Lecture Notes on Modeling Store

15-814: Types and Programming Languages
Frank Pfenning

Lecture 16
October 30, 2018

1 Introduction

The dynamics we have constructed so far treat both expressions and values as abstract terms, while in an actual machine architecture both expressions and values need to be stored in memory. In this lecture we introduce a store, arriving at the *S machine*. The idea is for the store to hold *values*. We leave *expressions* as terms with binders that we interpret directly. In the next lecture we'll look at expressions in (slightly) more detail.

We present the dynamics with store in the form of a *substructural operational semantics* [Pfe04, PS09, Sim12]. In this form of presentation the state is a collection of semantic objects which are rewritten following transition rules describing the semantics. We *can* think of them as inference rules, but unlike our original dynamics they would not have any premises.

2 Semantic Objects in the S Machine

At the heart of the S machine are *destinations* d (also called *addresses*) to hold values in the store. The only operation on them is to generate fresh ones—in a low-level implementation a system function such as `malloc` may be called. We assume the memory at a destination is not initialized until is written to.

The state of the S machine consists of the following objects:

eval e d**.** Evaluate expression e, storing the result in destination d. The destination d here is an address in the store which we assume has been

allocated with enough memory to hold the value of e.

!cell d c**.** Cell d has contents c. Because a value (such as a list) may be large, each cell contains only part of the value, and we use c to describe what (small) data may be stored in a cell. 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'.

As before, we will develop the semantics incrementally to see what cells might contain, and which continuations we might need.

2.1 Unit

Evaluating the unit element immediately just stores it in memory at the given destination. We write:

$$\mathsf{eval}\ \langle\,\rangle\ d \quad \mapsto \quad \mathsf{!cell}\ d\ \langle\,\rangle$$

The whole state of the S machine is a whole collection of objects, but we leave them implicit here because every rule is intended to apply to a subset of the objects, replacing those matching the left-hand side of the rule by the right-hand side. More explicit would be

$$S, \mathsf{eval}\ \langle\,\rangle\ d \quad \mapsto \quad S, \mathsf{!cell}\ d\ \langle\,\rangle$$

Second, if we have a case over a value of unit element we begin by evaluating the subject of the case, and remember in the continuation that we are waiting on this value.

$$\mathsf{eval}\ (\mathbf{case}\ e\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad \mapsto \quad \mathsf{eval}\ e\ d, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad (d \text{ fresh})$$

Let's read this. We create a fresh destination d to hold the value of e. The object cont d (eval _ $\{\langle\,\rangle \Rightarrow e'\})$ d' waits on the destination d before proceeding. Once the cell d holds a value (which must be $\langle\,\rangle$), the continuation must evaluate e' with destination d'.

$$\mathsf{!cell}\ d\ \langle\,\rangle, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle\,\rangle \Rightarrow e'\})\ d' \quad \mapsto \quad \mathsf{eval}\ e'\ d'$$

In this rule, the *persistent* !cell object on the left-hand side remains in the store, even though it is not explicitly mentioned on the right-hand side. The

continuation on the other hand is *ephemeral*, that is, it is consumed in the application of the rule and replaced by the eval object on the right-hand side.

As a simple example, consider the evaluation of $\mathbf{case}\ \langle\rangle\ \{\langle\rangle \Rightarrow \langle\rangle\}$ with some initial destination d_0, showing the whole state each time.

$$\begin{array}{lll} & \mathsf{eval}\ (\mathbf{case}\ \langle\rangle\ \{\langle\rangle \Rightarrow \langle\rangle\})\ d_0 & \\ \mapsto & \mathsf{eval}\ \langle\rangle\ d_1, \mathsf{cont}\ d_1\ (\mathbf{case}\ \_\ \{\langle\rangle \Rightarrow \langle\rangle\})\ d_0 & (d_1\ \text{fresh}) \\ \mapsto & \mathsf{!cell}\ d_1\ \langle\rangle, \mathsf{cont}\ d_1\ (\mathbf{case}\ \_\ \{\langle\rangle \Rightarrow \langle\rangle\})\ d_0 & \\ \mapsto & \mathsf{!cell}\ d_1\ \langle\rangle, \mathsf{eval}\ \langle\rangle\ d_0 & \\ \mapsto & \mathsf{!cell}\ d_1\ \langle\rangle, \mathsf{!cell}\ d_0\ \langle\rangle & \end{array}$$

We see that in the final state the initial destination d_0 holds the unit value $\langle\rangle$. In addition, there is some "junk" in the configuration, namely the cell d_1. This could safely be garbage-collected, although in this lecture we are not concerned with the definition and process of garbage collection.

In this example it may look like that the two objects that interact in the rules for continuations have to be next to each other, which is not the case in general. Even though we tend to write the state of the S machine in a sort-of canonical order with the store (cell objects) farthest to the left, then the eval object, if there is one, and then a sequence of continuations (cont objects) with the most recently created leftmost, this is not technically required.

2.2 Functions

Functions are relatively complex and thereby a good sample for how to design an abstract machine. λ-expressions are values, so the first rule is straightforward.

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

Some sound objection might be raised to this rule, since allocated memory should have fixed size but the a λ-expression may not. In this lecture, we ask you to suspend this objection; in the next lecture we will present one way to make this aspect of the S machine more realistic.

As usual in a call-by-value language, an application is evaluated by first evaluating the function, then the argument, and then perform a β-reduction. We will reuse the continuations previously created for this purpose in the K machine.

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

The first two rules should be expected, since they are a a straighforward rewrite of the K machine's transition rules. Note that in the second rule we check that the cell d_1 holds a value (c_1), but we actually do not use the contents. Nevertheless, this check is necessary to ensure that operation of the S machine is deterministic: there always is a unique next step, assuming we start in state

$$\mathsf{eval}\ e\ d_0$$

and stop when there are no eval or cont cells left.

The interesting aspect of the last rule is that it we substitute not a *value* (as we have done in the dynamics so far, including the K machine), but the *address* d_2 of a value. This necessitates a further rule, namely how to evaluate a destination! The destination amounts to a reference to the store, so we have to copy the contents at one address to another. Since we imagine the size of storage locations to be fixed and small, this is a reasonable operation.

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

There is an alternative line of thought where we store in the cell d' not a copy of the value c, but a reference to the value c. Then, of course, we would have to follow chains of references and rules that need to access the contents of cells would become more complicated.

Because fixed points are usually used for functions, the simple and straightforward rule just unrolls the recursion.

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

In the next lecture we will look at a different semantics for fixed points since we want to avoid substitution into expressions.

3 A Simple Example

As a simple example, we consider the evaluation of $((\lambda x.\, \lambda y.\, x)\ 7)\ 5$ with an initial destination d_0. Here, 7 and 5 stand in for values that can be directly

stored in memory, to simplify the example.

$$
\begin{array}{lll}
 & \mathsf{eval}\ (((\lambda x.\,\lambda y.\,x)\,7)\,5)\ d_0 & \\
\mapsto & \mathsf{eval}\ ((\lambda x.\,\lambda y.\,x)\,7)\ d_1, \mathsf{cont}\ d_1\ (\_\,5)\ d_0 & (d_1 \text{ fresh}) \\
\mapsto & \mathsf{eval}\ (\lambda x.\,\lambda y.\,x)\ d_2, \mathsf{cont}\ d_2\ (\_\,7)\ d_1, \mathsf{cont}\ d_1\ (\_\,5)\ d_0 & (d_2 \text{ fresh}) \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\,\lambda y.\,x), \mathsf{cont}\ d_2\ (\_\,7)\ d_1, \mathsf{cont}\ d_1\ (\_\,5)\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\,\lambda y.\,x), \mathsf{eval}\ 7\ d_3, \mathsf{cont}\ d_3\ (d_2\,\_)\ d_1, \mathsf{cont}\ d_1\ (\_\,5)\ d_0 & (d_3 \text{ fresh}) \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\,\lambda y.\,x), \mathsf{!cell}\ d_3\ 7, \mathsf{cont}\ d_3\ (d_2\,\_)\ d_1, \mathsf{cont}\ d_1\ (\_\,5)\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\,\lambda y.\,x), \mathsf{!cell}\ d_3\ 7, \mathsf{eval}\ (\lambda y.\,d_3)\ d_1, \mathsf{cont}\ d_1\ (\_\,5)\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\,\lambda y.\,x), \mathsf{!cell}\ d_3\ 7, \mathsf{!cell}\ d_1\ (\lambda y.\,d_3), \mathsf{cont}\ d_1\ (\_\,5)\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\,\lambda y.\,x), \mathsf{!cell}\ d_3\ 7, \mathsf{!cell}\ d_1\ (\lambda y.\,d_3), \mathsf{eval}\ 5\ d_4, \mathsf{cont}\ d_4\ (d_1\,\_)\ d_0 & (d_4 \text{ fresh}) \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\,\lambda y.\,x), \mathsf{!cell}\ d_3\ 7, \mathsf{!cell}\ d_1\ (\lambda y.\,d_3), \mathsf{!cell}\ d_4\ 5, \mathsf{cont}\ d_4\ (d_1\,\_)\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\,\lambda y.\,x), \mathsf{!cell}\ d_3\ 7, \mathsf{!cell}\ d_1\ (\lambda y.\,d_3), \mathsf{!cell}\ d_4\ 5, \mathsf{eval}\ d_3\ d_0 & \\
\mapsto & \mathsf{!cell}\ d_2\ (\lambda x.\,\lambda y.\,x), \mathsf{!cell}\ d_3\ 7, \mathsf{!cell}\ d_1\ (\lambda y.\,d_3), \mathsf{!cell}\ d_4\ 5, \mathsf{!cell}\ d_0\ 7 &
\end{array}
$$

4 Eager Pairs

Eager pairs are somewhat similar to functions, but we construct a pair in memory as soon as the two components are evaluated. An interesting aspect of the S machine is that we form a new cell containing just a pair of destinations, indicating where the components of the pair are stored.

$$
\begin{array}{lll}
\mathsf{eval}\ \langle e_1, e_2\rangle\ d & \mapsto & \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1\ \langle \_, e_2\rangle\ d \qquad (d_1 \text{ fresh}) \\
\mathsf{!cell}\ d_1\ c_1, \mathsf{cont}\ d_1\ \langle \_, e_2\rangle\ d & \mapsto & \mathsf{eval}\ e_2\ d_2, \mathsf{cont}\ d_2\ \langle d_1, \_\rangle\ d \qquad (d_2 \text{ fresh}) \\
\mathsf{!cell}\ d_2\ c_2, \mathsf{cont}\ d_2\ \langle d_1, \_\rangle\ d & \mapsto & \mathsf{!cell}\ d\ \langle d_1, d_2\rangle
\end{array}
$$

In lecture, it was pointed out is might make sense to also check that cell d_1 holds a value, with a another persistent $\mathsf{!cell}\ d_1\ c_1$ on the left-hand side. This is redundant because in a sequential semantics the continuation $\langle d_1, \_\rangle$ only makes sense if d_1 already holds a value. The difference between the rules is therefore just a matter of style.

In the rule for the destructor of eager pairs we perform again a substitution of destinations in an expression, as already seen for functions.

$$
\begin{array}{lll}
\mathsf{eval}\ (\mathbf{case}\ e\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' & \mapsto & \mathsf{eval}\ e\ d, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' \\
 & & (d \text{ fresh}) \\
\mathsf{!cell}\ d\ \langle d_1, d_2\rangle, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' & \mapsto & \mathsf{eval}\ ([d_1/x_2, d_2/x_2]e')\ d'
\end{array}
$$

5 Typing the Store

First, a summary of the three types we have considered so far.

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

From these examples we can readily extrapolate the rest of the S machine. Continuations haven't really changed from the K machine except we only use a small piece at a time and not whole stacks. We just show the possible cell contents, organized by type, thereby describing the possible shapes of memory.

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

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

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

In a store typing, all destinations must be distinct. Notice the difference to the usual typing context Γ that types *variables*, while Σ assign types to destinations. At runtime, we only execute expression without free variables, but several rules (for example, for function calls) will substitute a destination for a variable. Therefore, we type expressions with $\Sigma, \Gamma \vdash e : \tau$ with the additional rule

$$\frac{d : \tau \in \Sigma}{\Sigma, \Gamma \vdash d : \tau}\ \textsf{Dest}$$

while in all other rules we just add Σ and propagate it from the conclusion to all premises.

Next we move on to typing objects. For uniformity we write $\Sigma \vdash d : \tau$ if $d : \tau \in \Sigma$. We type each object P with the judgment $\Sigma \vdash P\ \mathit{obj}$. From this, the typings are rather straightforward.

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

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

A state is well-typed with respect to store typing Σ if each object in it is a valid object. This form of typing is inadequate in several respects and, in particular, it does not guarantee progress. An initial state has the form $\mathsf{eval}\ e\ d_0$ for a destination d_0 and a final state consists solely of memory cells $!\mathsf{cell}\ d_i\ c_i$ (which should include d_0). However, a state such as $\mathit{cont}\ d_2\ \langle d_1, \_ \rangle\ d_0$ is a perfectly valid state for the store typing

$$d_0 : \tau_1 \otimes \tau_2, d_1 : \tau_1, d_2 : \tau_2$$

for any types τ_1, τ_2, but cannot make a transition. We may address the question how to obtain a more precise typing for states of the machine with store in a later lecture.

We still owe the rules for the contents of the store. They do not present any difficulty. In the rules for the eager constructs ((C-1), (C-$\otimes$), (C-$+$), (C-ρ)) we refer only directly to the types of other destinations, while for the lazy ones ((C-$\&$), (C-$\to$)) we have to type the embedded expressions.

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

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

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

6 Concurrency/Parallelism

Both in the K machine and the S machine we ensured that evaluation was *sequential*: there was always a unique next step to take. Our dynamics

formalism is general enough to support parallel or concurrent evaluation. Consider, for example, an eager pair. We can evaluate the components of a pair independently, each with a new separate destination. Moreover, we can immediately fill the destination with a pair so that further computation can proceed before either component finishes!

$$\mathsf{eval}\ \langle e_1, e_2\rangle\ d \quad \mapsto \quad \mathsf{!cell}\ d\ \langle d_1, d_2\rangle, \mathsf{eval}\ e_1\ d_1, \mathsf{eval}\ e_2\ d_2$$

Recall the rules for the pair destructor.

$$\mathsf{eval}\ (\mathbf{case}\ e\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' \quad \mapsto \quad \mathsf{eval}\ e\ d, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' \qquad (d \text{ fresh})$$

$$\mathsf{!cell}\ d\ \langle d_1, d_2\rangle, \mathsf{cont}\ d\ (\mathbf{case}\ \_\ \{\langle x_1, x_2\rangle \Rightarrow e'\})\ d' \quad \mapsto \quad \mathsf{eval}\ ([d_1/x_2, d_2/x_2]e')\ d'$$

We see that the body of the **case** construct can evaluate as soon as the cell d has been filled with a pair of destinations, but before either of these destinations has been filled. This enables a lot of fine-grained parallelism, so much so, that if we try to do everything in parallel in many programs there would simply be too many threads of control to execute efficiently.

We also observe that the distinction between *eager* (or *strict*) and *lazy* is difficult to apply to this situation. Both components of the pair are evaluated, but we don't wait for them to finish. If only one component is needed in the body of the **case**, the other might not terminate and yet we may have filled the initial destination d_0.

We may return to a closer examination of a language supporting parallelism or concurrency in a future lecture.

References

[Pfe04] Frank Pfenning. Substructural operational semantics and linear destination-passing style. In W.-N. Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04)*, page 196, Taipei, Taiwan, November 2004. Springer-Verlag LNCS 3302. Abstract of invited talk.

[PS09] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 24th Annual Symposium on Logic in Computer Science (LICS 2009)*, pages 101–110, Los Angeles, California, August 2009. IEEE Computer Society Press.

[Sim12] Robert J. Simmons. *Substructural Logical Specifications*. PhD thesis, Carnegie Mellon University, November 2012. Available as Technical Report CMU-CS-12-142.