Lecture Notes on Storage Management

15-312: Foundations of Programming Languages
Daniel Spoonhower
Modified by Frank Pfenning

Lecture 21
November 11, 2004

In our discussion of mutable storage, a question was raised: if we allocate a new storage cell for each `ref` expression we encounter, when do we release these storage cells? As we will discover today, a similar question will be raised when we reconsider our implementation of pairs, lists, and closures, or generally any aggregate data structure.

In designing the E machine, our goal was to describe a machine that more accurately modeled the way that programs are executed on real hardware (for example, by using environments rather than substitution). However, most real machines will treat *small* values (such as integers) differently from *large* values (such as pairs and closures). Small values may be stored in registers or on the stack, while larger values, such as pairs and closures, must be allocated from the *heap*. While the storage associated with registers and the stack can be reclaimed at the end of a function invocation or lexical scope, there is no "obvious" program point at which we can reuse the storage allocated from the heap.

Clearly, for programs that run for hours, days, or weeks, we must periodically reclaim any unused storage. One possible solution is to require the programmer to explicitly manage storage, as one might in languages such as C or C++. However, doing so not only exposes the programmer to a host of new programming errors, but also makes it exceedingly difficult to prove properties of languages such as preservation.

An alternative approach is to require that the implementation of the language manage storage *for* the programmer. Automatic memory management or *garbage collection* can be found in most modern languages, including Java, C#, Haskell, and SML.

In this lecture, we will modify and extend the semantics of the E machine to account for the differences between small and large values and include new transition rules for automatically reclaiming unused storage.

The A Machine

In order to extend the semantics of the E machine with transition rules for automatic storage management, we must enrich our model of expressions, values, and program states. For the purposes of our discussion today, we will use a version of MinML that includes integers, functions, and lists. As we alluded to above, in order to provide a framework for automatic storage management, the A machine will distinguish *small* values from *large* values, as follows.

$$\begin{array}{llll} \textit{Small Values} & v & ::= & \mathtt{num}(n) \mid \mathtt{nil} \\ \textit{Large Values} & w & ::= & \langle\!\langle \eta; e \rangle\!\rangle \mid \mathtt{cons}(v_1, v_2) \end{array}$$

Closures and cons cells (*i.e.* large values) will not be stored directly in the stack or environment; instead we will use *locations* to refer to them indirectly. As in our formulation of references, locations (denoted syntactically as l) will not appear in the concrete syntax.

$$\begin{array}{llll} \textit{Locations} & l & & \\ \textit{Expressions} & e & ::= & \ldots \mid \mathtt{loc}(l) \\ \textit{Small Values} & v & ::= & \ldots \mid \mathtt{loc}(l) \end{array}$$

We will also maintain a finite mapping from locations to large values, called a *heap*. We allow locations to appear in the stack and environment, but whenever we are forced to compute with a pair or closure, we must look-up the actual value in the heap.[1]

$$\begin{array}{llll} \textit{Heaps} & H & ::= & \cdot \mid H, l{=}w \\ \textit{Environments} & \eta & ::= & \cdot \mid \eta, x{=}v \\ \textit{States} & s & ::= & H \ ; \ k > e \\ & & \mid & H \ ; \ k < v \end{array}$$

Frames f and stacks k are given as before but with the replacement of small values for values.

[1]The heap is similar in notion to the *store* as it appeared in our discussion of mutable references; however, while the store may be updated by assignment, the heap is immutable from the programmer's perspective.

Since the A machine does not allow small values to be maintained in or returned to the stack, in states where we previously returned large values, we must instead create and look-up locations. For example, cons cells are now introduced and eliminated according to the following rules.

$$H \; ; \; k > \texttt{cons}(e_1, e_2) \mapsto_{\mathsf{a}} H \; ; \; k \triangleright \texttt{cons}(\Box, e_2) > e_1$$

$$H \; ; \; k \triangleright \texttt{cons}(\Box, e_2) < v_1 \mapsto_{\mathsf{a}} H \; ; \; k \triangleright \texttt{cons}(v_1, \Box) > e_2$$

$$H \; ; \; k \triangleright \texttt{cons}(v_1, \Box) < v_2 \mapsto_{\mathsf{a}} H, l{=}\texttt{cons}(v_1, v_2) \; ; \; k < \texttt{loc}(l)$$

$$H \; ; \; k > \texttt{case}(e_1, e_2, x.y.e_3) \mapsto_{\mathsf{a}} H \; ; \; k \triangleright \texttt{case}(\Box, e_2, x.y.e_3) > e_1$$

$$H \; ; \; k \triangleright \texttt{case}(\Box, e_2, x.y.e_3) < \texttt{nil} \mapsto_{\mathsf{a}} H \; ; \; k > e_2$$

$$H \; ; \; k \triangleright \texttt{case}(\Box, e_2, x.y.e_3) < \texttt{loc}(l) \mapsto_{\mathsf{a}} H \; ; \; k \blacktriangleright (x{=}v_1, y{=}v_2) > e_3$$
$$\text{where } l{=}\texttt{cons}(v_1, v_2) \text{ in } H$$

Recall that environment frames $k \blacktriangleright \eta$ on the stack are popped when values are returned past them, and that variables are looked up in the environments on the stack from right to left (see also Assignment 4 and the code in the sample solution). We will now return to the question, when can values safely be removed from the heap?[2]

Garbage and Collection

We would like to state that "the collector does not change the behavior of the program." That is, garbage should be exactly those parts of the program state that do not affect the result of evaluation. Consider the following program,

```
(let p = cons(3,cons(4,nil)) in
  case p of nil => 2
     | cons(n,k) = p in
    [a] fn x => n
  end
end [b]) 7 [c];
```

If we allocate p as described above, when it is safe to free it? At point $[a]$? $[b]$? $[c]$? We would like to release the storage associated with a location

[2] Though if we recall our original question with respect to references, we should note that the ideas described here can also be extended to encompass mutable storage.

as soon as it becomes unnecessary to the correct execution of the program. As it turns out, we will not be able to determine exactly when a particular location is no longer necessary: doing so is undecidable!

Instead we will make a conservative[3] assumption about whether or not a location is necessary: we will assume that any location that is *reachable* may be necessary. To do so, we will need to enumerate the *free locations* of a heap, stack, environment or value. (For the moment will we use the syntax $FL()$ to informally refer to these free locations; we will be more precise later.)

Given this notion of garbage, collection is exactly the process of removing garbage from the heap. During our discussion of mutable storage, something akin to the following transition rule was suggested.

$$\frac{FL(H, k, \eta) = \emptyset}{H \cup H' \; ; \; k > e \mapsto_{\mathsf{a}} H \; ; \; k > e} \; \textbf{?}$$

Recall that this rule was deficient in its inability to reclaim (unreachable) cycles in H. For the time being, however, we will tackle a larger problem: how can we separate H from H'?

Tracing Collection

At the most abstract level, the garbage collector has to traverse the stack k and follow chains of location pointers in the heap in order to see which locations may still be relevant to the evaluation of e in k. Note that an expression e may contain free variables (which will be bound to small values in environment in k), but never free locations. This means we don't have to traverse e to see which heap cells may be "live" for the current computation. This general technique is called *tracing*. We now describe a tracing collector using our notation of judgments. In what follows we describe more concrete realizations of this general idea that are closer to what actual implementations do.

The state of the garbage collector has the form $H_f \; ; \; k \; ; \; H_t$ where H_f is the so-called *from-space* that we are traversing and H_t is the so-called *to-space* where we move reachable locations found in H_f. Since locations remain abstract, we simply move them from H_f to H_t. The judgment above is invoked in the following way:

[3]"Conservative" is also, somewhat erroneously, used to describe garbage collection in the presence of incomplete knowledge of the structure of the stack or heap (*e.g.* as in an implementation of C).

$$\frac{H \;;\; k \;;\; \cdot \mapsto_{\mathsf{g}}^{*} H_f \;;\; \bullet \;;\; H'}{H \;;\; k > e \mapsto_{\mathsf{a}} H' \;;\; k > e}$$

That is, we start the garbage collector with the current heap H as the from-space and an empty to-space. Then we trace k and H, moving locations to the to-space until the stack is empty and we can return to the normal evaluation.

Note that this rule can apply whenever we are in the process of evaluating an expression. In a more realistic scenario the garbage collector either starts when we run out of space or acts concurrently on the heap.

Next we describe the rules for garbage collection, using single-step transitions. We use the stack k as a "stack", pushing onto it those portions of the small values that we may still have to trace. Since a stack cannot have values on it directly, only environments, we will use environment with anonymous variables. Recall the invariants on expressions (only free variables, no locations), environments (binds variable to small values) and heaps (binds locations to large values).

$$
\begin{array}{l}
H_f \;;\; k \triangleright \mathtt{cons}(\Box, e_2) \;;\; H_t \quad \mapsto_{\mathsf{g}} \quad H_f \;;\; k \;;\; H_t \\
H_f \;;\; k \triangleright \mathtt{cons}(v_1, \Box) \;;\; H_t \quad \mapsto_{\mathsf{g}} \quad H_f \;;\; k \blacktriangleright (\_{=}v_1) \;;\; H_t \\
H_f \;;\; k \triangleright \mathtt{case}(\Box, e_2, x.y.e_3) \;;\; H_t \quad \mapsto_{\mathsf{g}} \quad H_f \;;\; k \;;\; H_t \\[1ex]
H_f \;;\; k \blacktriangleright \cdot \;;\; H_t \quad \mapsto_{\mathsf{g}} \quad H_f \;;\; k \;;\; H_t \\
H_f \;;\; k \blacktriangleright (\eta, x{=}\mathtt{nil}) \;;\; H_t \quad \mapsto_{\mathsf{g}} \quad H_f \;;\; k \blacktriangleright \eta \;;\; H_t \\
(H_f, l{=}\mathtt{cons}(v_1, v_2)) \;;\; k \blacktriangleright (\eta, x{=}l) \;;\; H_t \quad \mapsto_{\mathsf{g}} \\
\qquad\qquad H_f \;;\; k \blacktriangleright (\eta, \_{=}v_1, \_{=}v_2) \;;\; H_t, l{=}\mathtt{cons}(v_1, v_2) \\
H_f \;;\; k \blacktriangleright (\eta, x{=}l) \;;\; (H_t, l{=}w) \quad \mapsto_{\mathsf{g}} \quad H_f \;;\; k \blacktriangleright \eta \;;\; (H_t, l{=}w)
\end{array}
$$

Similar rules apply to closures; some of them are given in Assignment 8 on *Garbage Collection* where more details can be found. Note that the last two rules distinguish the two cases where a heap cell has still to be moved, or has already been moved. In the first case, we push v_1 and v_2 onto the stack, since we have to trace any pointers in them as well. Note that circular data structures, although they cannot be constructed in the given language fragment, present no problem to the garbage collector.

Given our definition of a garbage collector, we could now prove not only that the algorithm terminates, but that it is safe, and it preserves the meaning of programs according to our previous definitions of MinML. The first proof is relatively straightforward; the latter two follow in a manner similar to our proofs for the E machine (with the addition of typing rules for the heap H).

Copying Collection

We now give a slightly lower level view of garbage collection where both from-space and to-space are actually regions in memory whose cells are addressed by integers. In this case, we actually divide the whole available memory into two disjoint regions: one that the evaluator uses, and one that is reserved for the time that we need to call the collector.

Heap cells are allocated from lower to higher addresses, using a special `next` pointer to keep track of the next available address. The garbage collector is invoked when we are attempting to use more than half of the available space.

We then trace the stack and the cells in from-space, moving the cell contents to to-space as we encounter them. Of course, references to memory in the stack need to be updated to point to the new locations of the cells.

Moreoever, we need to account for multiple pointers to the same locations. In order to preserve sharing, we replace the cell content by a *forwarding pointer* that goes from from-space to to-space. When we encounter a forwarding pointer when tracing the heap, we just update the pointer in the stack to the destination of the forwarding pointers.

Once the whole stack has been traced, all reachable cells have been moved to the beginning of the to-space. As this point we flip the roles of the two semi-spaces and resume evaluation.

A pictorial example of copying collection can be found in Figure 1. The contents of blank cells is irrelevant for the purposes of the garbage collection algorithm. They will never be visited because tracing never reaches them.

There are many refinements of copying collection. For example, in order to avoid using additional stack space for tracing, we use a second pointer in to-space so that we always know we still have to trace the region between this second pointer and the `next` pointer. In essence, we use the heap as a kind of special purpose stack.

Other refinements include *incremental collection*, where we do not completely stop the running program but interleave actions of the garbage collector with actions of the running program, and *generational collection* where we collect smaller parts instead the whole semi-space all at once.

Mark-and-Sweep Collection

Another important algorithm for garbage collection is mark-and-sweep, even though it seems to have fallen into disfavor more recently.

A mark-and-sweep collector does not divide the heap into two semi-space, but reserves an additional bit for each heap cell called a mark. Initially all heap cells are unmarked, and the heap is arranged into a linked list of cells called the *free list*. When we allocate an element from the heap we take the first element from the free list and update the free list pointer to its next element.

When the free list become empty, we have to invoke the garbage collector. It traces the heap, starting from the stack, much in the same way as the copying collector. However, rather than copying heap elements it marks them as being reachable.

In a second phase the garbage collector sweeps through the whole memory (not just the reachable cells). During this sweep it adds any unmarked cells to the free list and removes the mark from any marked cells.

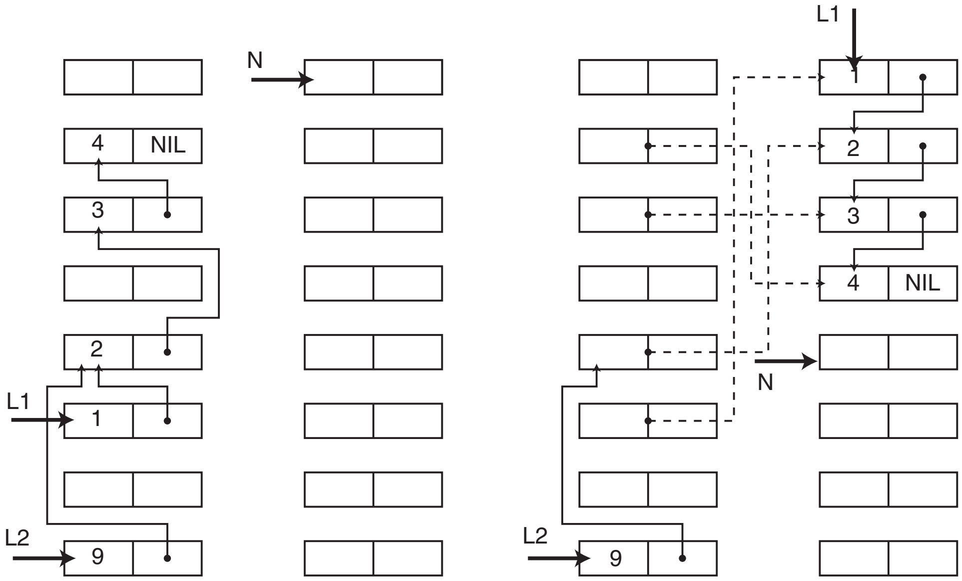
A graphic example of mark-and-sweep collection can be found in Figure 2.

In Assignment 8 you have the opportunity to compare copying and mark-and-sweep collection and assess their relative merits, so we will not give a detailed analysis here. One advantage of copying collection that your analysis will probably not be able to reveal is *locality*. When copying, we actually move the elements of the data structures closer together, at the beginning of the to-space. This means better cache behavior which can have dramatic impact on running times on modern machine architectures. As a result, even more mark-and-sweep garbage collectors some algorithm for compacting memory have been developed to avoid the natural *fragmentation* of the heap.

Reference Counting

In a reference counting garbage collector every cell has a counter associated with it that tracks the number of references to it. When we allocate a cell, this counter is initialized to 1. Operations of the (abstract) machine need to maintain these counters. As soon as one of them becomes 0, the cell is deallocated and the reference counts of the cells that it might point to are decremented, leading perhaps to further garbage collection.

Reference counting is suspect for the heaps of functional languages because of the overhead of maintaining the reference counts, and because it does not work properly with circular structures which prevent reference counts from going to 0! However, the are many less general situations where reference counts are appropriate, such as file descriptors in an operating system, or channels for communication in a distributed environment. In those situations, the overhead of maintaining reference counts is small, while a tracing collector would be hard or impossible to implement because we may not know or even have access to the internals of all processes that my access a resource.



Initial heap with roots L1 and L2. Next pointer N exceeds bounds of current semi-space.

Heap after copying L1. Dashed lines represent forwarding pointers.

Heap after copying L2. Next pointer N is initialized. Semi-spaces have switched roles.

Figure 1: Example of Copying Garbage Collection

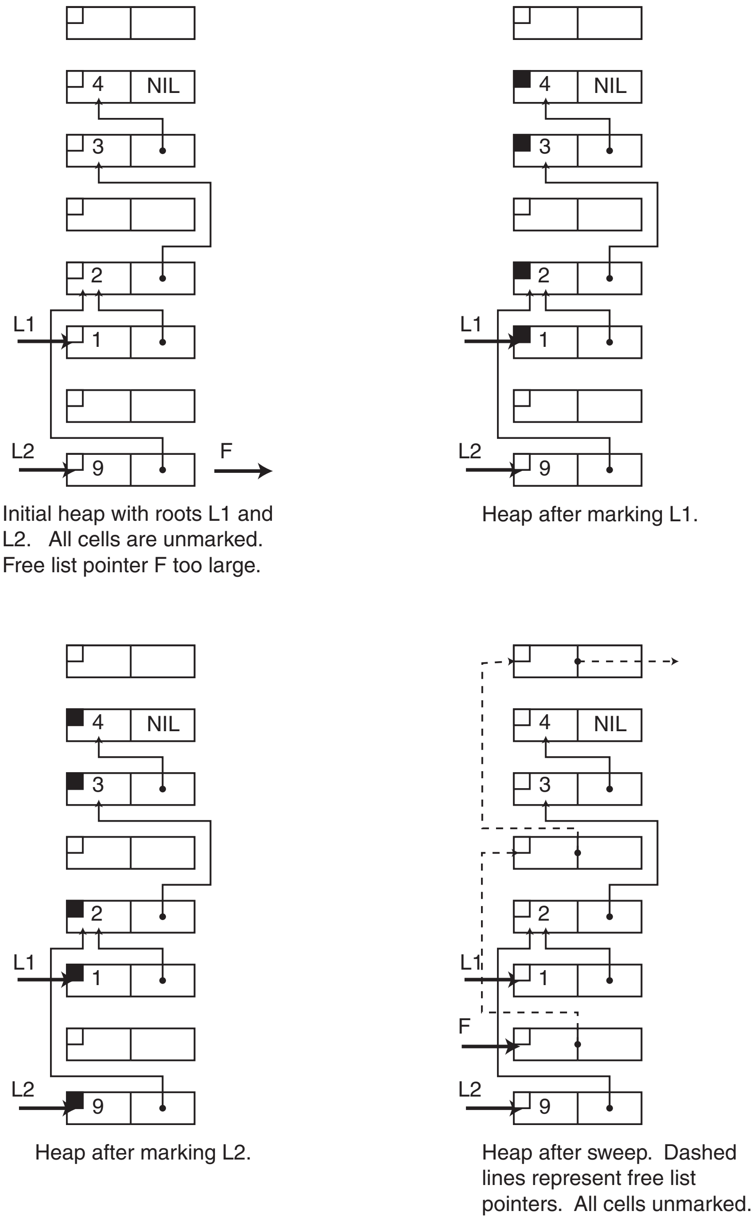


Figure 2: Examples of Mark and Sweep Garbage Collection