Environment model

Brigitte Pientka

October 9, 2015

Introduction

So far, we have taken a high-level view of the operational semantics. Our evaluation rules are based on substitutions. An advantage of the substitution model is that it is simple and easy to use in proving properties about programs. It provides a high-level abstract view of how programs are evaluated. Recall the rule for evaluating functions:

$$\begin{array}{lcl} \texttt{(fun } x \texttt{ -> } e\texttt{) } v & \longrightarrow & [v/x]e \\ \texttt{let } x = v \texttt{ in } e & \longrightarrow & [v/x]e \end{array}$$

Although this high-level view is convenient, because it abstracts over many implementation details and allows us to easily reason about programs and their behavior, it has also some drawbacks. One drawback is that we copy the value of v multiple times, if x occurred multiple times in the expression e. It would be nicer and more efficient, if we could just remember this binding between x and the value v in an environment, and if we need it during evaluation of the expression e, we just look it up in the environment. The other drawback of the substitution model is that it does not easily extend to references and assignment. To illustustrate, consider how one would evaluate

```
let x = ref 0 in
  (x := 3) ;  (!x)
```

If we use a substitution model this leads to

$$\begin{array}{l} [\texttt{ref 0/x}]\texttt{((x := 3) ; (!x))} = \\ \texttt{(ref 0) := 3 ; (!(ref 0))} \end{array}$$

This seems horribly wrong, since the result would be 0, instead of 3. We would read from a reference cell with content `0`, instead of reading from a reference cell with content `3`. The essence of the problem is that substitutions do not keep track of state, or update of a location in memory. Although the substitution model can be extended to keep track of state and memory locations, we will take a different path here in this note.

We will introduce a different evaluation model, the *environment model*. It will provide a different view of evaluation where the environment keeps track of the binding between a variable name and a value. It provides a lower level view of the operational semantics, which is one step closer to an implementation, and gives a good explanation for references.

First, let us introduce some terminology.

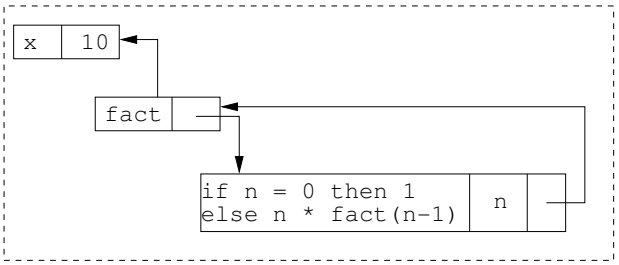
Terminology

1. A *binding* is the association between the name of a variable and a value. A name can be the name of a variable, the name of a function, or the name of a memory location. For instance, in the expression `let x = 10 in x + 3 end`, we encounter the binding between the name `x` and the value `10`. Similarly, in the expression `let square = (fun x -> x * x)`, we have the binding between the name of the function `square`, and its input argument `x` and the function body `x * x`. Finally, in the expression `let x = ref 2 in !x + 3`, we have the name `x` which is bound to a location in memory where we store `2`.

2. A frame is a collection of zero or more bindings, as well as a pointer to another frame, which is called its enclosing environment. An environment is a structured collection of frames, starting from a particular frame and going back through each frame's enclosing environment until the global environment is reached.

In the environment model, an expression is always evaluated in the context of a particular environment. The environment determines what values correspond to the names occurring in the expression. The purpose of an environment is to provide a way to associate a value with a particular name. The way this works is that the first frame in the environment is searched to see if it contains a binding for that name. If so, the associated value is used. If not, the first frame of the enclosing environment is searched, and so on up to the global environment. If the name is not found there, an error is reported.

There are three different possible bindings. We may bind a variable name to an integer, real, etc, or we may bind a variable name to a function (= closure) or we may bind it to a location in memory. Let's look at some examples. A binding is typically represented by a box with two parts. The left part has the name of the binding while the right part either contains the value (if the value is an integer, boolean, etc) or, if the structure is too large, it contains a pointer to another structure (= box).

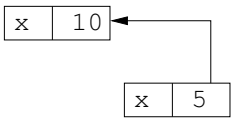



```
let x = 10;
let rec fact n = if n = 0 then 1
                 else n * fact(n-1)
```

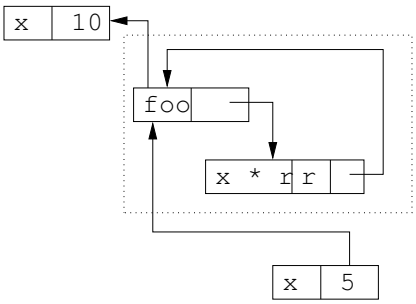
First, we encounter the binding of the name `x` to the value `10`. We have drawn a box with two parts, to represent this binding in the picture. Next, we encounter a function `fact`. The binding between the name of the function `fact` and a pointer another structure which contains the function body `if n = 0 then 1 else n * fact(n-1)`, the input argument `n`, and a pointer again back to the name of the function `fact`. Hence we build up a stack of bindings, where possibly later bindings can refer to earlier ones.

Note that the pointers or arrows are indicating where we need to look up the current binding. Since `val x = 10` is declared before the function definition of `fact`, there is a pointer from `fact` to the binding for `x`.

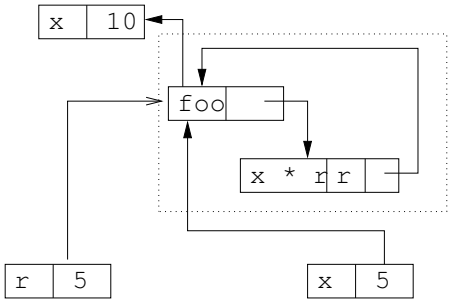
To look up a binding for `x`, we would follow the pointers (or arrows) until we find the first binding for it. So let us consider the next example.




```
let x = 10;;
let x = 5;;
```



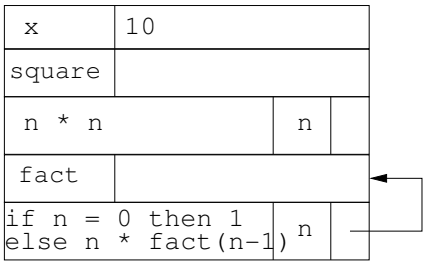

```
let x = 10;;
let foo r = x * r;;
let x = 5;;
```

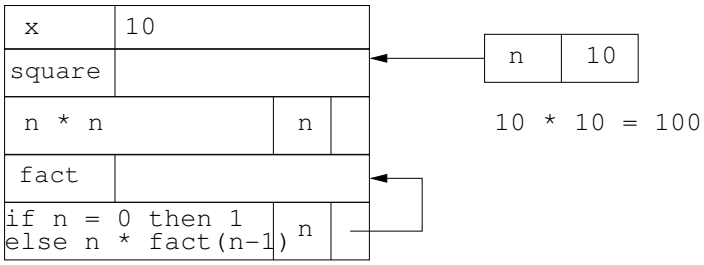



```
let x = 10 in
let foo r = x * r in
let x = 5 in
  foo x
```

The left part illustrates what happens if we have first a declaration `let x = 10` which is followed by another declaration `let x = 5`. The new binding for `x` is simply put next to the first binding for `x`. If we would insert a simple function `foo` betwen the two bindings, we can see that this new binding for `foo` will only refer to the first binding where `x` is bound to `10`, since it only can refer to earlier bindings. So in order to look up the value for `x`, we can follow the arrows, and will end up at `x` is bound to `10`. To the far right we see what happens when we evaluate `foo x` in this environment. We create a new binding between `r` and `5`. Note that this binding is pointing to the function `foo`. So when we evaluate the body of `foo`, and we have to look up the binding for `x`, we follow the pointers to where `x` is bound to `10`. Hence we must evaluate `10 * 5 = 50`.

Next, we show what happens when evaluating `square x` in the environment where we have defined `val x = 10`, `fun square n = n * n`, and `fun fact n = ...`. We will use a more compact representation of the bindings for `x`, `square`, and `fact`, and combine them in one frame, where we have multiple bindings.



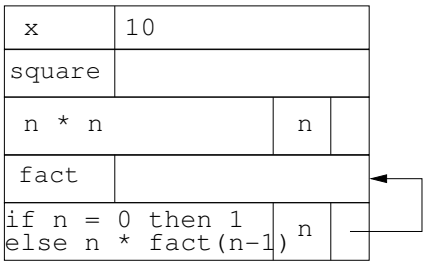


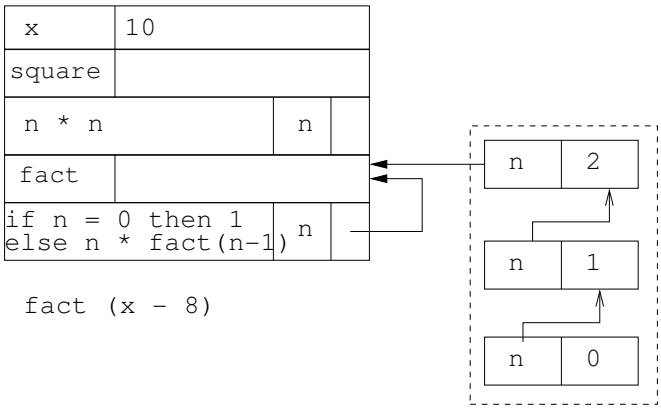

```
let x = 10;;
let square n = n * n
let rec fact n =
if n = 0 then 1
else n * fact(n-1)
```

```
square x
```

To evaluate `square x`, a new frame is created with a binding between the input argument `n` of the square function and the value `10`. Where did we get this value from? We looked above the definition for the `square` function. The body of square `n * n` will be executed with this binding. Hence we compute `10 * 10` which will evaluate to `100`. After evaluation the frame with the binding `n` to value `10` will be removed.

To illustrate what happens when we have recursion, consider evaluating the factorial function `fact`.



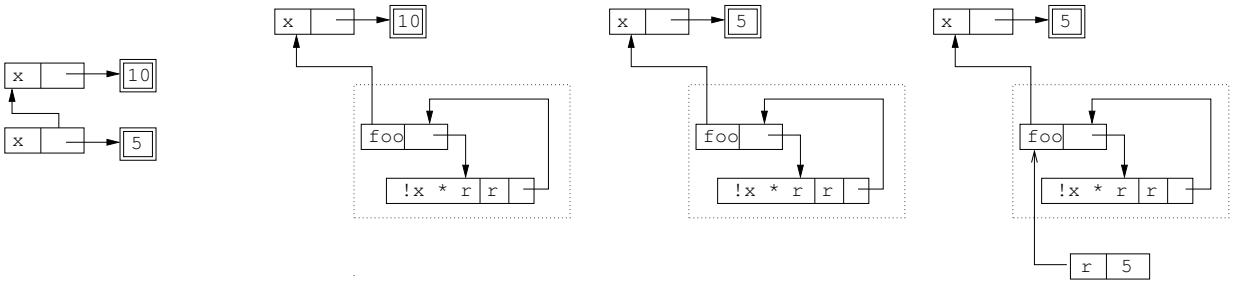



```
let x = 10;;
let square n = n * n;;
let rec fact n =
if n = 0 then 1
else n * fact(n-1);;
```

```
fact (x - 8)
```

Again a new frame is created to keep track of the binding between the input `n` to `fact` and its current value. In the first step we must evalue `fact (x - 8)`. Since `x` is bound to `10`, this means we need to bind the input argument `n` to `fact` to `2`. This is the top most binding in this frame. When executing the body of `fact`, namely `if 2 = 0 then 1 else 2 * fact(1)`, we call factorial recursively. Now the inupt argument is bound to `1`. Therefore we establish another binding for `n` which will point to the previous binding where `n` was `2`. So in every recursion step, we will keep track of the binding between the input argument and the current value it is bound to, until we have reached the final value. It is worth mentioning that the environment model only keeps track of bindings. The computation still to be done in each recursion is typically tracked by a run-time stack which we do not model in these notes.

As we have mentioned in the beginning, we can not only have a binding between a name and a number or a name and a function, but also between a name and a location. This way we can model references. References are represented by a name and a pointer to a memory location (or cell). To demonstrate the differences between the binding of a name introduced for example by a let-expression, and the binding of a name to a location, we will rewrite some of the previous examples slightly. First, consider what happens if we have two bindings for `x`, where the first one is a pointer to a location with content `10`, while the second one is a pointer to a location with content `6`.




```
let x = ref 10;;
let x = ref 5;;
```

```
let x = ref 10;;
let foo r =  (!x) * r;;
```

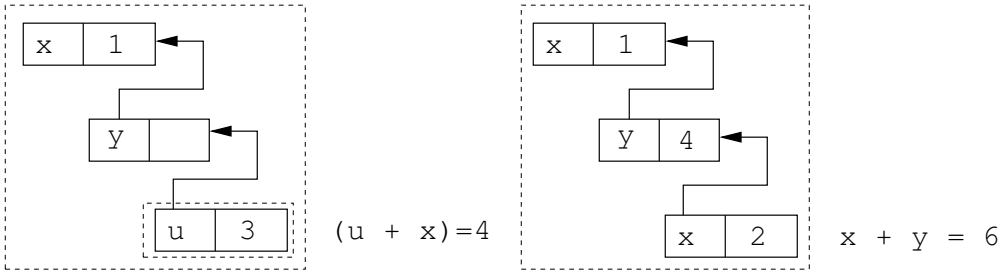
```
let x = ref 10;;
let foo r =  (!x) * r;;
x := 5;;
```

```
let x = ref 10;;
let foo r =  (!x) * r;;
x := 5;;
foo (!x);;
```

Let us consider what happens, if we modify the previous example slightly. In the middle, we see the representation for the function `fun foo r = (!x)*r`. The interesting bit comes when we update `x`, as seen on the right. Assignment directly modifies the cell associated with `x`, and replaces the previous value `10` with the new value `5`.

Next, we consider two more complicated examples:

```
let x = 1 in
let y = (let u = 3 in u + x) in
let x = 2 in
   x + y
```

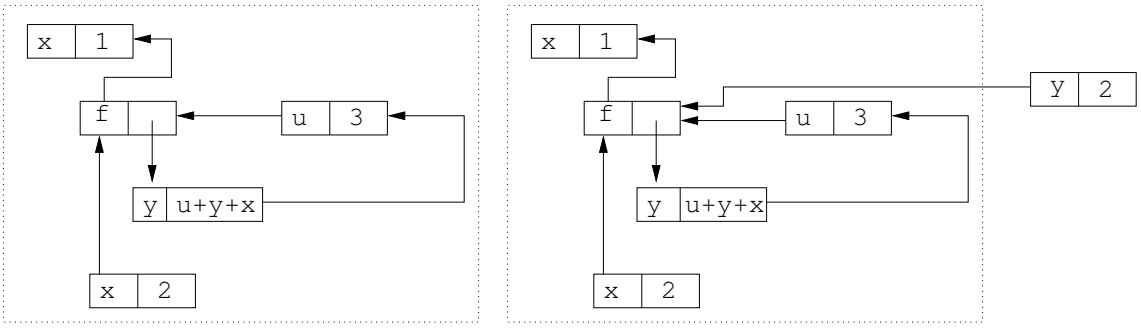


On the right, we see how the bindings for `x`, `y`, and `u` are established. Note that we do not yet have a value for `y`. The binding for `u` is only temporary. Once we have computed the value for `y`, we remove the binding of `u`. The left shows the next stage after computing the value for `y`.

A final example combines the ideas we have seen so far.

```
let x = 1 in
let f = (let u = 3 in (fun y -> u + y + x) end) in
let x = 2 in
  f(x)
```

The following model describes what happens. On the right we have the environment built before we execute the body of the let-expression `f(x)`. On the left, we see that the binding for `y` is established before the execution of the function body `u+y+x`.



3+2+1 = 6

The final result of this computation is 6.

Summary

The environment model can be summarized as follows:

An environment is a structured collection of *frames*. Each frame is a box (possibly empty) of bindings, which associate variable names with their corresponding values. (A single frame may contain at most one binding for any variable.) Each frame also has a pointer to its enclosing environment. The value of a variable with respect to an environment is the value given by the binding of the variable in the first frame in the environment that contains a binding for that variable. If no frame in the collection specifies a binding for the variable, then the variable is said to be unbound in the environment.