Copying Collection

Even if most of the heapspace is garbage, a mark and sweep algorithm will touch the entire heap. In such cases it would be better if the algorithm only touched the live objects.

Copying collection is such an algorithm. The basic idea is:

1. The heap is divided into two spaces, the from-space and the to-space.
2. We start out by allocating objects in the from-space.
3. When from-space is full, all live objects are copied from from-space to to-space.
4. We then continue allocating in to-space until it fills up, and a new GC starts.
An important side-effect of copying collection is that we get automatic compaction – after a collection to-space consists of the live objects in a contiguous piece of memory, followed by the free space.

This sounds really easy, but ···:

- We have to traverse the object graph (just like in mark and sweep), and so we need to decide the order in which this should be done, depth-first or breadth-first.
- DFS requires a stack (but we can, of course, use pointer reversal just as with mark and sweep), and BFS a queue. We will see later that encoding a queue is very simple, and hence most implementations of copying collection make use of BFS.
This sounds really easy, but …

An object in from-space will generally have several objects pointing to it. So, when an object is moved from from-space to to-space we have to make sure that we change the pointers to point to the new copy.
Mark-and-sweep touches the entire heap, even if most of it is garbage. Copying collection only touches live cells.

Copying collection divides the heap in two parts: \textit{from-space} and \textit{to-space}.

\textit{to-space} is automatically compacted.

How to traverse object graph: BFS or DFS?

How to update pointers to moved objects?

\textbf{Algorithm:}

1. Start allocating in \textit{from-space}.
2. When \textit{from-space} is full, copy live objects to \textit{to-space}.
3. Now allocate in \textit{to-space}.
Traversing the Object Graph:

- Most implementations use BFS.
- Use the to-space as the queue.

Updating (Forwarding) Pointers:

- When an object is moved its new address is stored first in the old copy.

Example:
Copying Collection Algorithm

1. scan := next := ADDR(to-space)
   - [scan ... next] hold the BFS queue.
   - Objects above scan point into to-space. Objects between scan and next point into from-space.

2. Copy objects pointed to by the root pointers to to-space.

3. Update the root pointers to point to to-space.

4. Put each object’s new address first in the original.

5. Repeat (recursively) with all the pointers in the new to-space.
   (a) Update scan to point past the last processed node.
   (b) Update next to point past the last copied node.
   Continue while scan < next.
Copying Collection Example...(A)
Copying Collection Example... (B)
Cost of Garbage Collection
The size of the heap is $H$, the amount of reachable memory is $R$, the amount of memory reclaimed is $H - R$.

\[
\text{amortized GC cost} = \frac{\text{time spent in GC}}{\text{amount of garbage collected}}
\]

\[
= \frac{\text{time spent in GC}}{H - R}
\]
The breadth first search phase touches all live nodes. Hence, it takes time $c_3 R$, for some constant $c_3$. $c_3 \approx 10$.

The heap is divided into a from-space and a to-space, so each collection reclaims $\frac{H}{2} - R$ words.

\[ \text{GC cost} = \frac{c_3 R}{\frac{H}{2} - R} \approx \frac{10R}{\frac{H}{2} - R} \]
Cost of GC — Copying Collection...

- If there are few live objects ($H \gg R$) the GC cost is low.
- If $H = 4R$, we get

$$GC \ cost = \frac{c_3 R}{4R - R} \approx 10.$$ 

This is expensive: 4 times as much memory as reachable data, 10 instruction GC cost per object allocated.
Read Scott, pp. 387–388.