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
     \((\text{CONS } A \ B)\) 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.

Generational Collection

- divides the heap into generations, \(G_0\) is the youngest, \(G_n\) the oldest.
- allocates new objects in \(G_0\).
- GC’s only newer generations.
- We have to keep track of back pointers (from old
generations to new).

Functional Language:
\[
(\text{cons } 'a ' (b c))
\]
\[
t_1: x \leftarrow \text{new } '(b c);
t_2: y \leftarrow \text{new } 'a;
t_3: \text{return new cons}(x, y)
\]

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

Object Oriented Language:
\[
t_1: T \leftarrow \text{new Table}(0);
t_2: x \leftarrow \text{new Integer}(5);
t_3: T.\text{insert}(x);
\]

A new object (created at time \(t_2\)) is inserted into an older object, which
then points to the new object.
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 $G_0$. 

Remembering Back Pointers
Remembering Back Pointers

**Remembered List**

After each pointer update \( x.f := \cdots \), the compiler adds code to insert \( x \) in a list of updated memory locations:

\[
\begin{align*}
x.\uparrow.f &:= \cdots \\
\downarrow \\
x.\uparrow.f &:= \cdots; \\
\text{insert}(\text{UpdatedList}, \ x);
\end{align*}
\]

**Remembered Set**

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

\[
\begin{align*}
x.\uparrow.f &:= \cdots \\
\downarrow \\
x.\uparrow.f &:= \cdots; \\
\text{IF NOT } x.\uparrow.\text{inserted THEN} \\
\quad \text{insert}(\text{UpdatedList}, \ x); \\
\quad x.\uparrow.\text{inserted} := \text{TRUE}; \\
\text{ENDIF}
\end{align*}
\]

**Card marking**

- Divide the heap into “cards” of size \( 2^k \).
- Keep an array \textit{dirty} of bits, indexed by card number.
- After a pointer update \( x.\uparrow.f := \cdots \), set the dirty bit for card \( c \) that \( x \) is on:

\[
\begin{align*}
x.\uparrow.f &:= \cdots \\
\downarrow \\
x.\uparrow.f &:= \cdots; \\
\text{dirty}[x \nowhere 2^k] := \text{TRUE};
\end{align*}
\]

**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 \textit{dirty} bit of the page that \( x \) is on.
- We don’t have to insert any extra code!
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 $x \uparrow f := \cdots$ a protection fault is generated. We catch this fault and set a dirty bit manually.
- We don’t have to insert any extra code!

Readings and References

- Read Scott, pp. 395–401.