Lecture Notes on Storage Management

15-312: Foundations of Programming Languages
Daniel Spoonhower

Lecture 18
October 28, 2003

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 and closures.

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.

1 Definitions

1.1 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, booleans, functions, and pairs. As we eluded to above, in order to provide a framework for automatic storage management, the A machine will distinguish *small* values from *large* values, as follows.

$$\begin{array}{llll} \textit{(small values)} & v & ::= & \mathtt{num}(n) \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{unitel} \\ \textit{(large values)} & V & ::= & \langle\!\langle \eta; e \rangle\!\rangle \mid \mathtt{pair}(v_1, v_2) \end{array}$$

Closures and pairs (*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.

$$\begin{array}{llll} \textit{(locations)} & l & & \\ \textit{(expressions)} & e & ::= & \ldots \mid l \\ \textit{(small values)} & v & ::= & \ldots \mid l \end{array}$$

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]

$$\begin{array}{llll} \textit{(heaps)} & H & ::= & \cdot \mid H[l = V] \\ \textit{(environments)} & \eta & ::= & \cdot \mid \eta, x = v \\ \textit{(states)} & s & ::= & H \mid k \mid \eta > e \\ & & \mid & H \mid k \mid \eta < v \end{array}$$

[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.

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

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, pairs are now introduced and eliminated according to the following rules.

$$H \mid k \triangleright \texttt{pair}(v_1, \square) \mid \eta < v_2 \mapsto_{\mathsf{a}} H[l = \texttt{pair}(v_1, v_2)] \mid k \mid \eta < l$$

$$H[l = \texttt{pair}(v_1, v_2)] \mid k \triangleright \texttt{fst}(\square) \mid \eta < l \mapsto_{\mathsf{a}} H[l = \texttt{pair}(v_1, v_2)] \mid k \mid \eta < v_1$$

We will now return to the question, when can values safely be removed from the heap?[2]

1.2 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 = (3, 4) in
  let n = fst p in
    [a] fn x:int => 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 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

[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.

[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).

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.

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

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'?

2 Semi-space Collection

Consider the following heap and program state.

l	V
l_1	$\texttt{pair}(6, l_2)$
l_2	$\texttt{pair}(7, 8)$
l_3	$\texttt{pair}(9, l_6)$
l_4	$\texttt{pair}(l_3, l_6)$
l_5	$\texttt{pair}(l_1, 11)$
l_6	$\texttt{pair}(9, 10)$

$$\ldots \mid \cdot \mid x = l_4 > \texttt{snd}(\texttt{snd}(\texttt{fst}(x)))$$

What parts of the heap are garbage? How might we (algorithmically) determine this?

Below, we describe a collection algorithm that determines which values are reachable by moving those values that are reachable into the *to-space* H_t and then discarding any values that remain behind in the *from-space* H_f.

1. Initialize $H_f = H$, $S = \mathit{FLS}(k) \cup \mathit{FLE}(\eta)$, and $H_t = \emptyset$.
2. While S is not empty, repeat:
   - 2.1 Remove some location l from S.
   - 2.2 If $l \in \text{dom}(H_t)$ then continue,

2.3 Otherwise, remove the mapping $l = V$ from H_f,
Let $H_t = H_t[l = V]$,
and $S = S \cup \mathit{FLLV}(V)$

3. Let $H = H_t$.

Where by *FLLV*, we mean the free locations of a large value:

$$\begin{array}{rcl} \mathit{FLLV}(\texttt{pair}(v_1, v_2)) & = & \mathit{FLSV}(v_1) \cup \mathit{FLSV}(v_2) \\ \mathit{FLLV}(\langle\!\langle \eta; e \rangle\!\rangle) & = & \mathit{FLE}(\eta) \end{array}$$

Where $\mathit{FLSV}(v)$ and $\mathit{FLE}(\eta)$ are defined in the obvious way. Because this method of copying divides the heap into two halves, it is known as *semi-space* copying collection.

In order to get a better feel for the decisions that go into the implementation of a garbage collector, we might consider a machine model that captures the sizes of large values, their representation in the heap, and the adjacency of one heap location to another. Doing so will help us to understand the cost of statements such as "Let $H_t = H_t[l = V]$." Specifically, we will be required to understand how this distinction between H_f and H_t is made. A model of this level of sophistication, however, is beyond the scope of these notes.

2.1 The G Machine

So that we are better equipped to reason about our copying algorithm, we will introduce yet another model of computation, the G machine. This machine will model the step-by-step execution of the garbage collector. Since it will model a semi-space copying collector, we will the following syntax to describe its state.

$$(H_f, S, H_t)$$

On the left we have the *from-space*, in the center the *scan set* and on the right, the *to-space*. We can then rewrite the algorithm above using the following inference rules describing the transitions of the G machine.

$$\frac{}{(H_f[l = V], S \cup \{l\}, H_t) \mapsto_{\mathsf{g}} (H_f, S \cup \mathit{FLLV}(V), H_t[l = V])}\ \mathit{Copy}$$

$$\frac{}{(H_f, S \cup \{l\}, H_t[l = V]) \mapsto_{\mathsf{g}} (H_f, S, H_t[l = V])}\ \mathit{Discard}$$

Notice that these rules correspond to two cases in the body of the loop above. Finally, we can show how to invoke the G machine, and give a more precise definition to *Collect*, as it appears below.

$$\frac{(H, \mathit{FLS}(k) \cup \mathit{FLE}(\eta), \emptyset) \mapsto_{\mathsf{g}}^{*} (H'', \emptyset, H')}{H \mid k \mid \eta > e \mapsto_{\mathsf{a}} H' \mid k \mid \eta > e}\ \mathit{Collect}$$

Given our definition of a garbage collector, we might then 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).