Announcements:

- PS #6 due Thursday December 1, 11:59PM
- Prelim #2 tonight, 7:30-9PM, in B17 Upson
  - Graded until very late
- Guest lecture on Tuesday
- Final quiz on Tuesday 11/29

The Environment Model

- So far, we've used the substitution model to understand how OCaml programs evaluate.
  - The substitution model is very simple and mechanical.
  - Although there are a zillion cases to deal with in even a semi-realistic language, everything is reduced to a set of well-defined rules that govern the evaluation process.

- The basic idea is simple:
  - evaluate subexpressions to values,
  - and when you have a function call,
    - substitute the argument value for the formal parameter within the body of the function,
    - and then evaluate the resulting expression.

- But the substitution model is not without its shortcomings.
- First, it's not straightforward to extend the model with support for side effects (e.g., ref-assignment or array updates.)
- Second, it's not a very efficient or realistic model of how we really evaluate OCaml programs.
- In this lecture, we will introduce a somewhat more realistic model called the environment model that is a little closer to how the interpreter actually operates.
• To understand the environment model, let’s go back and revisit the substitution model on a very small subset of OCaml. The subset we will consider here is as follows:

\[ e ::= c \mid id \mid \text{fun } id \to e \mid (e_1 e_2) \]

• where \( e \) represents an expression, \( c \) a constant, \( id \) an identifier, \( \text{fun } id \to e \) a function, and \( (e_1 e_2) \) an application of a function to an argument.

• In the substitution model, we evaluate expressions according to the following inductive rules:

\[
\begin{align*}
\text{eval}(c) &= c \\
\text{eval}(id) &= \text{Error} \\
\text{eval}(\text{fun } id \to e) &= \text{fun } id \to e \\
\text{eval}((e_1 e_2)) &= v \\
\text{where} &\quad (\text{fun } id \to e) = \text{eval}(e_1) \\
&\text{and} \quad v_2 = \text{eval}(e_2) \\
&\text{and} \quad e' = \text{subst}(v_2, id, e) \\
&\text{and} \quad v = \text{eval}(e')
\end{align*}
\]

• where \( \text{subst}(v, id, e) \) is the expression that results from substituting the value \( v \) for all free occurrences of the identifier \( id \) in the expression \( e \).

• The substitution operator \( \text{subst} \) is defined formally inductively by:

\[
\begin{align*}
\text{subst}(v, id, c) &= c \\
\text{subst}(v, id, id') &= \text{if } id = id' \text{ then } v \text{ else } id' \\
\text{subst}(v, id, \text{fun } id' \to e) &= \\
&\text{if } id = id' \text{ then } (\text{fun } id' \to e) \text{ else } (\text{fun } id' \to e') \\
&\text{where } e' = \text{subst}(v, id, e) \\
\text{subst}(v, id, (e_1 e_2)) &= (\text{subst}(v, id, e_1) \text{ subst}(v, id, e_2))
\end{align*}
\]
• For this fragment of the language, all of the action occurs in function applications.

• Recall that to apply a function,
  o we first evaluate the function expression until we get a function value,
  o then we evaluate the function argument,
  o then substitute the argument for all free occurrences of the function parameter within the body of the function,
  o then finally evaluate the resulting expression.

• Now consider that when we substitute \( v2 \) for \( id \) in \( e \), we must crawl over all of \( e \) looking for free occurrences of the variable \( id \).
• Afterwards, we must crawl over the resulting expression again in order to evaluate it.
• Clearly, this is a very inefficient process, as we're crawling over the same expressions again and again.
Combining Substitution and Evaluation

- How can we avoid crawling over expressions twice, once for substitution and once for evaluation?
- One idea is to do them both at once. For example, we could rewrite the eval code for functions as follows:

\[
\text{eval}((e_1 \ e_2)) = \nu \\
\text{where } (\text{fun } id \rightarrow e) = \text{eval}(e_1) \\
\text{and } \nu_2 = \text{eval}(e_2) \\
\text{and } \nu = \text{eval\_and\_subst}(e, id, \nu_2)
\]

- \text{eval\_and\_subst}(e, id, \nu_2) will eval e, remembering to replace id by \nu_2.
- We want to combine evaluation and substitution into a single pass over the expression.

- So how would we write the function \text{eval\_and\_subst}? Here's a first attempt.
- For constants, substitution doesn't do anything, and evaluation doesn't do anything—they both return the same constant.
- So \text{eval\_and\_subst} on constants should just return the constant:

\[
\text{eval\_and\_subst}(c, id, \nu_2) = c
\]

- For variables, substitution checks to see if the variable is the same as the one we're supposed to substitute.
- If so, it returns the value being substituted.
- If not, it leaves the value alone.
- Eval on a variable is undefined, and eval of a value is always that value.
- So, when we put the two together we get:

\[
\text{eval\_and\_subst}(id', id, \nu_2) = \text{if } id = id' \text{ then } \nu_2 \text{ else Error}
\]
So far, so good. For applications, we simply \texttt{eval\_and\_subst} the subexpressions and then do what we did before:

\[
\text{eval\_and\_subst}((e1, e2), id, v2) = v \\
\text{where } (\text{fun } id' \to e) = \text{eval\_and\_subst}(e1, id, v2) \\
\text{and } v2' = \text{eval\_and\_subst}(e2, id, v2) \\
\text{and } v = \text{eval\_and\_subst}(e, id', v2')
\]

So far, we've been able to combine substitution and evaluation.

But when we run into functions, it's difficult to combine the two.
The problem is that substitution needs to crawl over the body of the function, but evaluation does not.

Recall that \texttt{eval} of a function always returns the function with the body unevaluated.
We can't evaluate the body yet because we don't have a value for the parameter.
So, the idea of combining evaluation and substitution seems to break down.

Once we hit a function, we have no choice but to do the substitution separately, and then do the evaluation later, when the function is applied:

\[
\text{eval\_and\_subst}((\text{fun } id' \to e), id, v2) = \text{subst}(v2, id, \text{fun } id' \to e)
\]

While this certainly works, and is a bit more efficient than the substitution model, it's not quite satisfying.
In the next section, we'll discuss how we can always combine substitution and evaluation so that we never process code twice.
The basic idea is to be extremely lazy!
The Environment Model

- As we saw above, the basic idea of the environment model, as opposed to the substitution model, is to combine the process of substitution with the process of evaluation into a single pass over the code.
  - But we ran into problems with functions, because we need to substitute within their body, and yet we can't evaluate their body—at least until they're applied.

- But what if we were lazy about performing the substitution?
  - Instead of actually doing the substitution when we encountered the function, what if we made a promise to do the substitution at the point when the function was applied?

- Then we could continue to combine substitution and evaluation.

- Of course, the problem with this is that, when we go to apply the function, we'll need to substitute two things: the original value and variable that we were substituting, and the argument and formal parameter for the function.

- In fact, in general, we may need to substitute an arbitrary number of values for variables that we have deferred.

- So, we must rewrite the eval_and_subst code so that it takes an expression and a substitution of arbitrary size.
  - This substitution provides values for all of the free variables in the code.
  - When we encounter a variable during evaluation, we simply look up the variable's value in the substitution.

- That is, the substitution that we carry around during evaluation can serve as a dynamic environment that provides bindings for the free variables of the code. That's why we call this the environment model.
There are a number of ways to represent environments (i.e., substitutions).
Perhaps the easiest is to just use an association list, a list of pairs of which the first component is a variable and the second component is the variable's associated value.
When we want to lookup a variable's value, we walk down the list until we find the same variable and then return the associated value.
There's one more detail that we need to flesh out: when we go to evaluate a function, we're going to delay substitution.
We do this by building a data structure called a closure.

- A closure is just a pair of the function and its environment, and represents a promise to substitute the values in the environment whenever we go to evaluate the function.
- So, a closure is nothing more than a lazy substitution.

**Evaluation Rules for the Environment Model**

- To make the discussion above precise, we can write down a formal set of evaluation rules for the environment model.
- We begin by defining our values as either constants or closures.
- A closure is a pair of a function and a substitution, and that a substitution is an association list, mapping identifiers to values.
  - We use curly braces to denote a closure object:

  \[ \nu ::= c \mid \{\text{fun} \ id \to e\}, S \]

- Now we can write the evaluation rules for the environment model.
- The `eval` function now takes an extra input, an environment `S`, as evaluation in the environment model is always with respect to an environment.

  \[
  \begin{align*}
  \text{eval}(c, S) &= c \\
  \text{eval}(id, S) &= \text{lookup}(id, S) \\
  \text{eval}(\text{fun} \ id \to e, S) &= \{\text{fun} \ id \to e, S\} \\
  \text{eval}((e_1 \ e_2), S) &= \nu \\
  \text{where} \ \{\text{fun} \ id \to e, S'\} &= \text{eval}(e_1, S) \\
  &\text{ and } \nu_2 = \text{eval}(e_2, S)
  \end{align*}
  \]
and \( v = \text{eval}(e, (id, v2)::S') \)

- That's it! Note that when evaluate a function, we return a closure containing the current environment \( s \).
- When we evaluate a function call \((e1\ e2)\), we first evaluate \( e1 \) in the current environment to get a closure, and then evaluate \( e2 \) in the current environment to get a value \( v2 \).
  - The closure for \( e1 \) has its own environment \( s' \).
  - When we evaluate the body of the function, we must make sure to fulfill the promise of the closure and use its environment (\( s' \)) as we evaluate.
  - We must also extend the environment so that when the formal parameter of the function \( id \) is encountered, we know that its value is \( v2 \).
- Although the environment model appears simple, it's actually fairly subtle. You should practice evaluating some expressions using the environment model to see how they work out.