lec16: Mutable state

Joshua Dunfield

March 28, 2018

Covers Tuesday, 27 March

1 Introduction

Our language so far has been *purely functional*: it has no features that allow

- output (other than a final result v), such as writing to a console, output file, or network;
- input (other than input given as part of the source program), such as reading from a file, from a user device such as a keyboard, or from the network;
- *mutable state*, in which (some) program variables can be updated (re-assigned).

Except for *extremely* purely functional languages that do not provide these features in any form[1], we would like our semantics to describe these language features. Even if we only use purely functional languages, real machine architectures are not purely functional, and we would like semantics that can describe the connection between a purely functional source language and the machine code produced by a compiler.

Our existing judgment forms $e \mapsto e'$ and $e \Downarrow v$ are not suitable to describe these features. (It is *possible*, but not easy; it requires fairly drastic changes to the language, and the resulting semantics is especially difficult to understand.)

In these notes, we extend our operational semantics to describe one non-purely-functional language feature: *mutable state*.

2 State

"State is the coldest of all cold monsters." —Friedrich Nietzsche

The form of mutable state we'll consider is more or less the same as that found in Standard ML. Specifically:

- We will continue to not support writing to a variable as such: in $(\texttt{Lam}\ x\ e_{\text{body}})$, whatever was passed for the argument x (in a $\texttt{Call}$ expression) is the value of x throughout e_{body}; there is no general "assignment statement".
- However, our language will support *locations* in a *store*, which *can* be updated.

The resulting language is, roughly, "functional by default, imperative by request". If you want to maintain a counter that is updated, without passing the counter around as an extra argument or extra result (in our language, this could be done using a $\texttt{Pair}$), you can do that—but you must "request" that by explicitly allocating a location.

[1] Haskell, for example, provides input and output through monads, which "hide" the non-functional behaviour behind an interface that is purely functional.

2.1 Features

We will add four new expression forms: three that allocate, read, and write references; and one to describe locations themselves.

$$
\begin{array}{llcll}
\text{Locations} & L & ::= & L_1 \mid L_2 \mid \ldots & \\
\text{Expressions} & e & ::= & \ldots & \\
 & & & \mid (\mathtt{Ref}\ e) & \text{allocate a reference (or location) initialized to } e \\
 & & & \mid (\mathtt{Get}\ e) & \text{read the contents of a location} \\
 & & & \mid (\mathtt{Set}\ e\ e) & \text{update the contents of a location} \\
 & & & \mid L & \text{location} \\
\text{Values} & v & ::= & \ldots & \\
 & & & \mid L & \\
\text{Stores} & \Sigma & ::= & \mathsf{emp} & \text{empty store} \\
 & & & \mid \Sigma, L{=}v & \text{store } \Sigma\text{, with additional location } L \text{ containing } v
\end{array}
$$

We will change the judgment form $e \mapsto e'$ to a new form:

$$\Sigma; e \mapsto \Sigma'; e'$$

This judgment is read: *starting in store Σ, expression e steps to store Σ' and expression e'.*

2.2 Stepping rule

Instead of step-context deriving $\mathcal{C}[e] \mapsto \mathcal{C}[e']$, we will have a rule that derives $\Sigma; \mathcal{C}[e] \mapsto \Sigma'; \mathcal{C}[e']$.

$\boxed{\Sigma; e \mapsto \Sigma'; e'}$ starting in store Σ, expression e steps to store Σ' and expression e'

$$\frac{\Sigma; e \mapsto_R \Sigma'; e'}{\Sigma; \mathcal{C}[e] \mapsto \Sigma'; \mathcal{C}[e']}\ \text{Step-context}$$

(I capitalized the S in Step-context so that we would not have two rules with exactly the same name, one for the original $e \mapsto e'$ judgment form, one for the new form $\Sigma; e \mapsto \Sigma'; e'$.)

2.3 Reduction rules

Our existing reduction rules (red-beta, etc.) should not affect the store, so they can all be copied over, only adding Σ on both sides of $\mapsto_R$.

Our three new reduction rules, one for each of our three new expression forms, all make use of Σ.

$\boxed{\Sigma; e \mapsto_R \Sigma'; e'}$ starting in store Σ, expression e reduces to store Σ' and expression e'

$$\frac{}{\Sigma; (\mathtt{Call}\ (\mathtt{Lam}\ x\ e_{\text{body}})\ v) \mapsto_R \Sigma; [v/x]e_{\text{body}}}\ \text{Red-call}$$

$$\frac{L \notin \mathsf{locs}(\Sigma)}{\Sigma; (\mathtt{Ref}\ v) \mapsto_R \Sigma, L{=}v; L}\ \text{Red-ref}$$

$$\frac{(L{=}v) \in \Sigma}{\Sigma; (\mathtt{Get}\ L) \mapsto_R \Sigma; v}\ \text{Red-get}$$

$$\frac{}{\Sigma_1, L{=}v_{\text{old}}, \Sigma_2; (\mathtt{Set}\ L\ v_{\text{new}}) \mapsto_R \Sigma_1, L{=}v_{\text{new}}, \Sigma_2; ()}\ \text{Red-set}$$

Rule Red-ref uses an auxiliary definition, of a (mathematical) function locs:

$$\begin{aligned} \mathsf{locs}(\mathsf{emp}) &= \{\} \\ \mathsf{locs}(\Sigma, L{=}v) &= \mathsf{locs}(\Sigma) \cup \{L\} \end{aligned}$$

For example, the locations in the store $\mathsf{emp}, L_1{=}1, L_2{=}(\mathtt{Lam}\ x\ x), L_3{=}()$ are given by

$$\begin{aligned} \mathsf{locs}(\mathsf{emp}, L_1{=}1, L_2{=}(\mathtt{Lam}\ x\ x), L_3{=}()) &= \mathsf{locs}(\mathsf{emp}, L_1{=}1, L_2{=}(\mathtt{Lam}\ x\ x)) \cup \{L_3\} \\ &= \mathsf{locs}(\mathsf{emp}, L_1{=}1) \cup \{L_2\} \cup \{L_3\} \\ &= \mathsf{locs}(\mathsf{emp}) \cup \{L_1\} \cup \{L_2\} \cup \{L_3\} \\ &= \{\} \cup \{L_1\} \cup \{L_2\} \cup \{L_3\} = \{L_1, L_2, L_3\} \end{aligned}$$

The choice of $()$ in Red-set is quite arbitrary (beyond being the choice made by Standard ML); other vaguely reasonable options include returning v_{new} (similar to C's semantics for "chained" assignment), or returning L.

In addition to the meaningful, but arbitrary, choice of what expression to return, there are many ways to express "change $L{=}v_{\text{old}}$ to $L{=}v_{\text{new}}$", some of which we discussed in class. The way I have chosen here is different from all of those: Red-set "decomposes" the store into a left-hand part Σ_1 and a right-hand part Σ_2, with $L{=}v_{\text{old}}$ in the middle.

■ **Exercise 1.** Rewrite Red-get to use the same technique as the Red-set above.

2.4 Evaluation contexts

In class, as we worked out the reduction rules, we added to the evaluation contexts.

For each new expression form, we need a production for each of that form's subexpressions, allowing us to reduce non-values to values. Thus, $\mathtt{Ref}$ has one production for its one subexpression, etc.

$$\begin{array}{lll}
\text{Evaluation contexts} & ::= & \ldots \\
& & \mid (\mathrm{Ref}\ \mathcal{C}) \\
& & \mid (\mathrm{Get}\ \mathcal{C}) \\
& & \mid (\mathrm{Set}\ \mathcal{C}\ e) \\
& & \mid (\mathrm{Set}\ v\ \mathcal{C})
\end{array}$$