1 Big-Step Environment Semantics

In our previous coverage of formal semantics, we looked at a small-step operational semantics for Featherweight Java. It is an operational semantics because it describes, through rewriting rules, how a program operates during execution—as opposed to, say, the mathematical meaning of the program (which would be denotational semantics, something you can learn about in another course). It is small step because execution is broken into individual steps which make incremental progress, similar to one or a few processor instructions. Also, we handled variables using substitution, replacing all occurrences of a parameter variable in a method body with the value passed in for that parameter. This style of semantics was chosen because it is relatively simple, complete, and concrete, all valuable goals both for education and language modeling during design. The judgment form we used is $e \rightarrow e'$ indicating that the expression $e$ takes a small step and evaluates to the new expression $e'$.

When we consider implementing an interpreter, however, this style of semantics does not fully match the way the interpreter works. Interpreters are indeed operational, but they are not “small step” in terms of producing an entirely new program state after every micro operation. Instead, the interpreter is often written to be threaded through the run-time execution of the program, e.g. reprenting the call stack of the program as the call stack of the interpreter itself. The interpreter is oriented towards computing the end result of the program in one big operation. There is a different kind of semantics that represents this choice better: a big step operational semantics. Our judgments will be of the form $e \Downarrow v$ indicating that the expression $e$ evaluates to the value $v$.

Furthermore, interpreters do not typically implement variables by substituting them with values; that would potentially be a quite slow operation. Instead, the interpreter keeps track of an environment mapping variables to values. When a variable is accessed during interpretation, we look up the value of that variable in the environment. We can model this with an environment semantics. The judgment form therefore adds an environment $E$. We can add environments to a small-step semantics yielding a judgment like $E \vdash e \rightarrow e'$, read “In the context of environment $E$, expression $e$ takes a small step resulting in expression $e'$.” But in these notes, we will be adding environments to a big step semantics, with the judgment form $E \vdash e \Downarrow v$, read “In the context of environment $E$, expression $e$ evaluates to the value $v$.”

Let’s model some key constructs from the functional programming assignment. We’ll model the part of the grammar that corresponds to the lambda calculus: function definitions, variables, and function calls. We’ll also include integers and addition. In addition to the grammar for expressions $e$, we’ll define a subset of the grammar $v$ representing values (which right now means just function definitions and integers):
Let’s start by writing rules for numbers, variables, and addition:

\[
E \vdash n \Downarrow n \quad \text{big-num-val}
\]

\[
E, x \mapsto v \vdash x \Downarrow v \quad \text{big-var}
\]

\[
E \vdash e_1 \Downarrow n_1 \quad E \vdash e_2 \Downarrow n_2 \implies E \vdash e_1 + e_2 \Downarrow n_1 + n_2 \quad \text{big-add}
\]

In a big-step semantics, we have to say how all expressions reduce to a value—even expressions that already are values. In this case, numbers \(n\) are values, so they reduce to themselves. The second rule says that in order to evaluate an addition expression, you evaluate the first and second subexpressions down to numbers, then you add the numbers. I’ve used \(\hat{+}\) to denote mathematical addition, distinguishing it from the program-text symbol \(+\).

Here’s what a complete derivation looks like in this setting. Let’s assume that the local environment \(E\) specifies that \(x\) maps to 2:

\[
E \vdash 1 \Downarrow 1 \quad x \mapsto 2 \vdash x \Downarrow 2 \quad \text{big-num}
\]

\[
x \mapsto 2 \vdash x + 3 \Downarrow 5 \quad \text{big-add}
\]

\[
x \mapsto 2 \vdash 1 + (x + 3) \Downarrow 6 \quad \text{big-add}
\]

Now let’s add naive (spoiler: not correct!) big-step operational rules for function definitions and for function call evaluation:

\[
E \vdash \text{fun } x \rightarrow e \Downarrow \text{fun } x \rightarrow e \quad \text{big-fun-val} \quad \text{(WRONG!)}
\]

\[
E \vdash e_1 \Downarrow \text{fun } x \rightarrow e \quad E \vdash e_2 \Downarrow v_2 \quad E, x \mapsto e \vdash v_2 \Downarrow v \quad \text{big-app-dyn} \quad \text{(WRONG!)}
\]

The first rule just says that functions are already values. The second rule says that in order to evaluate an application \(e_1 \ e_2\) to a value in environment \(E\), we first use \(E\) to evaluate \(e_1\) to a value, which must be a function of the form \(\text{fun } x \rightarrow e\). We then use \(E\) to evaluate the argument \(e_2\) to a value \(v_2\). Finally, we extend \(E\) with a mapping from \(x\) to the \(v_2\) and evaluate the function body in that context.

**Exercise 1.** What’s wrong with these rules?

Unfortunately, we’ve forgotten to model closures! We made the same mistake as most Lisp variants prior to Common Lisp: dynamic scope. We wouldn’t need closures in a substitution-based semantics, but we do need them for an environment semantics. To fix this, we need to update our grammar. Functions are no longer technically values; instead, we introduce a new form, closures, that store the function’s environment:
\[
e ::= v \mid \text{fun } x \to e \mid x \mid e \mid e + e \mid \ldots \]

\[
v ::= (\text{closure } x \to e, E) \mid n \mid \ldots
\]

Now that we have a way to represent closures, we can write correct rules for function definition and function call:

\[
\frac{E \vdash \text{fun } x \to e}{E \vdash \text{Big-fun}}
\]

\[
\frac{E \vdash e_1 \Downarrow (\text{closure } x \to e, E)}{E \vdash e_2 \Downarrow v_2 \quad E_c, x_c \mapsto v_2 \vdash e_c \Downarrow v}{E \vdash e_1 \Downarrow e_2 \Downarrow v}
\]

The first rule says that function definitions are not values, but rather evaluate to a closure that includes the variable, the function body, and the environment as it was when the function was defined.

The second rule says that to evaluate a function call, we first evaluate the function to a closure and the argument to a value \(v_2\). We then extend the environment \(E_c\) from the closure with a mapping from \(x_c\) to \(v_2\), and use that to evaluate the body of the function to a value \(v\), which is the overall result of evaluating the function call.

Here’s a derivation that shows how closures work. The derivation is a bit too big for the width of these notes, so I’ll split it into three parts: the first is the primary derivation, followed by two subderivations that can be plugged in for sub-parts of the tree. I’ve also abbreviated the closures \(C_1\) and \(C_2\), defined at the bottom.

\[
\text{Exercise 2. Write two evaluation rules for an if } e \text{ then } e_1 \text{ else } e_2 \text{ construct.}
\]

\[
\text{Exercise 3. Write a evaluation rule for a let } x = e_1 \text{ in } e_2 \text{ construct. Before you write the rule, think about whether evaluating let requires a closure.}
\]
2 Modeling let rec

Let’s look at how to model let rec. As discussed separately, we need a formal model where we can mutate an environment. Thus, our mathematical model will represent environments $E$ as pointers within a store (i.e. heap) $S$.

Let $E$ be a set of environment locations. A distinguished location $\text{nil} \in E$. Let $S$ be a store mapping environment locations to a cell in a linked list. Each cell contains a variable, a value, and a next location. The grammar is:

$$S ::= \bullet | S, E \mapsto (x, v, E)$$

Stores will be passed through the evaluation judgment $S; E \vdash e \downarrow v \downarrow S'$ which is read “In the context of store $S$ and environment pointer $E$, the expression $e$ evaluates to a value $v$, producing a new store $S'$.

The lookup function iterates through a linked list starting with $E$ in a store $S$, looking for $x$ and returning the corresponding value:

$$S[E] = (x, v, E')$$
$$\text{lookup}(E, S, x) = v$$

$$S[E] = (x', v', E') \quad x \neq x'$$
$$\text{lookup}(E', S, x) = v$$

The update function returns a store that is the same as the input, except that the value at location $E$ is replaced with $v$. In the rule, we write $S[E \mapsto (x, v, E')]$ to mean a store that is the same as $S$, except that $E$ points to a cell consisting of $(x, v, E')$.

Now we can define some rules:

$$S; E \vdash x \downarrow v \mapsto \text{lookup}(E, S, x) \downarrow S$$ big-var

$$S; E \vdash e_1 \downarrow v_1 \downarrow S' \quad E' \text{ fresh} \quad S', E' \mapsto (x, v_1, E); E' \vdash e_2 \downarrow v_2 \downarrow S''$$
$$S; E \vdash \text{let } x = e_1 \text{ in } e_2 \downarrow v_2 \downarrow S''$$ big-let

$$S, E' \mapsto (x, \bullet, E); E' \vdash e_1 \downarrow v_1 \downarrow S' \quad E' \text{ fresh} \quad \text{update}(E', S', v_1); E' \vdash e_2 \downarrow v_2 \downarrow S''$$
$$S; E \vdash \text{let rec } x = e_1 \text{ in } e_2 \downarrow v_2 \downarrow S''$$ big-letrec

The big-var rule illustrates how variables work in the setting of environments represented as linked lists in a store. Similarly, we provide a big-let rule to illustrate how an “ordinary” let works. After evaluating the first expression $e_1$, we allocate a fresh environment location $E'$ in the store\footnote{technically fresh means that $E'$ is not in the domain of the store $S$} and extend the store to specify that $E'$ refers to a cell containing $(x, v_1, E)$—essentially adding a cell to the beginning of the linked list. We use the new store and the extended
environment $E'$ to evaluate the body of the \texttt{let}. This may result in more store updates (e.g., perhaps a new environment was allocated by a nested lambda, and that environment may have been captured) so we return the new store $S''$ from the rule after the \texttt{-i}.

Now we can see how the rule for \texttt{let rec} works. Here, we allocate a new cell before evaluating the first expression $e_1$; since we don’t yet have a value, we use $\bullet$ as a placeholder. With this new store and extended environment, we evaluate $e_1$ to a value $v_1$; this may involve capturing the environment $E'$ or some extension of it. We then update the resulting store $S'$ (which may, of course, have allocated other environment pointers) so that the entry for environment $E'$ now maps $x$ to the value $v_1$. If $E'$ was captured by some closure, that closure will see the modification, because we pass the modified store through to all evaluation later in the program.

3 Modeling Pattern Matching

Let’s assume a grammar for patterns and for values similar to that in Homework 6:

\[
\begin{align*}
  v & ::= n \mid b \mid (\text{closure } x \rightarrow e, E) \mid (\overline{p}) \mid \text{C } v \\
  p & ::= n \mid b \mid x \mid (p) \mid \text{C } p
\end{align*}
\]

The semantics of pattern matching is centered around (A) discovering whether the match succeeds or not, and (B) if it succeeds, computing the additions to an environment that come out of the pattern match. We define a function $\text{match}(v, p, E)$ that returns either $\text{fail}$ or a possibly extended environment $E'$. The rules are as follows:

\[
\begin{align*}
  v \neq v' & \quad \text{match-val-fail} \\
  \text{match}(v, v', E) = \text{fail} \\
  \text{match}(v, v, E) = E & \quad \text{match-val-succeed} \\
  \text{match}(v, \_ , E) = E & \quad \text{match-wildcard} \\
  \text{match}(v, x, E) = E[x \mapsto v] & \quad \text{match-var} \\
  v \neq (\overline{p}) \quad \text{or} \quad |\overline{p}| \neq |\overline{p}| & \quad \text{match-tuple-fail} \\
  \text{match}(v, (\overline{p}), E) = \text{fail} \\
  \forall i \in 1 \ldots n: \text{match}(v_i, p_i, E_i) = E_{i+1} & \quad \text{match-tuple-succeed} \\
  \text{match}(v, \text{C } p, E) = \text{fail} & \quad \text{match-constructor-fail} \\
  \text{match}(v, p, E) = E' & \quad \text{match-constructor-succeed} \
\end{align*}
\]