Automatic Storage Management

What is automatic storage management?

- "Large" objects are allocated in memory, either explicitly or implicitly.
- Unused memory is reclaimed by the language implementation whenever necessary (or useful).
- Language implementation determines when memory may be reclaimed.

Value vs. Object-Oriented

In ML tuples are as convenient as integers:

```ml
val tuple = (1, 2.0, "\$")
fun f(x:int,y:int):int*int = ...
```

No need to think about storage allocation.

A cost semantics for space gives a clean abstract accounting of memory usage.

Value vs. Object-Oriented

In Java it's much more laborious:

```java
class Pair {
  int x;
  float y;
  Pair (int x, float y) {
    this.x = x; this.y = y;
  }
  ... new Pair (1, 2.0) ...
```

But it's clear exactly where allocation occurs.
Some Advantages

- It’s automatic.
- Guaranteed space safety (no dangling pointers).
- Avoids space leaks caused by holding on to reclaimable memory.
- Much more efficient than malloc/free.
- Plays well with multiple threads of control.

Some Disadvantages

- It’s automatic.
- Does not eliminate all space leaks.
- Wastes storage for the collector (more on this later).
- Language must be designed for automatic storage management.

Some Myths

- Less efficient than manual memory management. Completely bogus!
- Inherently incompatible with real-time. There are good real-time collectors.
- Collector overhead reduces throughput. There are good parallel collectors.
- “I know better than any collector.” Repeatedly proved false.

Modelling Storage Management

The \(\kappa\)-machine treats all values equally.

- Substitution for variables.
- Storage in stack frames.
- Manipulation by primitives.

But this is completely unrealistic!

- Tuples, closures, etc. are “bigger” than integers or booleans!
- Does not account for the cost of manipulating large objects.

Today: a more realistic model of storage.

- Large objects are allocated in the heap.
- Pointers are small objects that represent allocated objects.

The \(\lambda\)-Machine

Classify values as small and large.

- Small: integers, booleans, locations.
- Large: functions (tuples, injections, objects, \ldots).

Locations are abstract pointers.

- No “pointer arithmetic”.
- Indices into the heap.
The A-Machine

States: \((H, k, e)\), where

- \(H\) is a heap mapping locations \(l\) to large values \(H(l)\).
- \(k\) is a control stack, as in the \(K\)-machine.
- \(e\) is an expression, for example in \(\Pi\text{-}\lambda\).

Initial: \((\emptyset, e, \epsilon)\), where \(e\) is a closed expression.

Final: \((H, \epsilon, v)\), where \(v\) is a small value.

The A-Machine functions are not values any more!

- Work must be done to allocate them.
- Cannot stop evaluation at a function expression.

Stack frames contain only small values.

- e.g., \(\nu_1(-)\), where \(\nu_1\) is a small value.
- Stack slots can accommodate only small values.

The heap is "passive" in these rules.

A heap \(H\) is closed, or self-contained, iff \(\text{Locs}(H) \subseteq \text{dom}(H)\).

- Every free location in any value in the heap . . .
- must occur within the domain of the heap.

That is, \(H\) contains no dangling pointers.

Preservation of Heap Closure

An A-machine state \((H, k, e)\) is closed iff

- \(H\) is closed (self-contained).
- \(\text{Locs}(k) \cup \text{Locs}(e) \subseteq \text{dom}(H)\).

Lemma 1

If \((H, k, e)\) is closed and \((H, k, e) \rightarrow (H', k', e')\), then \((H', k', e')\) is also closed.

The proof is by induction on evaluation.
Garbage Collection

The job of a garbage collector is to reclaim unnecessary storage.

- How do you reclaim storage?
- When is storage unnecessary?

In an abstract model there is no need to collect garbage (heaps are finite, but not fixed, capacity).

But the _A-machine provides a good framework for describing how collectors work._

---

Necessity

The heap $H_l$ is $H$ with location $l$ removed.

- The state $(H_l, k, e)$ might or might not be closed!
- That is, $l$ might or might not be dangling.

The location $l$ is unnecessary in $(H, e, e)$, where $e$ is

\[
\begin{array}{ll}
& \text{if true then } d \\
& \text{else } l(0)
\end{array}
\]

Execution completes without ever using $l$.

---

Reachability

Proposition 3

A location $l$ is unnecessary in state $(H, k, e)$ if (not only if) $l$ is unreachable in that state.

That is, reachable locations are not unnecessary. This does not mean that reachable locations are necessary!

The reachable locations are also called live; the unreachable ones are called dead.

---

Necessity

A location $l \in \text{dom}(H)$ is unnecessary for a state $(H, k, e)$ iff execution is unaffected by location $l$.

Formally,

\[
(H, k, e) \Rightarrow^* (H', e, v) \quad \text{iff} \quad (H_l, k, e) \Rightarrow^* (H', e, v).
\]

In particular, one diverges iff the other does.

---

Reachability

A location $l$ is reachable from a set $L$ of locations iff either

- $l \in L$, or
- $l$ is reachable from $L \cup \text{Locs}(H(l'))$ for some $l' \in L$.

A location $l$ is reachable in state $(H, k, e)$ iff $l$ is reachable from $\text{Locs}(k) \cup \text{Locs}(e)$.

Finding the reachable locations for a state is called tracing.
Reachability

Nearly all practical garbage collectors are based on tracing.

- Sufficient condition for being unnecessary.
- Practical, easy to implement.

So all practical garbage collectors may fail to collect some garbage.

- So space leaks are (theoretically) still possible!
- In practice this is rarely, if ever, a problem.

Collecting Garbage

Reachability-based collectors proceed by

- Computing the set of reachable locations;
- Reclaiming those that are deemed unreachable.

One efficient and popular technique is copying collection.

Copying Collection

The heap is divided into two semi-spaces:

- From space: where allocation takes place.
- To space: where reachable data is copied to.

At any moment only the “from” semi-space is active.

- Half of available storage is "wasted".
- Space loss can be mitigated by various tricks.

After copying, the collector performs a flip.

- “To” space becomes the new “from” space. It now contains only reachable data.
- “From” space becomes the new “to” space. It contains only garbage.

This can be done in constant time by changing a single pointer (the allocation pointer).
Adding GC to the A-Machine

Add a new instruction to invoke the garbage collector:

\[
(H, \text{Locs}(k) \cup \text{Locs}(e), \emptyset) \rightarrow (H' \cup \emptyset, H')
\]

\[
(H, k, e) \rightarrow (H, k, e)
\]

This instruction may be executed at any time.

- Models unpredictability of collection.
- Abstracts from any specific collection policy.

The G-Machine

Garbage collection is performed by the G-machine.

States: \((H_f, S, H_t)\), where

- \(H_f\) and \(H_t\) are the from- and to-spaces.
- \(S\) is a set of locations, called the scan set.

Initial state: \((H, \text{Locs}(k) \cup \text{Locs}(e), \emptyset)\).

Final state: \((H_f, \emptyset, H_t)\).

Collector Invariants

1. The scan set contains only “valid” locations: \(S \subseteq \text{dom}(H_f) \cup \text{dom}(H_t)\).

2. The “from” and “to” spaces are disjoint: \(\text{dom}(H_f) \cap \text{dom}(H_t) = \emptyset\).

3. Every location in the “to” space is either copied or pending: \(\text{Locs}(H_t) \subseteq \text{dom}(H_t) \cup S\).

4. Every location in “from” space has either been copied or is still in “from” space: \(\text{Locs}(H_f) \subseteq \text{dom}(H_f) \cup \text{dom}(H_t)\).

Lemma 5

If \((H_f, S, H_t) \rightarrow (H'_f, S', H'_t)\), then \(H_f \cup H_t = H'_f \cup H'_t\) and \(S \cup \text{dom}(H_t) \subseteq S' \cup \text{dom}(H'_t)\).

Corollary 6

Let \(S = \text{Locs}(k) \cup \text{Locs}(e)\) and let \(H\) be a closed heap such that \(S \subseteq \text{dom}(H)\). If \((H, S, \emptyset) \rightarrow (H' \cup \emptyset, H')\), then

1. The reachable portion of the heap is preserved: \(H' \subseteq H\).

2. The heap \(H'\) covers the control stack and current expression: \(S \subseteq \text{dom}(H')\).
Summary

Garbage collection is increasingly important in modern PLs.

Most collectors are based on copying collection.

Copying collection correctly collects unreachable locations and preserves all reachable locations. Consequently, it never collects an unnecessary location.