CSc 553 — Principles of Compilation

11 : Garbage Collection — Generational Collection

Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

February 8, 2011

1

Generational Collection

- Works best for functional and logic languages (LISP, Prolog, ML, ...) because
  1. they rarely modify allocated cells
  2. newly created objects only point to older objects (`(CONS A B)` creates a new two-pointer cell with pointers to old objects),
  3. new cells are shorter lived than older cells, and old objects are unlikely to die anytime soon.

2

Generational Collection...

- Generational Collection therefore
  1. divides the heap into generations, G_0 is the youngest, G_n the oldest.
  2. allocates new objects in G_0.
  3. GC's only newer generations.
- We have to keep track of back pointers (from old generations to new).

3

Generational Collection... Functional Language:

```
(cons 'a '(b c))
      ⇕
t1:  x ← new '(b c);
t2:  y ← new 'a;
t3:  return new cons(x, y)
```

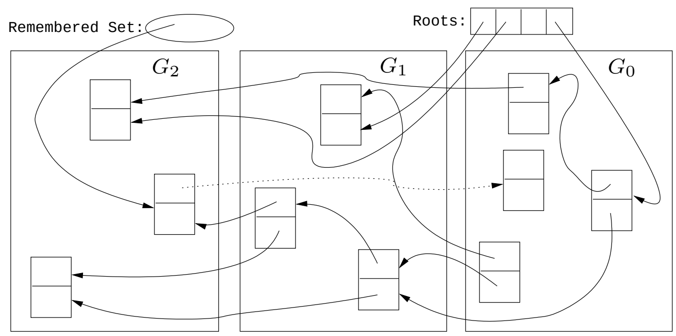
- A new object (created at time t_3) points to older objects.

Object Oriented Language:

t_1: `T` $\leftarrow$ `new Table(0);`
t_2: `x` $\leftarrow$ `new Integer(5);`
t_3: `T.insert(x);`

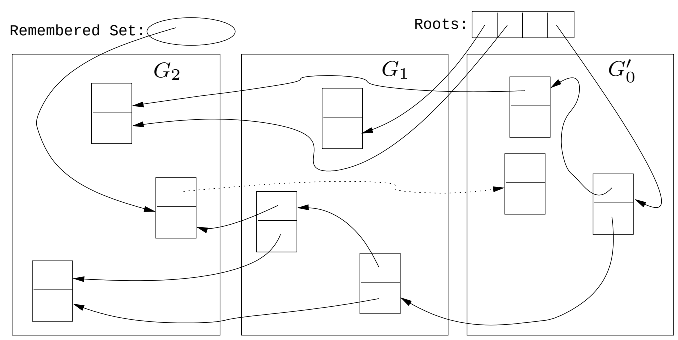
- A new object (created at time t_2) is *inserted into* an older object, which then points to the news object.

4



Generational Collection...

5



Generational Collection – After GC(G_0)

6

Generational Collection...

- Since old objects (in $G_n \cdots G_1$) are rarely changed (to point to new objects) they are unlikely to point into G_0.
- Apply the GC only to the youngest generation (G_0), since it is most likely to contain a lot of garbage.
- Use the stack and globals as roots.
- There might be some back pointers, pointing from an older generation into G_0. Maintain a special set of such pointers, and use them as roots.

- Occasionally GC older ($G_1 \cdots G_k$) generations.
- Use either mark-and-sweep or copying collection to GC G_0.

7

Remembering Back Pointers — Remembered List

After each pointer update `x.f := ···`, the compiler adds code to insert `x` in a list of updated memory locations:

```
x↑.f := ···
      ⇓
x↑.f := ···;
insert(UpdatedList, x);
```

8

Remembering Back Pointers — Remembered Set

As above, but set a bit in the updated object so that it is inserted only once in the list:

```
x↑.f := ···
      ⇓
x↑.f := ···;
IF NOT x↑.inserted THEN
   insert(UpdatedList, x);
   x.↑inserted := TRUE;
ENDIF
```

9

Remembering Back Pointers... — Card marking

- Divide the heap into "cards" of size 2^k.
- Keep an array `dirty` of bits, indexed by card number.
- After a pointer update `x↑.f := ···`, set the dirty bit for card `c` that `x` is on:

```
x↑.f := ···
      ⇓
x↑.f := ···;
dirty[x div 2^k] := TRUE;
```

10

Remembering Back Pointers... — Page marking I

- Similar to Card marking, but let the cards be virtual memory pages.
- When `x` is updated the VM system automatically sets the `dirty` bit of the page that `x` is on.
- We don't have to insert any extra code!

11

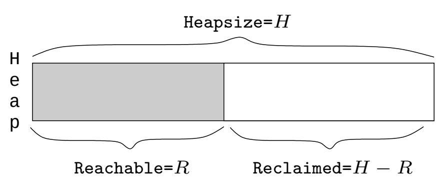
Remembering Back Pointers... ——— Page marking II ———

- The OS may not let us read the VM system's dirty bits.
- Instead, we write-protect the page `x` is on.
- On an update `x↑.f := ···` a protection fault is generated. We catch this fault and set a dirty bit manually.
- We don't have to insert any extra code!

12

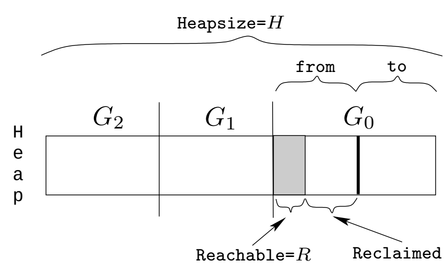
Cost of Garbage Collection

- The size of the heap is H, the amount of reachable memory is R, the amount of memory reclaimed is $H - R$.



$$\begin{aligned} \textit{amortized GC cost} &= \frac{\textit{time spent in GC}}{\textit{amount of garbage collected}} \\ &= \frac{\textit{time spent in GC}}{H - R} \end{aligned}$$

13

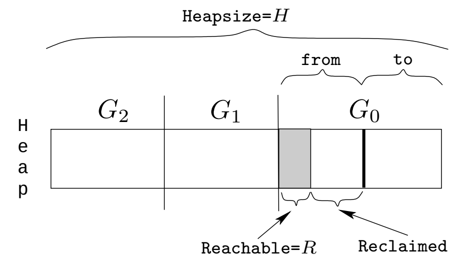


Cost of GC — Generational Collection

- Assume the youngest generation (G_0) has 10% live data, i.e. $H = 10R$.
- Assume we're using copying collection for G_0.

$$\textit{GC cost}_{G_0} = \frac{c_3 R}{\frac{H}{2} - R} = \frac{c_3 R}{\frac{10R}{2} - R} \approx \frac{10R}{4R} = 2.5$$

14



Cost of GC — Generational Collection…

$$GC\ cost_{G_0} = \frac{c_3 R}{\frac{H}{2} - R} = \frac{c_3 R}{\frac{10R}{2} - R} \approx \frac{10R}{4R} = 2.5$$

- If $R \approx 100$ kilobytes in G_0, then $H \approx 1$ megabyte.
- In other words, we've wasted about 900 kilobytes, to get 2.5 instruction/word GC cost (for G_0).

15

Readings and References

- Read Scott, pp. 388–389.