Automatic Storage Management

Most modern languages support automatic storage management, or garbage collection.

- Pioneered by Lisp in the early 1960's.
- Popularized by Java in the late 1990's.
- Increasingly seen as vital for building large systems.

Value- vs. Object-Oriented

Value-oriented languages (such as ML):

- Implicit allocation of “large” objects.
- Allocation is completely hidden from the programmer.

Object-oriented languages (such as Java):

- Explicit allocation of objects (“new”).
- Allocation is fully exposed to the programmer.

Value- vs. Object-Oriented

In ML tuples are as convenient as integers:

```ml
val tuple = (1, 2.0, "3")
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.
• Doesn’t 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 C-machine treats all values equally.

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

Modelling Storage Management

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 A-Machine

Classify values as small and large.

• Small: integers, booleans, locations.
• Large: functions (tuples, injections, objects, …).

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 C-machine.
- \(e\) is an **expression**, essentially as in MinML.

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

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

The A-Machine

Most rules are like the C-machine:

\[
(H, k, \text{if } e \text{ then } c_1 \text{ else } c_2 \text{ fi}) \rightarrow_{A} (H, k, c_1) \\
(H, k, \text{if } \Box \text{ then } c_1 \text{ else } c_2 \text{ fi}; k, e) \rightarrow_{A} (H, k, c_1) \\
(H, k, \text{if } \Box \text{ then } c_1 \text{ else } c_2 \text{ fi}; k, \text{true}) \rightarrow_{A} (H, k, c_1) \\
(H, k, \text{if } \Box \text{ then } c_1 \text{ else } c_2 \text{ fi}; k, \text{false}) \rightarrow_{A} (H, k, c_2)
\]

The heap is “passive” in these rules.

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., \(\text{apply}(v_1, \Box)\), where \(v_1\) is a small value.
- Stack slots can accommodate only small values.

A Variation

Alternatively, “unwind” the recursion when the function is allocated:

\[
(H, k, \text{fun } x(y; \tau_1): \tau_2 \text{ is end}) \rightarrow_{A} (H, k, \text{fun } x(y; \tau_1): \tau_2 \text{ is end}, k, l)
\]

Calling a function is correspondingly simplified:

\[
\text{apply}(v_1, \Box); k, v_2) \rightarrow_{A} (H, k, \text{fun } x(y; \tau_1): \tau_2 \text{ is end})
\]

This introduces cycles in the heap, whereas the first method does not.

The A-Machine

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

- 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).
- \(\text{FL}(k) \cup \text{FL}(e) \subseteq \text{dom}(H)\).

Lemma 1
If \((H, k, e)\) is closed and \((H, k, e) \rightarrow_A (H', k', e')\), then \((H', k', e')\) is also closed.

The proof is by induction on evaluation.

Safety of the A-Machine

When is an A-machine state well-formed?

First cut:
- \(k : \tau\) stack;
- \(e : \tau\);
- \(H : ?\).

What is the right condition on \(H\)?
What if \(k\) and \(e\) involve locations \(l\)?

Safety for the A-Machine

A location typing is a finite function \(\Lambda\) mapping locations \(l\) to types \(\Lambda(l)\).

Define \(\Lambda; \Gamma \vdash e : \tau\) by adding one rule to MinML:

\[
\frac{\Lambda(l) = \tau}{\Lambda; \Gamma \vdash l : \tau}
\]

Locations are treated similarly to variables for type checking purposes.

Safety for the A-Machine

If the heap can be cyclic, we define \(H : \Lambda\) iff

- \(\text{dom}(H) = \text{dom}(\Lambda)\);
- for every \(l \in \text{dom}(H)\), \(\Lambda; \emptyset \vdash H(l) : \Lambda(l)\).

This allows for cycles, since we assume what we need to prove!

(Compare the typing rule for recursive functions.)
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

\[
\text{if true then 3 else } l(0)
\]

Execution completes without ever using \(l\).

Necessity

Theorem 3

It is undecidable whether a given location is unnecessary in a given machine state.

Proof (sketch): Arrange that a location \(l\) is used iff some function \(f\) returns true when applied to 0 (e.g., if \(f(0)\) then \(l(9)\) else 17).

Consequently, every practical garbage collector is conservative!

- Cannot always find the “true” garbage.
- Must rely on an approximation.

Reachability

Proposition 4

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.

Reachability

A location \(l\) is reachable from a set \(L\) of locations iff either

- \(l \in L\), or
- \(l\) is reachable from \(L \cup FL(H(l'))\) for some \(l' \in L\).

A location \(l\) is reachable in state \((H, k, e)\) iff \(l\) is reachable from \(FL(k) \cup FL(e)\).

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.

Copying Collection

Collection proceeds by simultaneously

- Maintaining a set of reachable locations, and
- Copying the reachable locations to “to” space.

Initially, the reachable locations are those on the stack and in the current expression.

Finally, the reachable locations are empty (when all live locations are copied).

Copying Collection

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).

Copying Collection

The crucial advantage of copying collection: GC takes time proportional to the size of the live data.

Mark-sweep collectors do not share this property.

- Scan entire heap looking for live data.
- Somehow reclaim the dead data (via a free list).
- Avoid copying.
Adding GC to the A-Machine

Add a new instruction to invoke the garbage collector:

\[
(H, \text{FL}(k) \cup \text{FL}(e), \emptyset) \xrightarrow{\ell} (H'', \emptyset, H')
\]

\[
(H, k, e) \xrightarrow{\ell} (H', k, e)
\]

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: \((H_f, \text{FL}(k) \cup \text{FL}(e), \emptyset)\).

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:
   \(\text{FL}(H_t) \subseteq \text{dom}(H_f) \cup S\).

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

Collector Correctness

Lemma 6

If \((H_f, S, H_t) \xrightarrow{\ell} (H'_f, S', H'_t)\), 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 7

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