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.

Generational Collection therefore 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).

Functional Language:
\[(\text{cons } 'a ' (b c)) \]
\[
\begin{align*}
  \text{t}_1: & \quad x \leftarrow \text{new } '(b c); \\
  \text{t}_2: & \quad y \leftarrow \text{new } 'a; \\
  \text{t}_3: & \quad \text{return new cons}(x, y)
\end{align*}
\]

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

Object Oriented Language:
\[
\begin{align*}
  \text{t}_1: & \quad T \leftarrow \text{new Table}(0); \\
  \text{t}_2: & \quad x \leftarrow \text{new Integer}(5); \\
  \text{t}_3: & \quad T.\text{insert}(x);
\end{align*}
\]

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

Remembered List

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

\[
x.f := \cdots
\]

\[
\downarrow
\]

\[
x.f := \cdots;
\]

\[
\text{insert(UpdatedList, x)};
\]
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:

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

__________________________ 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 \(x \uparrow.f := \cdots\), set the dirty bit for card \(c\) that \(x\) is on:

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

__________________________ 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 dirty bit of the page that \(x\) is on.
- We don’t have to insert any extra code!

__________________________ 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!
Cost of Garbage Collection

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

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

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 $G_0$.

  \[
  \text{GC cost}_{G_0} = \frac{c_3 R}{H - R} = \frac{c_3 R}{10R - R} \approx \frac{10R}{4R} = 2.5
  \]

Readings and References

- Read Scott, pp. 388–389.