Lecture 3: Environments

In the last lecture, we used *substitution* to evaluate let-expressions and function application. Substitution is a familiar idea from algebra, which is one of the reasons why we used it. However, it is not a very efficient implementation strategy. Consider the following expression:

$$\mathsf{let}\ x_1 = 1\ \mathsf{in}\ \mathsf{let}\ x_2 = 2\ \mathsf{in}\ \cdots \mathsf{let}\ x_n = n\ \mathsf{in}\ x_1 + \cdots + x_n$$

To evaluate this expression, we would first scan the entire program to substitute x_1 with 1, then scan again to substitute x_2 with 2, and so on, so evaluation by substitution would take $O(n^2)$ steps.

We now introduce an auxiliary data structure called an *environment* that stores the values bound to identifiers. Intuitively, instead of substituting, we lookup the value of identifiers in the environment.

1 Environment-based Semantics

Figure 4.1 presents an environment-based semantics for the language. Before we discuss the new semantics, we'll explain the more compact notation that the figure uses:

- Instead of spelling out every type of constant and binary operator in the definition of expressions, e, the figure uses the metavariable c to denote an arbitrary constant and op_2 to denote an arbitrary binary operator. In addition, instead of spelling out the semantics of every binary operator, we assume the presence of a partial function, δ, that consumes an operator and two constants and produces a result, if the result is defined. **Note**: we write $X \rightharpoonup Y$ to denote a partial function, whereas $X \rightarrow Y$ is a total function.

  The main reason for this change is that we can easily introduce new operators and new kinds of constants (e.g., strings) without changing the definition of e and $\Downarrow$.

- The syntax no longer has let-expressions. However, note that $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ is equivalent to $\lambda x.e_2\ e_1$. Therefore, we will continue using let-expressions in examples, but you should think of them as functions that have been immediately applied.

  As a brief exercise, you should try to prove that $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v$ if and only if $\lambda x.e_2\ e_1 \Downarrow v$.

The primary change in fig. 4.1 is that the semantics is defined by a ternary relation, where ρ is the environment. An environment is a partial function that maps identifiers to values. Therefore, instead of using substitution, we have a new rule for identifiers (Id) that "looks up"

$\boxed{e}$

Binary Operators

$$
\begin{array}{lcll}
op_2 & ::= & + \mid > \mid \cdots & \\
\delta & \in & op_2 \times c \times c \rightharpoonup c &
\end{array}
$$

Constants

$$
\begin{array}{lcll}
c & ::= & \mathsf{true} & \text{True} \\
 & \mid & \mathsf{false} & \text{False} \\
 & \mid & n & \text{Integers}
\end{array}
$$

Expressions

$$
\begin{array}{lcll}
e & ::= & c & \text{Constants} \\
 & \mid & x & \text{Identifiers} \\
 & \mid & op_2(e_1, e_2) & \text{Bin. Ops.} \\
 & \mid & \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \text{Conditionals} \\
 & \mid & e_1\ e_2 & \text{Applications} \\
 & \mid & \lambda x.e & \text{Functions}
\end{array}
$$

Environments

$$\rho \in x \rightharpoonup v$$

Values

$$
\begin{array}{lcll}
v & ::= & c & \text{Constants} \\
 & \mid & \langle \rho, x, e \rangle & \text{Closures}
\end{array}
$$

(a) Syntax.

$\boxed{\rho, e \Downarrow v}$

$$\textsc{Const}\ \rho, c \Downarrow c \qquad \textsc{BinOp}\ \frac{\rho, e_1 \Downarrow c_1 \qquad \rho, e_2 \Downarrow c_2}{op_2(e_1, e_2) \Downarrow \delta(op_2, c_1, c_2)}$$

$$\textsc{IfTrue}\ \frac{\rho, e_1 \Downarrow \mathsf{true} \qquad \rho, e_2 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v}$$

$$\textsc{IfFalse}\ \frac{\rho, e_1 \Downarrow \mathsf{false} \qquad \rho, e_3 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v}$$

$$\textsc{Fun}\ \rho, \lambda x.e \Downarrow \langle \rho, x, e \rangle \qquad \textsc{Id}\ \rho, x \Downarrow \rho(x)$$

$$\textsc{App}\ \frac{\rho, e_1 \Downarrow \langle \rho', x, e' \rangle \qquad \rho, e_2 \Downarrow v' \qquad \rho'[x \mapsto v'], e' \Downarrow v}{\rho, e_1\ e_2 \Downarrow v}$$

(b) Semantics.

Figure 4.1: Environment-based semantics. Contrast to fig. 3.2.

identifiers in the environment. (Recall that we didn't have a rule for identifiers earlier, since they were eliminated by substitution.) The rule for constants (Const) is updated to ignore ρ and the rules for operators and conditionals simply propagate ρ to their subexpressions.

The rules for functions (Fun) and applications (App) are significantly different. To understand why, consider the following program:

$$\mathsf{let}\ f = (\mathsf{let}\ x = 20\ \mathsf{in}\ \lambda y.x + y)\ \mathsf{in}\ \mathsf{let}\ x = 10\ \mathsf{in}\ f\ x$$

Using our old evaluation relation, we would substitute x with 20 and transform the named expression to $\lambda y.20 + y$, but leave the body unchanged, since it has another binding of x to 10. Therefore, we would effectively evaluate this program, where there is no scope for getting two x's mixed up:

$$\mathsf{let}\ f = \lambda y.20 + y\ \mathsf{in}\ \mathsf{let}\ x = 10\ \mathsf{in}\ f\ x$$

If we use environments naively and simply evaluate the function to $\lambda y.x + y$ without substitution, we'll be in trouble. In the context where this function is applied x is bound to 10 instead of 20.

To solve this problem, functions are no longer values. Instead, functions evaluate to *closures*, which are triples $\langle \rho', x, e' \rangle$, where (1) ρ' is the environment in which the function was defined (i.e., in our example, the environment where x is correctly bound to 20), (2) x is the name of the function's argument, and (3) e' is the function body. In function application (the App rule), we are careful to evaluate e' in the environment in which it was defined, and *not* the environment in which the application occurs.

Correctness At this point, we have two ways of defining the semantics of an expression: either $e \Downarrow v$ or $\rho, e \Downarrow v$. The environment-based semantics is more efficient than the substitution-based semantics, but it *must* produce results that are equivalent to the original semantics. So, we may try to prove that both semantics map e to the same value v. However, the definition of v has changed: our old values had functions, but our new values have closures.

However, notice that:

$$\langle y \mapsto 10, x, x + y \rangle \approx \lambda x . x + 10$$

i.e., we can eliminate the environment in the closure by substituting its variables into the body. We could formalize this intuition to relate our old and new notions of values and prove that the two semantics are equivalent to each other.

Performance Our motivation for introducing environments was that substitution is an expensive operation. But, are environments cheap? Notice that environments cannot be represented as traditional hash tables. E.g., in the App rule, ρ is augmented with x to evaluate e_1 but e_2 is evaluated with just ρ. In addition, suppose a closure holds an environment ρ that is later extended. When the closure is eventually evaluated, it must be evaluated with the original environment (without any future extensions). Therefore, it seems as though an implementation would require a functional map data structure, which is not terrible efficient.

An efficient implementation would use a technique known as *de Brujin Indices*. de Brujin indices leverage the insight that we can always calculate the static distance between an identifier and the let-expression that introduced it (similarly for functions). Therefore, in this program:

$$\mathsf{let}\ x = 10\ \mathsf{in}\ \mathsf{let}\ y = 10\ \mathsf{in}\ x + y$$

We can replace y with v_0, since y is the innermost enclosing let-expression and x with v_1 since x is the next let-expression:

$$\mathsf{let}\ \cdot = 10\ \mathsf{in}\ \mathsf{let}\ \cdot = 10\ \mathsf{in}\ v_1 + v_0$$

The following example is a little trickier:

$$\mathsf{let}\ x = 10\ \mathsf{in}\ \mathsf{let}\ y = 5 + x\ \mathsf{in}\ x + y$$

This example has two references to x. The first x refers to the innermost let-expression v_0. However, the second x refers to the next let-expression, since y is v_0 in its context:

$$\mathsf{let}\ \cdot = 10\ \mathsf{in}\ \mathsf{let}\ \cdot = 5 + v_0\ \mathsf{in}\ v_1 + v_0$$

Since this technique gives us fixed indices for variables, we can use array-like data structures to lookup variables in constant time.

There are other issues to consider in a real implementation. For example, how long does it take to allocate a closure in memory? How much memory do closures consume? When can they be allocated on the stack? Are there cases where they don't need to be allocated at all?