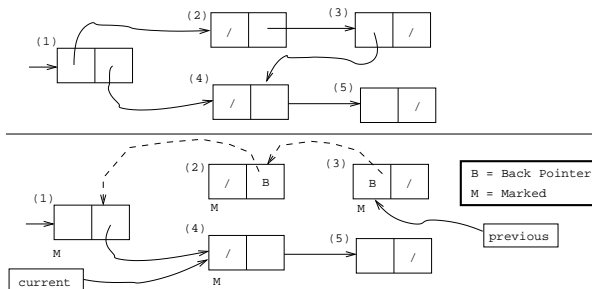
- A straight-forward implementation of mark and sweep may run into memory problems itself! A depth-first-search makes use of a stack, and the size of the stack will be the same as the depth of the object graph.
- Remember that the stack and the heap share the same memory space, and may even grow towards each other.
- So, if we're out of luck we might run into this situation:
 - the heap is full (otherwise we wouldn't be gc:ing!),
 - the object graph is deep,
 - we run out of stack space during the marking phase.We're now out of memory altogether. Difficult to recover from!

Marking Phase...

- Fortunately, there is a smart algorithm for marking in constant space, called the **Deutsch-Schorr-Waite algorithm**. Actually, it was developed simultaneously by Peter Deutsch and by Herbert Schorr and W. M. Waite.
- The basic idea is to store the DFS stack in the object graph itself.
 1. we set the “marked”-bit to 1,
 2. the node (object) is made to point to the previous node,
 3. two global variables `current` and `previous` are updated.`current` points to the current cell, `previous` to the previously visited cell.

Marking: “Look Ma, No Stack!”

- Use **pointer reversal** to encode the DFS stack in the object graph itself.
- When the DFS reaches a new cell, change a pointer in the cell to point back to the DFS parent cell. When we can go no deeper, return, following the back links, restoring the links.



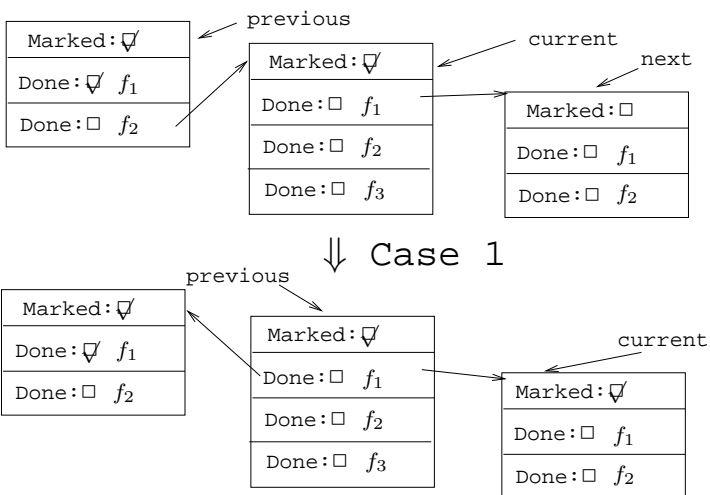
Marking: “Look Ma, No Stack!”...

LOOP

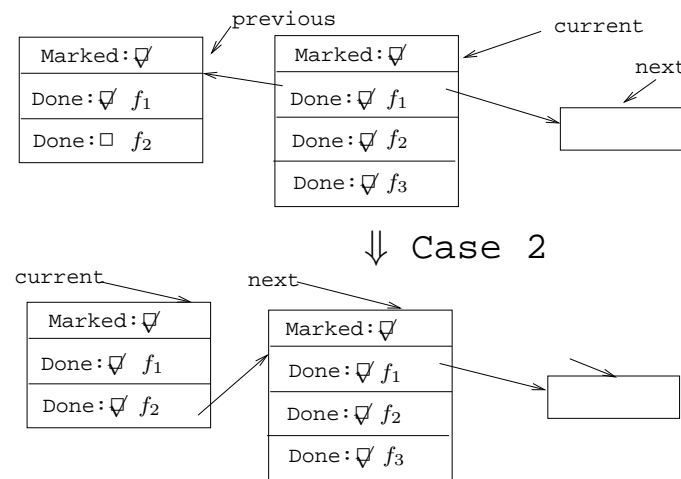
```
CASE 1: current's fields are not Done
  i := next field of current
  that's not Done;
  next := current↑.fi;
  IF next↑ isn't marked THEN
    current↑.fi := previous;
    previous := current;
    current := next;
  ENDIF;
CASE 2: current's fields are Done
  next := current;
  current := previous;
  i := next field of current
  that's not Done;
  previous := current↑.fi;
  current↑.fi := next;
```

ENDLOOP

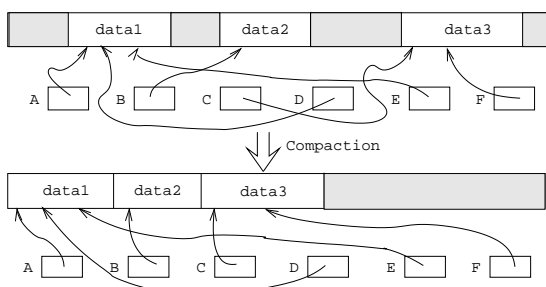
Marking: “Look Ma, No Stack!”...



Marking: “Look Ma, No Stack!”...



Sweeping: Compaction



1. Calculate the **forwarding address** of each cell.
2. Store the forwarding address of cell B in $B.forw_addr$.
3. If p points to cell B , replace p with $B.forw_addr$.
4. Move all cells to their forwarding addresses.

Readings and References

- Read Scott, pp. 395–401.