1 Introduction

In the S machine, we still freely substitute into expressions, which goes somewhat against the idea that expressions should be compiled. Also, we directly store expressions in memory cells, even though their space requirements are not clear and not small.

In this lecture we first review the S machine and then update it to avoid substitution into expressions. Instead we construct environments to hold the bindings for the variables in an expression and then closures to pair up an environment with an expression as a closed value.

2 Semantic Objects in the S Machine

We briefly summarize the S machine from the previous lecture. At its core are destinations d (also called addresses) to hold values in the store. The only operation on them is to generate fresh ones. The state of the S machine consists of the following objects:

eval e d. Evaluate expression e, storing the result in destination d.

!cell d c. Cell d has contents c. The exclamation mark ‘!’ indicates that cells are persistent, which means the value of a cell can never change and will be available during the whole remainder of the computation.

cont d k d’. Continuation k receives a value in destination in d and puts result into d’.
First, a summary of the three types we have considered so far.

**Expressions**

\[ e ::= \begin{array}{l}
  x \\
  d \\
  \lambda x. e \\
  e_1 e_2 \\
  \langle \rangle \\
  \langle e_1, e_2 \rangle \\
  \langle x_1, x_2 \rangle \\
\end{array} \]

*variables*  
*destinations*  
\(\rightarrow\)  
\(\otimes\)  
\(\oplus\)  
\(\&\)

**Continuations**

\[ k ::= \begin{array}{l}
  (\_ e_2) (d_1 \_) \\
  \text{case } (\_ ) \Rightarrow e' \\
  \langle , e_2 \rangle (d_1 , \\_ ) \text{ case } \langle x_1 , x_2 \rangle \Rightarrow e' \\
\end{array} \]

**Cell Contents**

\[ c ::= \begin{array}{l}
  \langle \rangle \\
  \langle d_1, d_2 \rangle \\
  \lambda x. e \\
\end{array} \]

From these examples we can readily extrapolate the rest of the S machine. We just show the possible cell contents, organized by type, thereby describing the possible shapes of memory.

**Store Typing**

\[ \Sigma ::= d_1 : \tau_1, \ldots, d_n : \tau_n \]

where all the destinations \(d_i\) are distinct. We type semantics objects as

\[ \begin{array}{l}
  \Sigma \vdash d : \tau \\
  \Sigma \vdash e : \tau \\
  \Sigma \vdash (\text{eval } e \ d \ \text{obj}) \\
  \Sigma \vdash (\text{cell } d \ c \ \text{obj}) \\
  \Sigma \vdash d_1 : \tau_1 \\
  \Sigma \vdash d_2 : \tau_2 \\
  \Sigma \vdash k \div \tau_1 \Rightarrow \tau_2 \\
  \Sigma \vdash (\text{cont } d_1 \ k \ d_2 \ \text{obj}) \\
\end{array} \]
The dynamics is given with the following rules:

\[
\begin{align*}
\Sigma \vdash \langle \rangle : 1 & \quad (C-1) \\
\Sigma \vdash d : \tau & \quad (C-\tau) \\
\Sigma \vdash \langle d, d \rangle : \tau \otimes \tau & \quad (C-\otimes)
\end{align*}
\]

\[
\begin{align*}
\Sigma \vdash i : \tau & \quad (C-\tau) \\
\Sigma \vdash \langle i \rangle \cdot d : \sum_{\ell \in L} (\ell : \tau) & \quad (C+\tau)
\end{align*}
\]

\[
\begin{align*}
\Sigma \vdash e_1 : \tau & \quad (C-\&)
\end{align*}
\]

\[
\begin{align*}
\Sigma \vdash \langle e_1, e_2 \rangle : \tau \& \tau & \quad (C-\&)
\end{align*}
\]

The dynamics is given with the following rules:

\[
\begin{align*}
!\text{cell } d \ c, \text{eval } d \ d' & \Rightarrow !\text{cell } d' \ c \\
\text{eval } \langle \rangle \ d & \Rightarrow !\text{cell } d \ \langle \rangle \\
\text{eval } (\text{case } e \ {\{\langle \rangle \Rightarrow e'\}}) \ d' & \Rightarrow \text{eval } e \ d, \ \text{cont } d \ (\text{case } _- {\{\langle \rangle \Rightarrow e'\}}) \ d' \quad (d \ \text{fresh}) \\
!\text{cell } d \ \langle \rangle, \ \text{cont } d \ (\text{case } _- {\{\langle \rangle \Rightarrow e'\}}) \ d' & \Rightarrow \text{eval } e' \ d' \\
\text{eval } (\lambda x. e) \ d & \Rightarrow !\text{cell } d \ (\lambda x. e) \\
\text{eval } (e_1 \ e_2) \ d & \Rightarrow \text{eval } e_1 \ d_1, \ \text{cont } d_1 \ (_- e_2) \ d \quad (d_1 \ \text{fresh}) \\
!\text{cell } d_1 \ c_1, \ \text{cont } d_1 \ (_- e_2) \ d & \Rightarrow \text{eval } e_2 \ d_2, \ \text{cont } d_2 \ (_- d_1) \ d \quad (d_2 \ \text{fresh}) \\
!\text{cell } d_1 \ (\lambda x. e'_1), !\text{cell } d_2 \ c_2, \ \text{cont } d_2 \ (_- d_1) \ d & \Rightarrow \text{eval } ([d_2/x]e'_1) \ d \\
\text{eval } (e_1, e_2) \ d & \Rightarrow \text{eval } e_1 \ d_1, \ \text{cont } d_1 \ (_- e_2) \ d \quad (d_1 \ \text{fresh}) \\
!\text{cell } d_1 \ c_1, \ \text{cont } d_1 \ (_- e_2) \ d & \Rightarrow \text{eval } e_2 \ d_2, \ \text{cont } d_2 \ (_- d_1) \ d \quad (d_2 \ \text{fresh}) \\
!\text{cell } d_2 \ c_2, \ \text{cont } d_2 \ (_- d_1) \ d & \Rightarrow !\text{cell } d \ (_- d_2) \\
\text{eval } (\text{case } e \ {\{\langle x_1, x_2 \rangle \Rightarrow e'\}}) \ d' & \Rightarrow \text{eval } e \ d, \ \text{cont } d \ (\text{case } _- {\{\langle x_1, x_2 \rangle \Rightarrow e'\}}) \ d' \quad (d \ \text{fresh}) \\
!\text{cell } d \ (_- d_1), \ \text{cont } d \ (\text{case } _- {\{\langle x_1, x_2 \rangle \Rightarrow e'\}}) \ d' & \Rightarrow \text{eval } ([d_1/x_2, d_2/x_2]e') \ d' \\
\text{eval } (\text{fix } x. e) \ d & \Rightarrow \text{eval } ([\text{fix } x. e/x]e) \ d
\end{align*}
\]

### 3 Environments

For the eager constructs of the language, this representation of values in the store is adequate, if perhaps consuming a bit too much space. For example,
the value 1 at destination \(d_0\) would be

\[
!\text{cell } d_0 (\text{fold } d_1), \\
!\text{cell } d_1 (s \cdot d_2), \\
!\text{cell } d_2 (\text{fold } d_3), \\
!\text{cell } d_3 (z \cdot d_4), \\
!\text{cell } d_4 \langle \rangle
\]

Up to a constant factor, this is what one might expect.

However, expressions such as the values \(\lambda x. e\) and \(\langle | e_1, e_2 | \rangle\) are treated not quite in the way we might envision in a lower-level semantics. Functions should be compiled to efficient machine code, which is justified in part by saying that we can not observe their internal forms. Moreover, in the dynamics of the S machine we substitute destinations into expressions to obtain new ones that we then evaluate. In a lower-level implementation, such a substitution is unrealistic. Instead, we compile variables so they reference the store, either on a stack or in the heap. While we don’t model this distinction here, we would still like to model that code is essentially immutable, and the values held in variables are stored in memory.

The first key idea is not to substitute into an expression, but instead maintain an environment that maps variables to values. In the case of the K machine, these values would be the same as we had in our original, high-level semantics. In the case of the S machine, the values are simply store addresses where the value is represented.

\[
\text{Environments } \eta ::= d_1/x_1, \ldots, d_n/x_n
\]

We require that all the variables \(x_i\) are distinct so that the value of each variable is uniquely determined. The destinations \(d_i\) however do not need to be distinct: it is perfectly possible that two different program variables contain references to the same storage cell.

Previously we were careful to evaluate only closed expressions. Now we evaluate expressions in an environment that substitutes destinations for all of its free variables. Of course, the type of the destination must match the type of the variables it substitutes for. To ensure this we use the typing judgment \(\Sigma \vdash \eta : \Gamma\) defined by the two rules

\[
\begin{aligned}
\Sigma \vdash (\cdot) : (\cdot) & \\
\Sigma \vdash (\eta, d/x) : (\Gamma, x : \tau)
\end{aligned}
\]

Now evaluation depends on an environment

\[
\begin{aligned}
\Sigma \vdash d : \tau & \\
\Sigma \vdash \eta : \Gamma & \\
\Gamma \vdash e : \tau
\end{aligned} \\
\Sigma \vdash \text{eval } \eta e d \text{ obj}
\]
Compared to the S machine in the previous lecture, expressions now no longer contain destinations, so the typing judgments for expressions revert to be pure, $\Gamma \vdash e : \tau$.

4 Evaluation with Environments

Now we revisit the rules of the S machine in the presence of environments. Let’s call this new machine the $S_\eta$ machine. Previously we had

$$
!\text{cell } d \ c, \text{eval } d' \mapsto !\text{cell } d' \ c \quad \text{(S machine)}
$$

Now, this becomes a rule for variables which must be defined in the environment

$$
!\text{cell } d \ c, \text{eval } \eta x \ d' \mapsto !\text{cell } d' \ c \quad (d/x \in \eta)
$$

For functions, we had

$$
\text{eval } (\lambda x. e) \ d \mapsto !\text{cell } d \ (\lambda x. e) \quad \text{(S machine)}
$$

Now we have to pair up the environment with the $\lambda$-abstraction in order to form a closure. It is called a closure because it “closes up” the expression $e$ all of whose free variables are defined in $\eta$.

$$
\text{eval } \eta (\lambda x. e) \ d \mapsto !\text{cell } d \ (\lambda x. e)
$$

For an application $e_1 e_2$ we have to evaluate $e_1$, but we also have to remember the environment in which $e_2$ makes sense. In a another implementation, this might be captured in an environment stack. Here, we just keep track of the environment in the continuation, building a temporary closure $\langle \eta, e_2 \rangle$. After evaluation of $e_1$ we continue the evaluation of $e_2$ in the saved environment.

$$
\text{eval } \eta \ (e_1 \ e_2) \ d \mapsto \text{eval } \eta \ e_1 \ d_1, \text{cont } d_1 \ (\langle \eta, e_2 \rangle) \ d \quad (d_1 \text{ fresh})
$$

$$
!\text{cell } d_1 \ c_1, \text{cont } d_1 \ (\langle \eta, e_2 \rangle) \ d \mapsto \text{eval } \eta \ e_2 \ d_2, \text{cont } d_2 \ (d_1 \_ \_ \_ ) \ d \quad (d_2 \text{ fresh})
$$

The most interesting rule is the one where we actually pass the argument to the function. Previously, we just substituted the address of the argument value; now we add it to the environment.

$$
!\text{cell } d_1 \ (\eta, \lambda x. e'_1), \!\text{cell } d_2 \ c_2, \text{cont } d_2 \ (d_1 \_ \_ \_ ) \ d \mapsto \text{eval } (\eta, d_2/x) \ e'_1 \ d
$$

It is easy to see that this environment is the correct one. On the left-hand side, given the store typing $\Sigma$, we have

$$
\Sigma \vdash \eta : \Gamma \quad \text{and} \quad \Gamma \vdash \lambda x. e'_1 : \tau
$$
for some $\Gamma$ and $\tau$. By inversion, we also have

$$\Gamma, x : \tau_2 \vdash e'_1 : \tau_1$$

with $\tau = \tau_2 \rightarrow \tau_1$. Also

$$\Sigma \vdash (\eta, d_2/x) : (\Gamma, x : \tau_2)$$

since $\Sigma \vdash d_1 : \tau_2 \rightarrow \tau_1$ and $\Sigma \vdash d_2 : \tau_2$ from inversion on the continuation typing.

There is nothing much interesting in the remaining rules, but we will show those for lazy pairs because they also involve closures precisely because they are lazy.

At this point we might ask if we have actually satisfied our goal of storing only data of fixed size. We imagine that in an implementation the code is compiled, with variables becoming references into an environment. Then the expression part of a closure is a pointer to code that in turn expect to be passed the address of the environment. As such, it is only the size of the environment which is of variable size. However, it can be predicted at the time of compilation. In our simple model, it consists of bindings for all variables that might occur free in $e$, that is, all variables in $\Gamma$ if $e$ was checked with $\Gamma \vdash e : \tau$. We can slightly improve on this, keeping only the variables of $\Gamma$ that actually occur free in $e$. Thus, while the space for different closures is of different size, we can calculate it at compile time, and it is proportional to the number of free variables in $e$.

## Fixed Points

Fixed points are interesting. The rule of the S machine

$$\text{eval } (\text{fix } x. e) \ d \ \mapsto \ \text{eval } ([\text{fix } x. e/x] e) \ d$$

(and also the corresponding rule of the K machine) substitutes an expression for a variable, while all other rules in our call-by-value language just substitute either values (K machine) or destinations (S machine). Since one of our goals is to eliminate substitution into expressions, we should change
this rule somehow. First idea might be to just add the expression to the environment, but a rule such as

\[
\text{eval } \eta (\text{fix } x. e) \ d \mapsto \text{eval } (\eta, (\text{fix } x. e)/x) \ e \ d \quad ??
\]

would add expressions to the environment, upsetting our carefully constructed system. In particular, looking up a variable doesn’t necessarily result in a destination. Perhaps even worse, the expression fix \(x. e\) is not closed, so at the very least we’d have to construct another closure.

\[
\text{eval } \eta (\text{fix } x. e) \ d \mapsto \text{eval } (\eta, (\eta, \text{fix } x. e)/x) \ e \ d \quad ??
\]

We pursue here a different approach, namely evaluating the body of the fixed point as if \(d\) already held a value!

\[
\text{eval } \eta (\text{fix } x. e) \ d \mapsto \text{eval } (\eta, d/x) \ e \ d
\]

This upsets another invariant of our semantics so far, namely that any destination in the environment is defined in the store. This new rule speculates that \(d\) will contain a value by the time \(e\) might reference the variable \(x\). This is not a trivial matter. Consider the expression fix \(x. x\) in the empty environment (\(\cdot\)).

\[
\text{eval } (\cdot) (\text{fix } x. x) \ d_0 \mapsto \text{eval } (d_0/x) \ x \ d_0
\]

At this point the rule for variables

\[
!\text{cell } d \ c. \text{eval } \eta \ x \ d' \mapsto !\text{cell } d' \ c \ (d/x \in \eta)
\]

cannot be applied because destination \(d_0\) does yet hold a value. In other words, the progress property fails!

This situation can indeed arise in Haskell where it is called a black hole. It is actually detected at runtime and a “black hole” is reported during execution. For example,

```haskell
blackHole :: Int
blackHole = blackHole

main = print blackHole
```

compiles, but running it reports

black_hole: <<loop>>
We can imagine how this may be done: when the fixed point is executed we actually allocate a destination for its value and initialize it with a recognizable value indicating it has not yet been written. We may then modify the progress theorem to account for a black hole as a third form of outcome of the computation, besides a value or divergence.

In a call-by-value language there is a different solution: we can restrict the body of the fixed point expression to be a value, where the fixed point variable $x$ does not count as a value. We believe that this guarantees that the destination of the fixed point will always be defined before the fixed point variable $x$ is encountered. The revised rule then reads

$$\text{eval } \eta (\text{fix } x. v) \ d \mapsto \text{eval } (\eta, d/x) v \ d$$

where we have to be careful not to count $x$ as a value. Evaluating the expression $v$ will construct its representation in the store.

As an example, consider the following definition of natural number streams:

$$\begin{align*}
nat & \simeq (z : 1) + (s : \text{nat}) \\
zero & = \text{fold } (z \cdot \langle \rangle) \\
\text{stream} & \simeq \text{nat } \& \text{ stream} \\
\text{zeros} & : \text{ stream} \\
\text{zeros} & = \text{fix } x. \text{fold } \langle |zero, x| \rangle
\end{align*}$$

The stream $\text{zeros}$ corresponds to a potentially unbounded number of zeros, computed lazily. We see that $\text{fold } \langle |zero, x| \rangle$ is a value even if $x$ is not, since any lazy pair is a value. Starting with the empty environment and initial destination $d_0$, we evaluate $\text{zeros}$ as follows:

$$\begin{align*}
\text{eval } (\langle \rangle) (\text{fix } x. \text{fold } \langle |zero, x| \rangle) \ d_0 & \\
& \mapsto \text{eval } (d_0/x) (\text{fold } \langle |zero, x| \rangle) \ d_0 \\
& \mapsto \text{eval } (d_0/x) \langle |zero, x| \rangle \ d_1, \text{cont } d_1 (\text{fold } \_ ) \ d_0 \quad (d_1 \text{ fresh}) \\
& \mapsto !\text{cell } d_1 ((d_0/x), \langle |zero, x| \rangle), \text{cont } d_1 (\text{fold } \_ ) \ d_0 \\
& \mapsto !\text{cell } d_1 ((d_0/x), \langle |zero, x| \rangle), !\text{cell } d_0 (\text{fold } d_1)
\end{align*}$$

At this point we have constructed a store with a circular chain of references: cell $d_0$ contains a reference to $d_1$, and $d_1$ contains a reference to $d_0$ in the

\[1\text{but have not proven} \]
environment stored with the closure. If we define

\[
\begin{align*}
hd & : \text{stream} \rightarrow \text{stream} \\
hd & = \lambda s. (\text{unfold } s) \cdot l \\
tl & : \text{stream} \rightarrow \text{stream} \\
tl & = \lambda s. (\text{unfold } s) \cdot r
\end{align*}
\]

we should be able to check that \(hd\) \text{zeros} returns (a representation of) zero, while \(tl\) \text{zeros} returns (a representation of) zeros.

It is a good exercise to check that the \text{ascending} function below behaves as expected, where \text{ascending} \(\langle n \rangle\) computes an ascending stream of natural numbers starting at \(\langle n \rangle\).

\[
\begin{align*}
succ & : \text{nat} \rightarrow \text{nat} \\
succ & = \lambda n. \text{fold } (s \cdot n) \\
\text{ascending} & : \text{nat} \rightarrow \text{stream} \\
\text{ascending} & = \lambda n. \text{fold } \langle n, \text{ascending } (\text{succ } n) \rangle
\end{align*}
\]