Part XIV

Storage Management

Chapter 31

Storage Management

The dynamic semantics for MinML given in Chapter 8, and even the C-machine given in Chapter 11, ignore questions of storage management. In particular, all values, be they integers, booleans, functions, or tuples, are treated the same way. But this is unrealistic. Physical machines are capable of handling only rather "small" values, namely those that can fit into a word. Thus, while it is reasonable to treat, say, integers and booleans as values directly, it is unreasonable to do the same with "large" objects such as tuples or functions.

In this chapter we consider an extension of the C-machine to account for storage management. We proceed in two steps. First, we give an abstract machine, called the A-machine, that includes a *heap* for allocating "large" objects. This introduces the problem of *garbage*, storage that is allocated for values that are no longer needed by the program. This leads to a discussion of *automatic storage management*, or *garbage collection*, which allows us to reclaim unused storage in the heap.

31.1 The A Machine

The A-machine is defined for an extension of MinML in which we add an additional form of expression, a *location*, l, which will serve as a "reference" or "pointer" into the heap.

Values are classified into two categories, *small* and *large*, by the following rules:

$$\frac{(l \in Loc)}{l \ \mathsf{svalue}} \tag{31.1}$$

$$\frac{(n \in \mathbb{Z})}{n \text{ svalue}} \tag{31.2}$$

$$\frac{}{\texttt{true} \text{ svalue}} \tag{31.3}$$

$$\frac{}{\texttt{false} \text{ svalue}} \tag{31.4}$$

$$\frac{x \text{ var} \quad y \text{ var} \quad e \text{ expr}}{\texttt{fun}\, x\, \texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e\, \texttt{end} \text{ lvalue}} \tag{31.5}$$

A state of the A-machine has the form (H, k, e), where H is a *heap*, a finite function mapping locations to large values, k is a *control stack*, and e is an expression. A heap H is said to be *self-contained* iff $\mathrm{FL}(H) \subseteq \mathrm{dom}(H)$, where $\mathrm{FL}(H)$ is the set of locations occuring free in any location in H, and $\mathrm{dom}\, H$ is the domain of H.

Stack frames are similar to those of the C-machine, but refined to account for the distinction between small and large values.

$$\frac{e_2 \text{ expr}}{+(\square, e_2) \text{ frame}} \tag{31.6}$$

$$\frac{v_1 \text{ svalue}}{+(v_1, \square) \text{ frame}} \tag{31.7}$$

(There are analogous frames associated with the other primitive operations.)

$$\frac{e_1 \text{ expr} \quad e_2 \text{ expr}}{\texttt{if}\, \square\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \text{ frame}} \tag{31.8}$$

$$\frac{e_2 \text{ expr}}{\texttt{apply(}\square, e_2\texttt{)} \text{ frame}} \tag{31.9}$$

$$\frac{v_1 \text{ svalue}}{\texttt{apply(}v_1, \square\texttt{)} \text{ frame}} \tag{31.10}$$

Notice that v_1 is required to be a *small* value; a function is represented by a location in the heap, which is small.

As with the C-machine, a stack is a sequence of frames:

$$\frac{}{\bullet \text{ stack}} \tag{31.11}$$

$$\frac{f \text{ frame} \quad k \text{ stack}}{f \triangleright k \text{ stack}} \tag{31.12}$$

The dynamic semantics of the A-machine is given by a set of rules defining the transition relation $(H, k, e) \mapsto_{\mathsf{A}} (H', k', e')$. The rules are similar to those for the C-machine, except for the treatment of functions.

Arithmetic expressions are handled as in the C-machine:

$$(H, k, +(e_1, e_2)) \mapsto_{\mathsf{A}} (H, +(\Box, e_2) \triangleright k, e_1) \tag{31.13}$$

$$(H, +(\Box, e_2) \triangleright k, v_1) \mapsto_{\mathsf{A}} (H, +(v_1, \Box) \triangleright k, e_2) \tag{31.14}$$

$$(H, +(n_1, \Box) \triangleright k, n_2) \mapsto_{\mathsf{A}} (H, k, n_1 + n_2) \tag{31.15}$$

Note that the heap is simply "along for the ride" in these rules.

Booleans are also handled similarly to the C-machine:

$$\begin{array}{c}(H, k, \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}) \\ \mapsto_{\mathsf{A}} \\ (H, \texttt{if}\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \triangleright k, e)\end{array} \tag{31.16}$$

$$(H, \texttt{if}\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \triangleright k, \texttt{true}) \mapsto_{\mathsf{A}} (H, k, e_1) \tag{31.17}$$

$$(H, \texttt{if}\, \Box\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \triangleright k, \texttt{false}) \mapsto_{\mathsf{A}} (H, k, e_2) \tag{31.18}$$

Here again the heap plays no essential role.

The real difference between the C-machine and the A-machine is in the treatment of functions. A function expression is no longer a (small) value, but rather requires an execution step to allocate it on the heap.

$$\begin{array}{c}(H, k, \texttt{fun}\, x\, \texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e\, \texttt{end}) \\ \mapsto_{\mathsf{A}} \\ (H[l \mapsto \texttt{fun}\, x\, \texttt{(}y\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e\, \texttt{end}], k, l)\end{array} \tag{31.19}$$

where l is chosen so that $l \notin \operatorname{dom} H$.

Evaluation of the function and argument position of an application is handled similarly to the C-machine.

$$(H, k, \texttt{apply(}e_1, e_2\texttt{)}) \mapsto_{\mathsf{A}} (H, \texttt{apply(}\Box, e_2\texttt{)} \triangleright k, e_1) \tag{31.20}$$

$$(H, \mathtt{apply(}\Box, e_2\mathtt{)} \triangleright k, v_1) \mapsto_{\mathsf{A}} (H, \mathtt{apply(}v_1, \Box\mathtt{)} \triangleright k, e_2) \tag{31.21}$$

Execution of a function call differs from the corresponding C-machine instruction in that the function must be retrieved from the heap in order to determine the appropriate instance of its body. Notice that the *location* of the function, and not the function itself, is substituted for the function variable!

$$\frac{v_1 \text{ loc} \quad H(v_1) = \mathtt{fun}\, f\, \mathtt{(}x\mathtt{:}\tau_1\mathtt{)}\mathtt{:}\tau_2\, \mathtt{is}\, e\, \mathtt{end}}{(H, \mathtt{apply(}v_1, \Box\mathtt{)} \triangleright k, v_2) \mapsto_{\mathsf{A}} (H, k, \{v_1, v_2/f, x\}e)} \tag{31.22}$$

The A-machine preserves self-containment of the heap. This follows from observing that whenever a location is allocated, it is immediately given a binding in the heap, and that the bindings of heap locations are simply those functions that are encountered during evaluation.

Lemma 104
If H is self-contained and $(H, k, e) \mapsto_{\mathsf{A}} (H', k', e')$, then H' is also self-contained. Moreover, if $\mathrm{FL}(k) \cup \mathrm{FL}(e) \subseteq \mathrm{dom}\, H$, then $\mathrm{FL}(k') \cup \mathrm{FL}(e') \subseteq \mathrm{dom}\, H'$.

It is not too difficult to see that the A-machine and the C-machine have the same "observable behavior" in the sense that both machines determine the same value for closed expressions of integer type. However, it is somewhat technically involved to develop a precise correspondence. The main idea is to define the *heap expansion* of an A-machine state to be the C-machine state obtained by replacing all locations in the stack and expression by their values in the heap. (It is important to take care that the locations occurring in a value stored are themselves replaced by their values in the heap!) We then prove that an A-machine state reaches a final state in accordance with the transition rules of the A-machines iff its expansion does in accordance with the rules of the C-machine. Finally, we observe that the value of a final state of integer type is the same for both machines.

Formally, let $\widehat{H}(e)$ stand for the substitution

$$\{H(l_1), \ldots, H(l_n)/l_1, \ldots, l_n\}e,$$

where $\mathrm{dom}\, H = \{\, l_1, \ldots, l_n \,\}$. Similarly, let $\widehat{H}(k)$ denote the result of performing this substitution on every expression occurring in the stack k.

Theorem 105
If $(H, k, e) \mapsto_{\mathsf{A}} (H', k', e')$*, then* $(\widehat{H}(k), \widehat{H}(e)) \mapsto_{\mathsf{C}}^{0,1} (\widehat{H'}(k'), \widehat{H'}(e'))$.

Notice that the allocation of a function in the A-machine corresponds to zero steps of execution on the C-machine, because in the latter case functions are values.

31.2 Garbage Collection

The purpose of the A-machine is to model the memory allocation that would be required in an implementation of MinML. This raises the question of *garbage*, storage that is no longer necessary for a computation to complete. The purpose of a *garbage collector* is to reclaim such storage for further use. Of course, in a purely abstract model there is no reason to perform garbage collection, but in practice we must contend with the limitations of finite, physical computers. For this reason we give a formal treatment of garbage collection for the A-machine.

The crucial issue for any garbage collector is to determine which locations are unnecessary for computation to complete. These are deemed garbage, and are reclaimed so as to conserve memory. But when is a location unnecessary for a computation to complete? Consider the A-machine state (H, k, e). A location $l \in \mathrm{dom}(H)$ is *unnecessary*, or *irrelevant*, for this machine state iff execution can be completed without referring to the contents of l. That is, $l \in \mathrm{dom}\, H$ is unnecessary iff $(H, k, e) \mapsto_{\mathsf{A}}^* (H', \bullet, v)$ iff $(H_l, k, e) \mapsto_{\mathsf{A}}^* (H'', \bullet, v)$, where H_l is H with the binding for l removed, and H'' is some heap.

Unfortunately, a machine cannot decide whether a location is unnecessary!

Theorem 106
It is mechanically undecidable whether or not a location l *is unnecessary for a given state of the* A*-machine.*

Intuitively, we cannot decide whether l is necessary without actually running the program. It is not hard to formulate a reduction from the halting problem to prove this theorem: simply arrange that l is used to complete a computation iff some given Turing machine diverges on blank input.

Given this fundamental limitation, practical garbage collectors must employ a *conservative approximation* to determine which locations are unnecessary in a given machine state. The most popular criterion is based

on *reachability*. A location l_n is *unreachable*, or *inaccessible*, iff there is no sequence of locations $l_1, \ldots, l_n$ such that l_1 occurs in either the current expression or on the control stack, and l_i occurs in l_{i+1} for each $1 \leq i < n$.

Theorem 107
If a location l is unreachable in a state (H, k, e), then it is also unnecessary for that state.

Each transition depends only on the locations occurring on the control stack or in the current expression. Some steps move values from the heap onto the stack or current expression. Therefore in a multi-step sequence, execution can depend only on reachable locations in the sense of the definition above.

The set of unreachable locations in a state may be determined by *tracing*. This is easily achieved by an iterative process that maintains a finite set of of locations, called the *roots*, containing the locations that have been found to be reachable up to that point in the trace. The root set is initialized to the locations occurring in the expression and control stack. The tracing process completes when no more locations can be added. Having found the reachable locations for a given state, we then deem all other heap locations to be unreachable, and hence unnecessary for computation to proceed. For this reason the reachable locations are said to be *live*, and the unreachable are said to be *dead*.

Essentially *all* garbage collectors used in practice work by tracing. But since reachability is only a conservative approximation of necessity, *all practical collectors are conservative*! So-called conservative collectors are, in fact, *incorrect* collectors that may deem as garbage storage that is actually necessary for the computation to proceed. Calling such a collector "conservative" is misleading (actually, wrong), but it is nevertheless common practice in the literature.

The job of a garbage collector is to dispose of the unreachable locations in the heap, freeing up memory for later use. In an abstract setting where we allow for heaps of unbounded size, it is never necessary to collect garbage, but of course in practical situations we cannot afford to waste unlimited amounts of storage. We will present an abstract model of a particular form of garbage collection, called *copying collection*, that is widely used in practice. The goal is to present the main ideas of copying collection, and to prove that garbage collection is semantically "invisible" in the sense that it does not change the outcome of execution.

The main idea of copying collection is to simultaneously determine which locations are reachable, and to arrange that the contents of all reachable locations are preserved. The rest are deemed garbage, and are reclaimed. In a copying collector this is achieved by partitioning storage into two parts, called *semi-spaces*. During normal execution allocation occurs in one of the two semi-spaces until it is completely filled, at which point the collector is invoked. The collector proceeds by copying all reachable storage from the current, filled semi-space, called the *from space*, to the other semi-space, called the *to space*. Once this is accomplished, execution continues using the "to space" as the new heap, and the old "from space" is reclaimed in bulk. This exchange of roles is called a *flip*.

By copying all and only the reachable locations the collector ensures that unreachable locations are reclaimed, and that no reachable locations are lost. Since reachability is a conservative criterion, the collector may preserve more storage than is strictly necessary, but, in view of the fundamental undecidability of necessity, this is the price we pay for mechanical collection. Another important property of copying collectors is that their execution time is proportion to the size of the live data; no work is expended manipulating reclaimable storage. This is the fundamental motivation for using semi-spaces: once the reachable locations have been copied, the unreachable ones are eliminated by the simple measure of "flipping" the roles of the spaces. Since the amount of work performed is proportional to the live data, we can amortize the cost of collection across the allocation of the live storage, so that garbage collection is (asymptotically) "free". However, this benefit comes at the cost of using only half of available memory at any time, thereby doubling the overall storage required.

Copying garbage collection may be formalized as an abstract machine with states of the form (H_f, S, H_t), where H_f is the " from" space, H_t is the "to" space, and S is the *scan set*, the set of reachable locations. The initial state of the collector is $(H, S, \emptyset)$, where H is the "current" heap and $\emptyset \neq S \subseteq \mathrm{dom}(H_f)$ is the set of locations occurring in the program or control stack. The final state of the collector is $(H_f, \emptyset, H_t)$, with an empty scan set.

The collector is invoked by adding the following instruction to the A-machine:

$$\frac{(H, \mathrm{FL}(k) \cup \mathrm{FL}(e), \emptyset) \mapsto_{\mathsf{G}}^{*} (H'', \emptyset, H')}{(H, k, e) \mapsto_{\mathsf{A}} (H', k, e)} \tag{31.23}$$

The scan set is initialized to the set of free locations occurring in either the current stack or the current expression. These are the locations that are immediately reachable in that state; the collector will determine those that

are transitively reachable, and preserve their bindings. Once the collector has finished, the "to" space is installed as the new heap.

Note that a garbage collection can be performed at any time! This correctly models the unpredictability of collection in an implementation, but avoids specifying the exact criteria under which the collector is invoked. As mentioned earlier, this is typically because the current heap is exhausted, but in an abstract setting we impose no fixed limit on heap sizes, preferring instead to simply allow collection to be performed spontaneously according to unspecified criteria.

The collection machine is defined by the following two rules:

$$\frac{}{(H_f[l = v], S \cup \{\, l \,\}, H_t) \mapsto_{\mathsf{G}} (H_f, S \cup \mathrm{FL}(v), H_t[l = v])} \tag{31.24}$$

$$\frac{}{(H_f, S \cup \{\, l \,\}, H_t[l = v]) \mapsto_{\mathsf{G}} (H_f, S, H_t[l = v])} \tag{31.25}$$

The first rule copies a reachable binding in the "from" space to the "to" space, and extends the scan set to include those locations occurring in the copied value. This ensures that we will correctly preserve those locations that occur in a reachable location. The second rule throws away any location in the scan set that has already been copied. This rule is necessary because when the scan set is updated by the free locations of a heap value, we may add locations that have already been copied, and we do not want to copy them twice!

The collector is governed by a number of important invariants.

1. The scan set contains only "valid" locations: $S \subseteq \mathrm{dom}\, H_f \cup \mathrm{dom}\, H_t$;
2. The "from" and "to" space are disjoint: $\mathrm{dom}\, H_f \cap \mathrm{dom}\, H_t = \emptyset$;
3. Every location in "to" space is either in "to" space, or in the scan set: $\mathrm{FL}(H_t) \subseteq S \cup \mathrm{dom}\, H_t$;
4. Every location in "from" space is either in "from" or "to" space: $\mathrm{FL}(H_f) \subseteq \mathrm{dom}\, H_f \cup \mathrm{dom}\, H_t$.

The first two invariants are minimal "sanity" conditions; the second two are crucial to the operation of the collector. The third states that the "to" space contains only locations that are either already copied into "to" space, or will eventually be copied, because they are in the scan set, and hence in "from" space (by disjointness). The fourth states that locations in "from" space contain only locations that either have already been copied or are yet to be copied.

These invariants are easily seen to hold of the initial state of the collector, since the "to" space is empty, and the "from" space is assumed to be self-contained. Moreover, if these invariants hold of a final state, then $\mathrm{FL}(H_t) \subseteq \mathrm{dom}\, H_t$, since $S = \emptyset$ in that case. Thus the heap remains self-contained after collection.

Theorem 108 (Preservation of Invariants)
If the collector invariants hold of (H_f, S, H_t) *and* $(H_f, S, H_t) \mapsto_{\mathsf{G}} (H'_f, S', H'_t)$, *then the same invariants hold of* (H'_f, S', H'_t).

The correctness of the collector follows from the following lemma.

Lemma 109
If $(H_f, S, H_t) \mapsto_{\mathsf{G}} (H'_f, S', H'_t)$, *then* $H_f \cup H_t = H'_f \cup H'_t$ *and* $S \cup \mathrm{dom}\, H_t \subseteq S' \cup \mathrm{dom}\, H'_t$.

The first property states that the union of the semi-spaces never changes; bindings are only copied from one to the other. The second property states that the domain of the "to" space together with the scan set does not change.

From this lemma we obtain the following crucial facts about the collector. Let $S = \mathrm{FL}(k) \cup \mathrm{FL}(e)$, and suppose that

$$(H, S, \emptyset) \mapsto^*_{\mathsf{G}} (H'', \emptyset, H').$$

Then we have the following properties:

1. The reachable locations are bound in H': $\mathrm{FL}(k) \cup \mathrm{FL}(e) \subseteq \mathrm{dom}\, H'$. This follows from the lemma, since the inital "to" space and the final scan set are empty.

2. The reachable data is correctly copied: $H' \subseteq H$. This follows from the lemma, which yields $H = H'' \cup H'$.