Lecture Notes on Closures

15-814: Types and Programming Languages
Frank Pfenning

Lecture 17
November 1, 2018

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:

$\mathsf{eval}\ e\ d$. Evaluate expression e, storing the result in destination d.

$!\mathsf{cell}\ d\ c$. 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.

$\mathsf{cont}\ d\ k\ d'$. 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.

$$\begin{array}{llcll}
\text{Expressions} & e & ::= & x & \text{(variables)} \\
 & & \mid & d & \text{(destinations)} \\
 & & \mid & \lambda x.\, e \mid e_1\, e_2 & (\to) \\
 & & \mid & \langle\,\rangle \mid \mathbf{case}\ e\ \{\langle\,\rangle \Rightarrow e'\} & (1) \\
 & & \mid & \langle e_1, e_2\rangle \mid \mathbf{case}\ e\ \{\langle x_1, x_2\rangle \Rightarrow e'\} & (\otimes) \\[1ex]
\text{Continuations} & k & ::= & (\_\ e_2) \mid (d_1\ \_) & (\to) \\
 & & \mid & \mathbf{case}\ \_\ \{\langle\,\rangle \Rightarrow e'\} & (1) \\
 & & \mid & \langle \_, e_2\rangle \mid \langle d_1, \_\rangle \mid \mathbf{case}\ \_\ \{\langle x_1, x_2\rangle \Rightarrow e'\} & (\otimes) \\[1ex]
\text{Cell Contents} & c & ::= & \langle\,\rangle \mid \langle d_1, d_2\rangle \mid \lambda x.\, e &
\end{array}$$

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.

$$\begin{array}{llcll}
\text{Cell Contents} & c & ::= & \langle\,\rangle & (1) \\
 & & \mid & \langle d_1, d_2\rangle & (\otimes) \\
 & & \mid & \ell \cdot d & (+) \\
 & & \mid & \mathbf{fold}\ d & (\rho) \\
 & & \mid & \langle\!| e_1, e_2 |\!\rangle & (\&) \\
 & & \mid & \lambda x.\, e & (\to)
\end{array}$$

We assign types to the store by typing each destination and then checking for consistent usage. We use

$$\text{Store Typing} \quad \Sigma \quad ::= \quad d_1 : \tau_1, \ldots, d_n : \tau_n$$

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

$$\frac{\Sigma \vdash d : \tau \quad \Sigma \vdash e : \tau}{\Sigma \vdash (\mathsf{eval}\ e\ d)\ \mathit{obj}} \qquad \frac{\Sigma \vdash d : \tau \quad \Sigma \vdash c :: \tau}{\Sigma \vdash (\mathsf{!cell}\ d\ c)\ \mathit{obj}}$$

$$\frac{\Sigma \vdash d_1 : \tau_1 \quad \Sigma \vdash d_2 : \tau_2 \quad \Sigma \vdash k \div \tau_1 \Rightarrow \tau_2}{\Sigma \vdash (\mathsf{cont}\ d_1\ k\ d_2)\ \mathit{obj}}$$

and the contents of cells with the following rules:

$$\frac{}{\Sigma \vdash \langle\,\rangle :: 1}\ (\text{C-}1) \qquad \frac{\Sigma \vdash d_1 : \tau_1 \quad \Sigma \vdash d_2 : \tau_2}{\Sigma \vdash \langle d_1, d_2 \rangle :: \tau_1 \otimes \tau_2}\ (\text{C-}\otimes)$$

$$\frac{\Sigma \vdash d : \tau_i \quad (i \in L)}{\Sigma \vdash i \cdot d :: \Sigma_{\ell \in L}(\ell : \tau_\ell)}\ (\text{C-}+) \qquad \frac{\Sigma \vdash d : [\rho\alpha.\,\tau/\alpha]\tau}{\Sigma \vdash \textbf{fold}\ d :: \rho\alpha.\,\tau}\ (\text{C-}\rho)$$

$$\frac{\Sigma \vdash e_1 : \tau_1 \quad \Sigma \vdash e_2 : \tau_2}{\Sigma \vdash \langle\!| e_1, e_2 |\!\rangle :: \tau_1 \mathbin{\&} \tau_2}\ (\text{C-}\&) \qquad \frac{\Sigma, x : \tau_1 \vdash e : \tau_2}{\Sigma \vdash \lambda x.\, e :: \tau_1 \to \tau_2}\ (\text{C-}\to)$$

The dynamics is given with the following rules:

$$\mathsf{!cell}\ d\ c, \mathsf{eval}\ d\ d' \quad\mapsto\quad \mathsf{!cell}\ d'\ c$$

$$\begin{array}{l}
\mathsf{eval}\ \langle\,\rangle\ d \quad\mapsto\quad \mathsf{!cell}\ d\ \langle\,\rangle \\
\mathsf{eval}\ (\textbf{case}\ e\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad\mapsto\quad \mathsf{eval}\ e\ d, \mathsf{cont}\ d\ (\textbf{case}\ \_\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad (d\ \text{fresh}) \\
\mathsf{!cell}\ d\ \langle\,\rangle, \mathsf{cont}\ d\ (\textbf{case}\ \_\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad\mapsto\quad \mathsf{eval}\ e'\ d'
\end{array}$$

$$\begin{array}{l}
\mathsf{eval}\ (\lambda x.\, e)\ d \quad\mapsto\quad \mathsf{!cell}\ d\ (\lambda x.\, e) \\
\mathsf{eval}\ (e_1\, e_2)\ d \quad\mapsto\quad \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1\ (\_\, e_2)\ d \qquad (d_1\ \text{fresh}) \\
\mathsf{!cell}\ d_1\ c_1, \mathsf{cont}\ d_1\ (\_\, e_2)\ d \quad\mapsto\quad \mathsf{eval}\ e_2\ d_2, \mathsf{cont}\ d_2\ (d_1\, \_)\ d \qquad (d_2\ \text{fresh}) \\
\mathsf{!cell}\ d_1\ (\lambda x.\, e_1'), \mathsf{!cell}\ d_2\ c_2, \mathsf{cont}\ d_2\ (d_1\, \_)\ d \quad\mapsto\quad \mathsf{eval}\ ([d_2/x]e_1')\ d
\end{array}$$

$$\begin{array}{l}
\mathsf{eval}\ \langle e_1, e_2 \rangle\ d \quad\mapsto\quad \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1\ \langle \_, e_2 \rangle\ d \qquad (d_1\ \text{fresh}) \\
\mathsf{!cell}\ d_1\ c_1, \mathsf{cont}\ d_1\ \langle \_, e_2 \rangle\ d \quad\mapsto\quad \mathsf{eval}\ e_2\ d_2, \mathsf{cont}\ d_2\ \langle d_1, \_ \rangle\ d \qquad (d_2\ \text{fresh}) \\
\mathsf{!cell}\ d_2\ c_2, \mathsf{cont}\ d_2\ \langle d_1, \_ \rangle\ d \quad\mapsto\quad \mathsf{!cell}\ d\ \langle d_1, d_2 \rangle \\
\mathsf{eval}\ (\textbf{case}\ e\ \{\langle x_1, x_2 \rangle \Rightarrow e'\})\ d' \quad\mapsto\quad \mathsf{eval}\ e\ d, \mathsf{cont}\ d\ (\textbf{case}\ \_\ \{\langle x_1, x_2 \rangle \Rightarrow e'\})\ d' \\
\hfill (d\ \text{fresh}) \\
\mathsf{!cell}\ d\ \langle d_1, d_2 \rangle, \mathsf{cont}\ d\ (\textbf{case}\ \_\ \{\langle x_1, x_2 \rangle \Rightarrow e'\})\ d' \quad\mapsto\quad \mathsf{eval}\ ([d_1/x_2, d_2/x_2]e')\ d'
\end{array}$$

$$\mathsf{eval}\ (\textbf{fix}\, x.\, e)\ d \quad\mapsto\quad \mathsf{eval}\ ([\textbf{fix}\, x.\, e/x]e)\ d$$

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

$$\begin{array}{l}
!\mathsf{cell}\ d_0\ (\mathbf{fold}\ d_1), \\
!\mathsf{cell}\ d_1\ (\mathsf{s} \cdot d_2), \\
!\mathsf{cell}\ d_2\ (\mathbf{fold}\ d_3), \\
!\mathsf{cell}\ d_3\ (\mathsf{z} \cdot d_4), \\
!\mathsf{cell}\ d_4\ \langle\,\rangle
\end{array}$$

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.

$$\text{Environments} \quad \eta \quad ::= \quad d_1/x_1, \ldots, d_n/x_n$$

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

$$\frac{}{\Sigma \vdash (\cdot) : (\cdot)} \qquad \frac{\Sigma \vdash \eta : \Gamma \quad \Sigma \vdash d : \tau}{\Sigma \vdash (\eta, d/x) : (\Gamma, x : \tau)}$$

Now evaluation depends on an environment

$$\frac{\Sigma \vdash d : \tau \quad \Sigma \vdash \eta : \Gamma \quad \Gamma \vdash e : \tau}{\Sigma \vdash \mathsf{eval}\ \eta\ e\ d\ \mathit{obj}}$$

Compared to the S machine in the previous lecture, expressions now no longer contain destinations, so the typing judgments for expressions reverts 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

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

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

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

For functions, we had

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

Now we have to pair up the environment with the λ-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 η.

$$\mathsf{eval}\ \eta\ (\lambda x.\, e)\ d \quad \mapsto \quad \mathsf{!cell}\ d\ \langle \eta, \lambda x.\, e \rangle$$

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.

$$\begin{array}{lcll} \mathsf{eval}\ \eta\ (e_1\, e_2)\ d & \mapsto & \mathsf{eval}\ \eta\ e_1\ d_1, \mathsf{cont}\ d_1\ (\_\ \langle \eta, e_2 \rangle)\ d & (d_1 \text{ fresh}) \\ \mathsf{!cell}\ d_1\ c_1, \mathsf{cont}\ d_1\ (\_\ \langle \eta, e_2 \rangle)\ d & \mapsto & \mathsf{eval}\ \eta\ e_2\ d_2, \mathsf{cont}\ d_2\ (d_1\, \_)\ d & (d_2 \text{ fresh}) \end{array}$$

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.

$$\mathsf{!cell}\ d_1\ \langle \eta, \lambda x.\, e_1' \rangle, \mathsf{!cell}\ d_2\ c_2, \mathsf{cont}\ d_2\ (d_1\, \_)\ d \quad \mapsto \quad \mathsf{eval}\ (\eta, d_2/x)\ e_1'\ d$$

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

$$\Sigma \vdash \eta : \Gamma \quad \text{and} \quad \Gamma \vdash \lambda x.\, e_1' : \tau$$

for some Γ and τ. 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.

$$\begin{array}{l}
\mathsf{eval}\ \eta\ \langle\!|e_\ell|\!\rangle_{\ell \in L}\ d \quad \mapsto \quad !\mathsf{cell}\ d\ \langle \eta, \langle\!|e_\ell|\!\rangle_{\ell \in L} \rangle \\
\mathsf{eval}\ \eta\ (e \cdot i)\ d \quad \mapsto \quad \mathsf{eval}\ \eta\ e\ d_1, \mathsf{cont}\ d_1\ (\_ \cdot i)\ d \quad (d_1 \text{ fresh}) \\
!\mathsf{cell}\ d_1\ \langle \eta, \langle\!|e_\ell|\!\rangle_{\ell \in L} \rangle, \mathsf{cont}\ d_1\ (\_ \cdot i)\ d \quad \mapsto \quad \mathsf{eval}\ \eta\ e_i\ d
\end{array}$$

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 variable in Γ if e was checked with $\Gamma \vdash e : \tau$. We can slightly improve on this, keeping only the variables of Γ 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.

5 Fixed Points

Fixed points are interesting. The rule of the S machine

$$\mathsf{eval}\ (\mathbf{fix}\, x.\, e)\ d \quad \mapsto \quad \mathsf{eval}\ ([\mathbf{fix}\, x.\, e/x]e)\ d$$

(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

$$\mathsf{eval}\ \eta\ (\mathbf{fix}\, x.\, e)\ d \quad \mapsto \quad \mathsf{eval}\ (\eta, (\mathbf{fix}\, x.\, e)/x)\ e\ d \quad ??$$

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.

$$\mathsf{eval}\ \eta\ (\mathbf{fix}\, x.\, e)\ d \quad \mapsto \quad \mathsf{eval}\ (\eta, \langle \eta, \mathbf{fix}\, x.\, e\rangle/x)\ e\ d \quad ??$$

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

$$\mathsf{eval}\ \eta\ (\mathbf{fix}\, x.\, e)\ d \quad \mapsto \quad \mathsf{eval}\ (\eta, d/x)\ e\ d$$

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 $\mathbf{fix}\, x.\, x$ in the empty environment $(\cdot)$.

$$\mathsf{eval}\ (\cdot)\ (\mathbf{fix}\, x.\, x)\ d_0 \quad \mapsto \quad \mathsf{eval}\ (d_0/x)\ x\ d_0$$

At this point the rule for variables

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

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,

```
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[1] 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

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

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:

$$\begin{array}{lcl} \mathit{nat} & \simeq & (\mathsf{z} : 1) + (\mathsf{s} : \mathit{nat}) \\ \mathit{zero} & = & \mathbf{fold}\ (\mathsf{z} \cdot \langle\,\rangle) \\ \\ \mathit{stream} & \simeq & \mathit{nat}\ \&\ \mathit{stream} \\ \\ \mathit{zeros} & : & \mathit{stream} \\ \mathit{zeros} & = & \mathbf{fix}\, x.\, \mathbf{fold}\ \langle\!|\mathit{zero}, x|\!\rangle \end{array}$$

The stream *zeros* corresponds to a potentially unbounded number of zeros, computed lazily. We see that **fold** $\langle\!|\mathit{zero}, x|\!\rangle$ 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 *zeros* as follows:

$$\begin{array}{ll} & \mathsf{eval}\ (\cdot)\ (\mathbf{fix}\ x.\, \mathbf{fold}\ \langle\!|\mathit{zero}, x|\!\rangle)\ d_0 \\ \mapsto & \mathsf{eval}\ (d_0/x)\ (\mathbf{fold}\ \langle\!|\mathit{zero}, x|\!\rangle)\ d_0 \\ \mapsto & \mathsf{eval}\ (d_0/x)\ \langle\!|\mathit{zero}, x|\!\rangle\ d_1, \mathsf{cont}\ d_1\ (\mathbf{fold}\ \_)\ d_0 \qquad (d_1 \text{ fresh}) \\ \mapsto & !\mathsf{cell}\ d_1\ \langle (d_0/x), \langle\!|\mathit{zero}, x|\!\rangle\rangle, \mathsf{cont}\ d_1\ (\mathbf{fold}\ \_)\ d_0 \\ \mapsto & !\mathsf{cell}\ d_1\ \langle (d_0/x), \langle\!|\mathit{zero}, x|\!\rangle\rangle, !\mathsf{cell}\ d_0\ (\mathbf{fold}\ d_1) \end{array}$$

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]but have not proven

environment stored with the closure. If we define

$$\begin{array}{lcl} \mathit{hd} & : & \mathit{stream} \to \mathit{stream} \\ \mathit{hd} & = & \lambda s.\, (\mathbf{unfold}\, s) \cdot l \\[1ex] \mathit{tl} & : & \mathit{stream} \to \mathit{stream} \\ \mathit{tl} & = & \lambda s.\, (\mathbf{unfold}\, s) \cdot r \end{array}$$

we should be able to check that *hd zeros* returns (a representation of) *zero*, while *tail zeros* returns (a representation of) *zeros*.

It is a good exercise to check that the *ascending* function below behaves as expected, where *ascending* $\ulcorner n \urcorner$ computes an ascending stream of natural numbers starting at $\ulcorner n \urcorner$.

$$\begin{array}{lcl} \mathit{succ} & : & \mathit{nat} \to \mathit{nat} \\ \mathit{succ} & = & \lambda n.\, \mathbf{fold}\ (\mathsf{s} \cdot n) \\[1ex] \mathit{ascending} & : & \mathit{nat} \to \mathit{stream} \\ \mathit{ascending} & = & \lambda n.\, \mathbf{fold}\ \langle\!| n, \mathit{ascending}\ (\mathit{succ}\ n) |\!\rangle \end{array}$$