Handout 8: Environment model of evaluation

Reduction semantics is an abstract system to understand program behaviour. However, it is not well-suited for implementation on computers. The reasons are that it works with expressions (which form the “source code” of programs), and it has to continually transform them by doing textual substitutions. In contrast, computers work by remembering the values of variables in memory, rather than substituting them into program code. A similar model of evaluation can be developed at an abstract level, where the values of variables are remembered in structures called environments. In this Handout, we look at how to evaluate programs in call-by-value programming languages based on the lambda calculus using environments.

In addition to environments, we introduce the so-called “big-step” semantics for describing the operational behaviour of programs. Each reduction step can be viewed as one single step of the execution of a program. At the end of such a reduction step, we obtain another expression which requires further evaluation through reductions. If we carry out all the reductions required for the evaluation of the expression, we eventually reach a “value,” i.e., a term that requires no further evaluation. The process of going from the original expression all the way till its value is reached is called a “big-step” evaluation.

The environment model is well-suited to describing the evaluation of programs in terms of “big-steps” (in contrast to the reduction model, which is better expressed in terms of “small steps.”)

Basic concepts

1. Environment

In order to evaluate expressions with variables, we maintain a table of values for all the variables. Such a table is called the binding environment or just environment for short. For example \( E = \{x \mapsto 2, y \mapsto 3, z \mapsto 4\} \) is an environment that gives values for three variables \( x, y \) and \( z \).

The notation we are using for environments is that of functions (or mappings) in mathematics. However, the notation is not particularly important. We could have used any notation for tables.

2. Evaluation using environments

Consider evaluating the arithmetic expression \( x + (y \times 4) \) using the above environment \( E \). We proceed as follows:

1. The value of \( x \) using the environment \( E \) is 2. (We just look up the value of \( x \) in \( E \).)
2. The value of \( y \times 4 \) using the environment \( E \) is 12. But this requires two subsidiary steps and one calculation.
   2.1. The value of \( y \) using the environment \( E \) is 3. (We look up the value of \( y \) in \( E \).)
   2.2. The value of 4 using the environment \( E \) is just 4. (In fact, since 4 is a constant, its value would be 4 no matter what environment is used.)
3. Therefore, the value of \( y \times 4 \) using the environment \( E \) is \( 3 \times 4 = 12 \).
4. Hence the value of \( x + (y \times 4) \) using the environment \( E \) is \( 2 + 12 = 14 \).

What has been done here is a “big step” evaluation. For instance, in step 2, we said that the value of \( y \times 4 \) is 12 even though this evaluation requires several subsidiary steps.

If we had done a “small step” evaluation, then we might have said that \( x + (y \times 4) \) becomes, in one step, the expression \( 2 + (y \times 4) \). Here, part of the expression \( x \) is evaluated but the other part, \( (y \times 4) \), is unevaluated. In this sense, small-step evaluation involves modifying program code. On the other hand, big-step evaluation leaves the code unchanged. To evaluate \( M_1 + M_2 \), we run the code of expression \( M_1 \) to obtain a value, say \( n_1 \), we run the code of \( M_2 \) to obtain another value, say \( n_2 \), and then add the two numbers.

3. Notation for big-step evaluations

Big-step evaluations involve statements of the form: “the value of \( M \) using the environment \( E \) is \( V \)”. We denote this by the formula \((M, E) \downarrow V\). In this formula, \( M \) stands for an expression, \( E \) for an environment and \( V \) for a value.
Using this notation, the evaluation of the previous paragraph can be exhibited as a tree:

\[
\begin{array}{c}
(x, E) \downarrow 2 \\
E, x, V \\
(x + (y * 4), E) \downarrow 14 \\
\end{array}
\]

4. Environments, abstractly
As mentioned earlier, the notation used for writing environments is not particularly important. That is because most of our use of environments will be carried out through two basic operations:

- **lookup**\((E, x)\) looks up the binding of \(x\) in the environment \(E\) and returns it.
- **upd**\((E, x, V)\) returns an “update” of the environment with a binding of \(x\) for \(V\). If \(E\) already has a binding for \(x\), then that binding is discarded and the new binding \(x \mapsto V\) is added. If \(E\) does not have a binding for \(x\), then the new binding is simply added. Note that **upd** is a “function” that returns a new copy of the environment. The original environment \(E\) is not modified by the use of an **upd** operation.

For example, let \(E\) and \(E'\) be two environments with the following bindings:

\[
E = \{x \mapsto 2, y \mapsto 3, z \mapsto 4\}
\]

\[
E' = \{x \mapsto 5, y \mapsto 2, z \mapsto 4\}
\]

Then, we have **lookup**\((E, x)\) = 2, **lookup**\((E', x)\) = 5, **upd**\((E, x, 10)\) = \(\{x \mapsto 10, y \mapsto 3, z \mapsto 4\}\) and **upd**\((E, w, 10)\) = \(\{x \mapsto 2, y \mapsto 3, z \mapsto 4, w \mapsto 10\}\).

Environment evaluation for call-by-value lambda calculi

5. Beta reduction using environments
Consider the expression \((\lambda x. x + y)\ 1\), which is a beta-redex. We would like to evaluate it in the environment \(E = \{x \mapsto 2, y \mapsto 3, z \mapsto 4\}\).

Normal beta-reduction would have us do a textual substitution, so that we obtain the expression \(1 + y\). However, this involves modifying the program code, which we would like to avoid. Instead of substituting the argument for \(x\), we remember it as the value of \(x\) in the environment.

More precisely, we construct the environment **upd**\((E, x, 1)\), which consists of the bindings \(\{x \mapsto 1, y \mapsto 3, z \mapsto 4\}\), and evaluate the body of the function, \(x + y\), in this environment. The value obtained is \(4\).

Here is the general rule: To evaluate \((\lambda x. M)\ V\ in\ an\ environment\ E\), evaluate \(M\ in\ the\ environment\ **upd**\((E, x, V)\).

6. Environments are stored substitutions
The above rule gives us some insight into what environments are about. They are just delayed substitutions. Instead of substituting variables textually by their values, we remember the values in environments and look them up whenever necessary. The net effect should be just the same as beta-reduction by substitution.

The operations implied by the beta-reduction rule are just like the normal implementation of function calls in computers. We store the value of the argument in memory, to be used as the value of \(x\), and then execute the body of the function. When the body finishes execution, it will return a value, which is then returned as the value of the function call.

7. Lambda needs a special treatment
We do not often apply lambda functions to arguments right away. We might pass lambda functions to other functions or bind them to names using a **let** declaration (which has the same effect as passing them to functions - via the desugaring of **let**). This raises a tricky issue, which is reminiscent of the of free variable capture problem in doing substitutions.

Consider the expression \(((\lambda f. \lambda y. f\ y)\ (\lambda x. x + y)\ 1)\) where we are passing a lambda function \((\lambda x. x + y)\) as an argument to the first lambda function. Consider evaluating the whole term in the environment \(E = \{y \mapsto 3, z \mapsto 4\}\). The first beta reduction is simulated by updating the environment with a binding for \(f\):

\[
E_1 = **upd**\((E, f, (\lambda x. x + y))\) = \{y \mapsto 3, z \mapsto 4, f \mapsto (\lambda x. x + y)\}
\]

The second beta reduction is simulated by updating it further with a binding for \(y\):

\[
E_2 = **upd**\((E_1, y, 1)\) = \{y \mapsto 1, z \mapsto 4, f \mapsto (\lambda x. x + y)\}\]
The problem is that the original binding of \( y \) to 3 is now lost. If we evaluate \( f \ y \) in this environment, we get 2 instead of the correct result 4.

Notice that exactly the same problem arises if we carry out beta reductions without worrying about the free variable capture:

\[
(\lambda f. \lambda y. f \ y) (\lambda x. x + [y]) 1 \rightarrow \beta (\lambda y. (\lambda x. x + [y]) y) 1
\]

If we continue the reduction, we obtain the result 2. The problem is that the free variable \( y \) of the argument, shown in box, is brought into the scope of a new binding of \( y \). Hence its meaning is altered.

8. Closures

Let us pause for a moment and think abstractly. We have said that environments are delayed substitutions. So, if we use the notation \( \text{cls} \) obtain a closed term. The “closure” is a representation for the closed term that we would obtain in this fashion.

Lambda abstraction terms packaged up together with their environments are called closures. The term “closure” comes from the idea that the environments are delayed substitutions. If we carry out those substitutions then we obtain a closed term. The “closure” is a representation for the closed term that we would obtain in this fashion.

We use the notation \( \text{cls}(M, E) \) to denote the closure consisting of a term \( M \) and an environment \( E \). When lambda functions are passed as values to other functions, they are packaged up with the original environments as closures. So, the environments \( E_1 \) and \( E_2 \) in paragraph 7 should bind \( f \) to a closure:

\[
E_1 = \text{upd}(E, f, \text{cls}(\lambda x. x + y, E)) = \{ y \rightarrow 3, z \rightarrow 4, f \rightarrow \text{cls}(\lambda x. x + y, E) \}
\]

\[
E_2 = \text{upd}(E_1, y, 1) = \{ y \rightarrow 1, z \rightarrow 4, f \rightarrow \text{cls}(\lambda x. x + y, E) \}
\]

When we evaluate \( f \ y \) in the environment \( E_2 \), the function term \( f \) evaluates to a closure, \( y \) evaluates to 1, and then we evaluate the body of the lambda function by switching to the environment \( E \) stored in the closure. Thus, the original binding of \( y \) will be used for the evaluation of \( x + y \).

It is best to remember the subtlety involved in lambda abstraction terms (i.e., functions) in terms of the following concepts:

Environment at the point of definition When a function is defined, there is an environment that is in effect. The values of the free variables of the function should be obtained from this environment.

Environment at the point of call When a function is called, there is an environment that is in effect in the calling context. This environment should be ignored inside the function call.

In our example, the environment at the point of definition for \( \lambda x. x + y \) is \( E \). The value of \( y \) should be looked up here. The environment at the point of call is \( E_2 \). This environment should be ignored inside the function call.

9. Lisp and dynamic binding

When the programming language Lisp was first implemented, these subtleties of variable binding were not understood. So, Lisp did not use closures to represent lambda abstractions passed as arguments to other functions. The environments used in evaluating function bodies are the environments at the point of call.

Even though this is a bug in the implementation of Lisp, over time, it has turned into a feature. The feature is called “dynamic binding”, i.e., binding of variables is determined dynamically, by the order in which functions get called. In contrast, the legitimate form of binding is called “static binding”, i.e., the binding of variables is determined by static nesting of variable scopes.

10. Big-step semantics for pure lambda terms

We are now ready to present the environment-based evaluation rules for pure lambda terms (ignoring constants). Recall that \( (M, E) \Downarrow V \) is the notation for saying that the value of \( M \) using the environment \( E \) is \( V \). The term \( M \) can be a variable \( x \), a lambda abstraction \( \lambda x. M' \) or a function application \( M_1 \ M_2 \). We have one rule for each.

\[
(x, E) \Downarrow \text{lookup}(E, x) \quad (\lambda x. M, E) \Downarrow \text{cls}(\lambda x. M, E) \quad (M, E) \Downarrow \text{cls}(\lambda x. M', E') \quad (N, E) \Downarrow V \quad (M', \text{upd}(E', x, V)) \Downarrow V' \quad (M \ N, E) \Downarrow V'
\]

The explanation of these rules is as follows:
• To evaluate \( x \) in an environment \( E \), look up the value of \( x \) in the environment.

• To evaluate \( \lambda x. M \) in an environment \( E \), package up the lambda abstraction along with the environment into a closure.

• To evaluate a function application \( M N \) in an environment \( E \):
  
  **GO LEFT:** evaluate the function term \( M \), letting it evaluate to a closure \( \text{cls}(\lambda x. M', E) \),

  **GO RIGHT:** evaluate the argument term \( N \), letting it evaluate to a value \( V \), and

  **REDUCE:** extract the body of the function, \( M' \), and evaluate it an updated environment of \( E' \) with a new value for \( x \).

(Note that the environment at the “point of definition” \( E' \) is used inside the function, and the environment at the “point of call” \( E \) is ignored.)

### Environment frames

#### 11. Frames

In the normal implementation of programming languages, the environment is not stored as a single monolithic table (or storage area). Rather, it is composed of a number of small tables, each of which arises from a single function call. Such small tables are called environment frames (or “frames” for short). In some contexts they are also called activation records because they arise from an activation of a function/procedure.

We will use the symbols with \( F \) to denote frames. An environment is then constructed by stringing together a number of frames, which we write as: \( F_0 \leftarrow F_1 \leftarrow \ldots \leftarrow F_n \). New environments are constructed by adding frames at the right of another environment. When variables need to be looked up, they are looked up in the frames starting from the right. The rightmost frame is thus the most recent frame and the leftmost frame is often the “global” frame where the global declarations and bindings are stored.

All said and done, what we have is a new and efficient representation of environments, where the basic operations lookup and upd are defined as follows:

- \( \text{upd}(E, x, V) \) is represented as \( E \leftarrow F \) where \( F = \{ x \mapsto V \} \). Note that we create a new frame and add it at the right, without modifying any of the older frames.

- \( \text{lookup}(E, x) \), looks up the binding of \( x \) in the most recent frame in \( E \) that has a binding for \( x \).

The two operations are implemented in such a way that the most recent binding of a variable is always looked up. Hence, this representation of environments is equivalent to the use of monolithic tables. It is just better for efficiency.

We will also assume that the names of frames \( F_0, F_1, \ldots \) can be used in other places, including closures. Thus they have the status of addresses or pointers in low-level implementation.

#### 12. Example

Consider the environments arising in the evaluation of

\[
((\lambda f. \lambda y. f \ y) \ (\lambda x. x + y) \ 1)
\]

starting from an environment with a single frame \( E = F_0 = \{ y \mapsto 3, z \mapsto 4 \} \).

Each of the lambda abstraction terms evaluates to a closure:

- \( (\lambda f. \lambda y. f \ y, E) \downarrow \text{cls}(\lambda f. \lambda y. f \ y, E) \).

- \( (\lambda x. x + y, E) \downarrow \text{cls}(\lambda x. x + y, E) \).

The application of the first closure to the second one involves constructing a new environment:

\[
E_1 = \text{upd}(E, f, \text{cls}(\lambda x. x + y \ E)) = F_0 \leftarrow F_1 \quad \text{where} \quad F_1 = \{ f \mapsto \text{cls}(\lambda x. x + y, E) \}
\]

The function returns the closure \( \text{cls}(\lambda y, f, E_1) \).

The application of this closure to 1 constructs another environment:

\[
E_2 = \text{upd}(E_1, y, 1) = F_1 \leftarrow F_2 \quad \text{where} \quad F_2 = \{ y \mapsto 1 \}
\]

The body of the function \( f \ y \) is evaluated in the environment \( E_2 \). This involves looking up the value of \( f \), which is \( \text{cls}(\lambda x. x + y, E) \), and the value of \( y \), which is 1. Applying \( f \) to 1 now involves a third environment:

\[
E_3 = \text{upd}(E, x, 1) = F_0 \leftarrow F_3 \quad \text{where} \quad F_3 = \{ x \mapsto 1 \}
\]

The function body, \( x + y \), is evaluated in \( E_3 \), where \( x \) has the value 1 and \( y \) has the value 3. The result is 4.