STACK-BASED STORAGE

The last language showed how the idea of different contexts for names can be handled by providing an environment that maps identifiers to locations. When a new environment was created, we simply modified (a copy of) the old environment by updating the function that represented it. The property of updated functions that means that the more recent of the updates takes precedence over the earlier ones when the same identifier is used. i.e.

\[ e_2 = \left[ [X] \mapsto l_0 \right] e_1 \]

maps \( X \) to \( \text{one} \), whereas

\[ e_3 = \left[ [X] \mapsto l_i \right] e_2 \]

maps \( X \) to \( l_i \), since the earlier mapping is ignored.

This was coupled with the execution of a sequence of commands using the same environment but different stores:

\[ \text{C}[\text{C}_1; \text{C}_2] = \lambda e. (\text{C}[\text{C}_2] e) \circ (\text{C}[\text{C}_1] e) \]

If we put the store back into the equation we get:

\[ \text{C}[\text{C}_1; \text{C}_2] e s = (\text{C}[\text{C}_2] e) (\text{C}[\text{C}_1] e s) \]

The store produced from \( \text{C}_1 \) is passed to \( \text{C}_2 \), but the environment stays the same. If new variables were declared in \( \text{C}_1 \) (if it was a block) then they are forgotten when \( \text{C}_2 \) is executed. The store retains the new locations, but they are inaccessible. A picture will make this clear. Let \( e \) contain two variables \( X \) and \( Y \). \( s \) then contains two locations for these two variables. If \( \text{C}_1 \) declares a variable \( Z \), the assignment in the block will create a new store with \( Z \)'s location, \( l_3 \), mapped to \( \text{three} \). Thus we will have:

\[
\begin{align*}
\left( \text{C}[\text{C}_2] \begin{bmatrix} X & l_i \cr Y & l_2 \end{bmatrix} \right) \left( \text{C}[\begin{bmatrix} \text{begin var \( Z \); \( Z := 3 \)\end{bmatrix}} \begin{bmatrix} X & l_i \cr Y & l_2 \end{bmatrix} \begin{bmatrix} l_1 & \text{one} \cr l_2 & \text{two} \end{bmatrix} \right)
\end{align*}
\]

\[
= \left( \text{C}[\text{C}_2] \begin{bmatrix} X & l_i \cr Y & l_2 \end{bmatrix} \right) \left( \text{C}[Z := 3] \begin{bmatrix} X & l_i \cr Y & l_2 \end{bmatrix} \begin{bmatrix} l_1 & \text{one} \cr l_2 & \text{two} \end{bmatrix} \right)
\]

\[
= \left( \text{C}[\text{C}_2] \begin{bmatrix} X & l_i \cr Y & l_2 \end{bmatrix} \begin{bmatrix} l_3 \cr l_1 \cr l_2 \end{bmatrix} \begin{bmatrix} \text{three} \cr \text{one} \cr \text{two} \end{bmatrix} \right)
\]

This is not exactly what we expect in a real language. In fact this the semantics of memory leakage! The only good thing is that if \( \text{C}_2 \) is a block, then any new variables will reuse location \( l_3 \) and higher, but the store does not either allocate them or deallocate them. In contrast, most modern languages exhibit a stack-based discipline in the store, at run-time. We will modify our language to do just that. This will again show the power of the denotational method to explore different operational behavior in a safe denotational language.

STACK-BASED STORAGE

The problem with our language is that the allocation of space (the reservation of locations) is in the environment, whereas it should be in the store. So instead of doing the allocation in the environment algebra, we will move it to the store domain. i.e. from

Domain \( \text{Environment} = (\text{Id} \rightarrow \text{Denotable - value}) \times \text{Location} \)

Domain \( \text{Store} = (\text{Location} \rightarrow \text{Storeable - value}) \)
we move to

Domain \( Environment = Id \rightarrow Denotable\cdot value \)

Domain \( Store = ( Location \rightarrow Storeable\cdot value ) \times Location \)

Here the \( Location \) part of the store will be like the top-of-stack marker, and the algebra will be responsible for allocation and de-allocation. The \textit{access} operation tests whether the location it receives is below top-of-stack, and retrieves the value if it is. If it is not, then it returns an error store (\( Err\text{value} = Unit \)).

\[
\text{access} : Location \rightarrow Store \rightarrow ( Storeable\cdot value + Err\text{value} )
\]

\[
\text{access} = \lambda l.\lambda [map, top]. l \less than - locn top \rightarrow \text{inStoreable}\cdot value ( map l )
\]

\[\Box \text{inErr}(\text{map, top})\]

\textit{update} updates the map, but leaves top-of-stack unchanged.

\[
\text{update} : Location \rightarrow Storable\cdot value \rightarrow Store \rightarrow Poststore
\]

\[
\text{update} = \lambda l.\lambda v.\lambda [map, top]. l \less than - locn top \rightarrow \text{inOK}(\{ [l \mapsto v]. map, top \})
\]

\[\Box \text{inErr}(\text{map, top})\]

\textit{mark-locn} simply returns the top-of-stack location.

\[
\text{mark-locn} : Store \rightarrow Location
\]

\[
\text{mark-locn} = \lambda [map, top]. top
\]

Allocation is done by using \textit{next-locn} by “pushing” a new location on the stack. It returns the current top-of-stack as well, so that it can be restored on exit from a block.

\[
\text{allocate-locn} : Store \rightarrow ( Location \times Poststore )
\]

\[
\text{allocate-locn} = \lambda [map, top]. [ top, \text{inOK}(\{ [map, next-locn(top)] \}) ]
\]

Deallocation is done by “popping” the stack down to a given location. It must, of course, be under (or equal to) top-of-stack for this to happen, otherwise an error store is returned.

\[
\text{deallocate-locns} : Location \rightarrow Store \rightarrow Poststore
\]

\[
\text{deallocate-locns} = \lambda l.\lambda [map, top]. l \less than - locn top \text{ or } l \text{ equal - locn top } \rightarrow
\]

\[\text{inOK}(\{ [map, l] \})
\]

\[\Box \text{inErr}(\text{map, top})\]

The valuation functions also have to change since the store is now actively involved in declarations.

\textbf{D}: Declaration \( \rightarrow Environment \rightarrow Store \rightarrow ( Environment \times Poststore ) \)

\[
\text{D}[\text{var } l] = \lambda e.\lambda s.\text{let } [l, p] = \text{allocate-locn } s \text{ in }
\]

\[
\text{updateenv}(\{ l \mapsto \text{inLocation}(l)e \}, p)
\]

\[
\text{D}[D_1; D_2] = \lambda e.\lambda s.\text{let } [e', p] = (\text{D}[D_1] e\ s) \text{ in } (\text{check } \text{D}[D_1] e')(p)
\]

Now both environment and store change when a new variable is declared. The environment maps the identifier to the new location returned by running \textit{allocate-locn} on the store, which is altered to reflect the allocation. The block function also changes:
As an example of the new store in action, consider the program

```
begin
var X;
X := 1;
begin
var X;
X := 2
end;
X := 3
end.
```

If we start with an empty environment, $e_0 = \lambda i.\text{in}Errorvalue()$ and an empty store, $s_0 = [\lambda i.\text{zero}, l_0]$, then we get:

- The sequence in $\mathbf{K}$ is:
  1. get the top-of-stack location from the store
  2. for each declaration, add the next location to the environment and put it in the store as top-of-stack
  3. execute the block body with the new environment and store
  4. deallocate the new locations from the store by resetting top-of-stack

So we get:

$$\mathbf{K}[\text{begin } \ldots \text{end}] e_0 s_0 =$$

Let $l = \text{mark-locn} s_0$ in

$$\text{let } [e', p] = \mathbf{D}[D] e_0 s_0 \text{ in}$$

$$\text{let } p' = (\text{check } (C[C][e'])(p) \text{ in}$$

$$\text{(check } (\text{dealloate-locns } l))(p')$$

Executing $X := 1$ gives a store (tagged as an OK poststore) $s_2 = [[l_0 \mapsto \text{one}] (\lambda i.\text{zero}), l_1]$. Now when this is handed to $\text{deallocate-locns}$, it is “popped” back to the top-of-stack at the start of the block, $l_1$. This means that $l_1$ itself cannot be accessed any longer since $\text{access}$ checks that any location must be less than top-of-stack.
The final assignment is executed with $e_i = [[X] \mapsto l_0]e_0$ and the (tagged) store $s_5 = [[l_1 \mapsto \text{two}][l_0 \mapsto \text{one}](\lambda i.\text{zero}), l_1]$, giving a final store $s_6 = [[l_1 \mapsto \text{two}][l_0 \mapsto \text{three}](\lambda i.\text{zero}), l_1]$. Finally this store is handed to \textit{deallocate-locns}, which is popped back to $[[l_1 \mapsto \text{two}][l_0 \mapsto \text{three}](\lambda i.\text{zero}), l_0]$.

Although the store still contains mappings for the two locations corresponding to the two versions of $X$, they are no longer accessible because of the form of \textit{access}. This is much closer to the way real languages operate their memory management.