The environment model
Notes for COMP 302 Winter 2019

Prakash Panangaden

January 30, 2019

Introduction

Until now we have understood function application through the substitution model. The intuitive idea is that when a function f with a single parameter, say x, is applied to an *expression* e the evaluation proceeds as follows. First, the expression e is evaluated to produce a value v. Then all *free* occurrences of x in the body of f are replaced by v. The resulting expression is then evaluated.

This intuitive view is what is captured by the operational semantics. Our evaluation rules are based on substitutions, which were formalized by an inductive definition. 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 behaviour, it has also some drawbacks. The main drawback is that *this is not what actually happens*! If one were to implement the language literally using the substitution model then one must 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 correspondence between x and the value v, and if we need it during evaluation of the expression e, we just look it up. The other drawback of the substitution model is that it does not easily extend to references and assignment. We will worry about that later.

We will introduce a different evaluation model, the *environment model.* It will provide a different view of evaluation where a particular structure called *the environment* keeps track of the correspondence between a 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.

The `let` construct allows us to discuss the structure of the environment in detail. In F# one normally uses the lightweight syntax without the keyword `in`, but here I will not have room to show the whitespace necessary so I will use `in` in all my examples.

Terminology

1. A *binding* is an association between a name and a value. A name can name a value of any type such as an integer, a list a function or a memory location (a reference)[1]. For instance, in the expression `let x = 10 in x + 3`, 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`.
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.

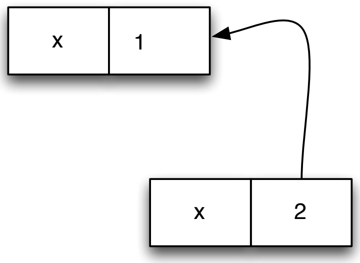
In the environment model, an expression is always evaluated in the context of a particular environment. The environment determines what values correspond to the names occurring in the expression. The purpose of an environment is to provide a way to associate a value with a particular name. The way this 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. The evaluator **never** follows a pointer backwards!

There are three different possible bindings. We may bind a name to an integer, float, etc, or we may bind a name to a function, or we may bind it to a location in memory. A function is a complicated entity, it is not just the body of the function. It is a more complex entity called a *closure*. We will discuss closures in some detail below.

Let us 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. For more complex entities like functions we will introduce some more notational conventions later.

To look up a binding for `x`, we follow the pointers (or arrows) until we find the first binding for it. Consider the first example.

[1] I will avoid the word "variable" because it suggests that things "vary"; sometimes they do, as with ref types, and at other times they do not.



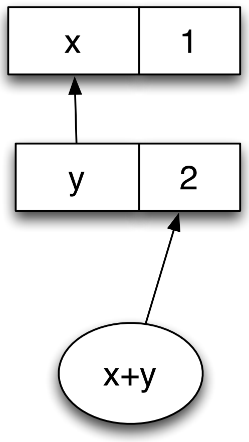
```
let x = 1
let x = 2
```

Here first `x` is declared with `let x =1` and then *inside the scope of this binding* we have another declaration for `x`. The structure is shown in the picture above. It is important to realize that we are **not changing the value of `x`** with the second declaration, we are creating a **new** `x` with a new binding in **its own frame**. *Both bindings exist at the same time.*

Now consider

```
let x = 1 in
  let y = 2 in
    x + y;;
```

The environment and the expression being evaluated is shown below.

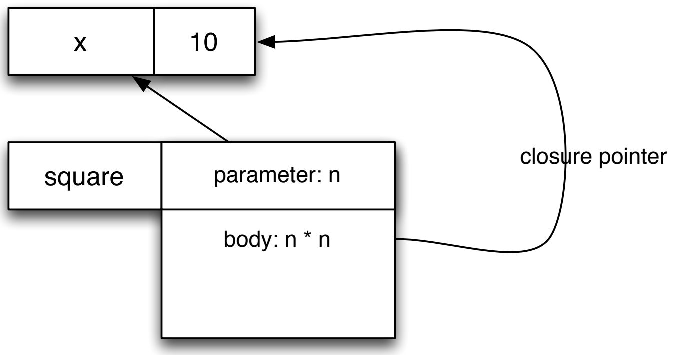


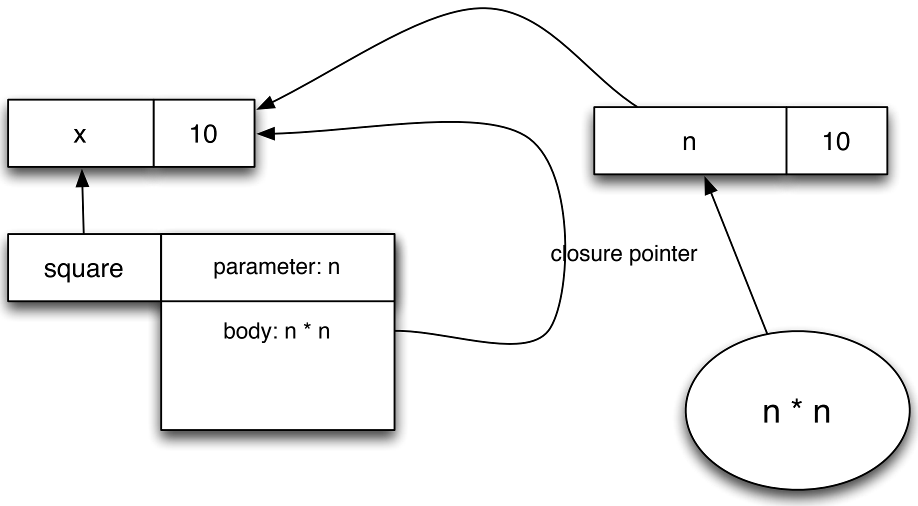
The arrows show the pointers from a frame to the enclosing frame. There is no problem with this example. We wish to evaluate the expression `x + y`; we need bindings for `x` and `y`. We follow the arrows looking for the binding in each frame. Incidentally, we also need a definition for the plus symbol: this is in the top-level frame which is loaded when F# starts up, it contains all the pre-defined names. We will look at more subtle cases later. Let us turn to function evaluation for now.

Next, we show what happens when evaluating `square x` in the environment where we have defined

```
let x = 10 in
let square n = n * n in
square x
```

This gives the environment shown in the first picture below. What happens if we try to evaluate `square x`? This expression is inside the inner `let` so it sees both the frames above. The first thing the evaluator looks for is the name "square"; it finds it and sees that it is indeed a function. It then looks for `x`, which is does not find in this frame, so it follows the pointer up and finds another frame where it does find the definition of `x`. Now it looks for the parameter name in the function definition and sees that it is `n`; **it creates a new frame** which is shown in the picture below, and binds `n` to the result it got from evaluating `x`. **This frame is temporary; it will be removed when evaluation is over**. Now the body is evaluated in the environment shown in the lower part of the picture.



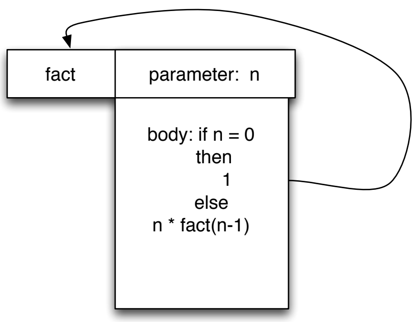


Why did the arrow from the (temporary) frame for `n` point to the frame for `x` rather than the frame for `square`? The new frame points to **the same place as the pointer from the function**. You see now that if inside the body of `square` there were another reference to `square` there would be an error. In short, functions defined this way cannot be called recursively. One has to do something special for recursive functions.

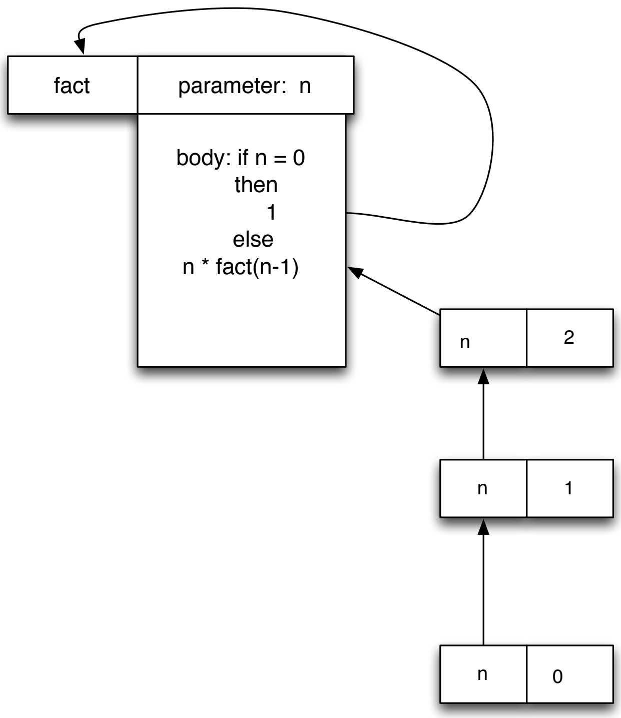
To illustrate what happens when we have recursion, consider evaluating the factorial function `fact`. Here is the code:

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

Notice we have sais `let rec`; this is a signal to send the arrow back to frame being created. The picture below shows the result.



Now consider what happens when we compute `fact 2`. The resulting environment is shown below:



We need to bind the input argument `n` of `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 input 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 in our discussion of the environment model we are only talking about keeping 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.

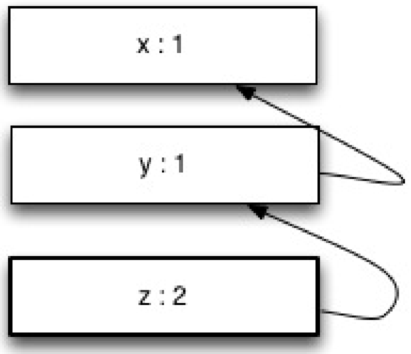
Local scopes

We can create local scopes by using `let`. The scope is delimited by the `let` keyword. Furthermore, when the expression evaluation is complete the *let bindings are removed.* Thus they are *local* and *temporary.* Every language has this feature, with whatever notation[2]. In Java, C and $C++$ one uses nested curly braces for exactly this purpose.

Consider the following:

```
let  x = 1 in
  let  y = x in
    let  z = 2 in
         y + z
```

Each nested `let` creates a new layer as shown in the figure.



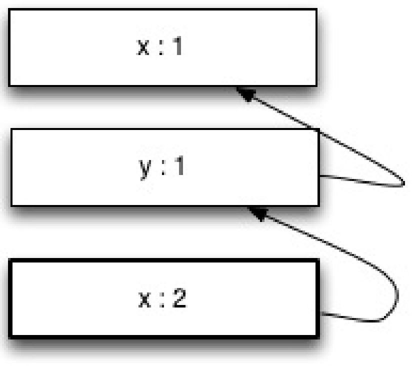
One *crucial* point: we do **not** store the binding as `y:x`. When the binding for `y` is established, the system evaluates `x` and finds the value 1. Thus, if later frames define new bindings for `x` the binding for `y` does **not** change. This discipline is called *static binding.*

Let me illustrate this point with a more subtle example.

[2]This is **not** a class about F#!

```
let  x = 1 in
  let  y = x in
    let  x = 2 in
         x + y
```

Does this give $2, 3$ or 4? The environment looks like this at the end

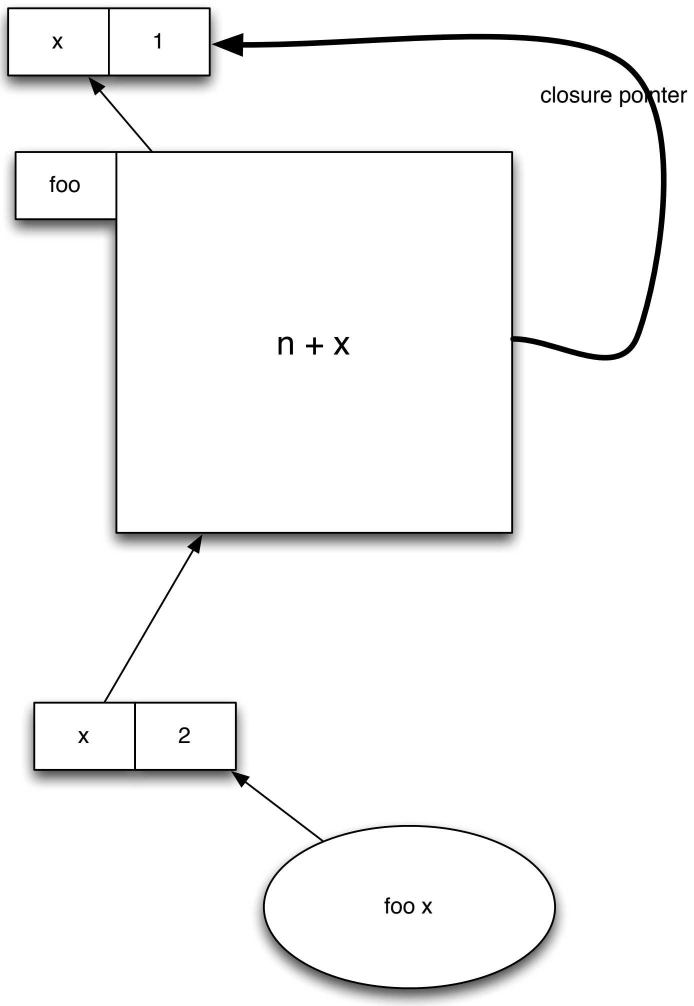


When we evaluate `x + y`, the `x` is looked up and the latest value 2 is seen but the `y` is bound to the same value as it was when it was set up: the bindings do not change, that is why it is called static binding.

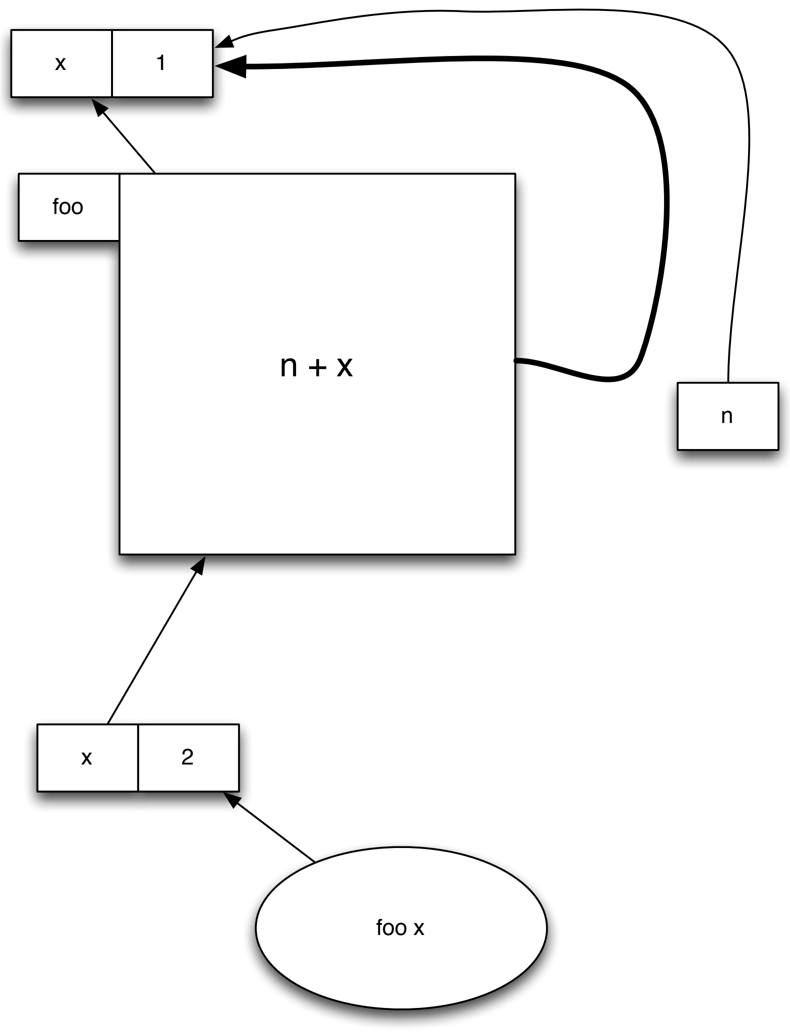
The same phenomenon can be illustrated with combinations of `let` and function definitions. Consider

```
let  x = 1 in
  let foo n = n + x in
    let  x = 2 in
         foo x
```

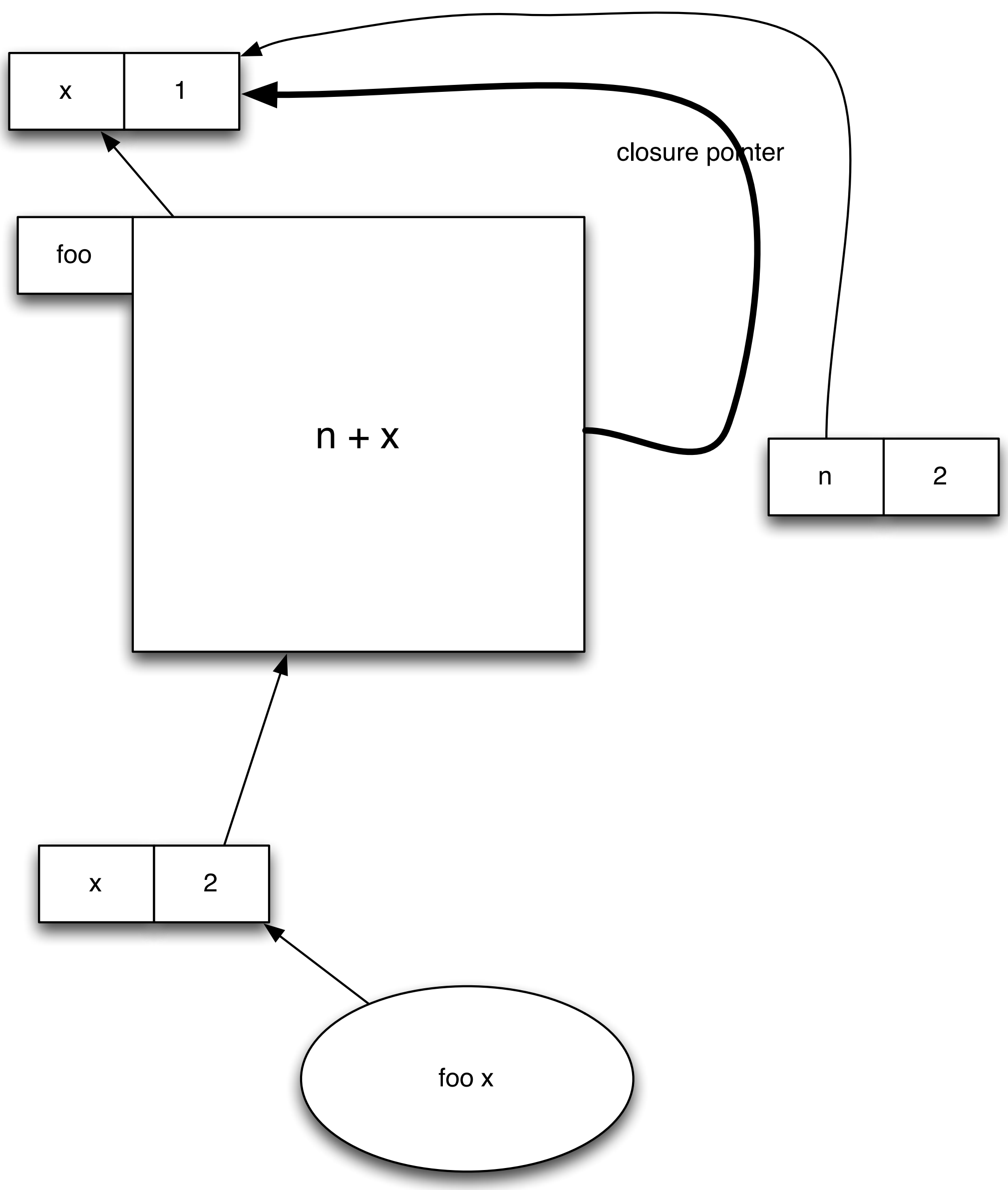
This gives the value 3 for exactly the same reason. When the function `foo` is defined it contains a pointed to the environment that exists *at the time that it is created*. The only binding for `x` is $(x, 1)$ at that point. Even though when `foo` is called there is a new binding for `x`, the environment inside the closure for `foo` will not have this new binding. This picture shows what the environment looks like just after the three bindings are set up and the evaluation of `foo x` is about to start.



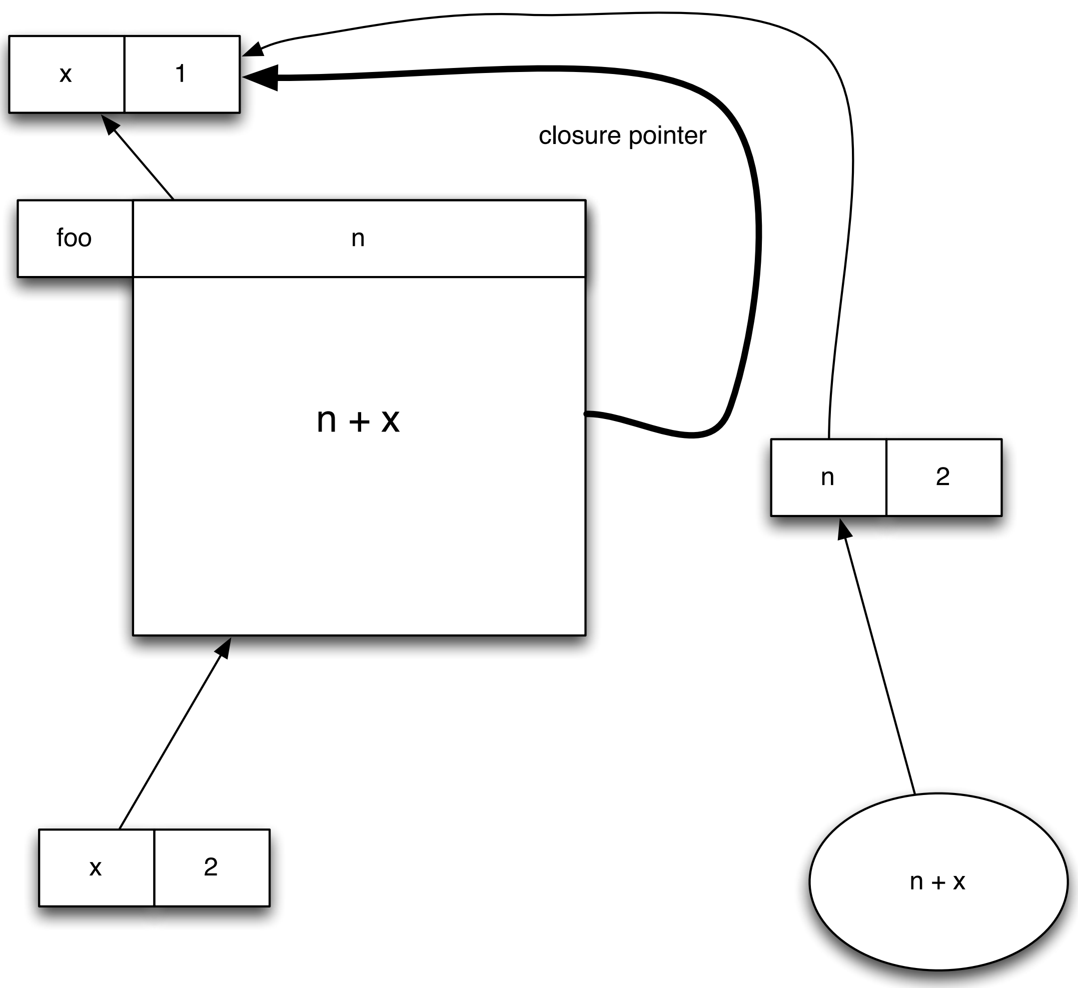
When `foo x` is evaluated, the name `foo` is looked up first and it is found to be a function with one parameter. Accordingly, a frame is set up for the parameter; **the pointer from this frame goes to wherever the pointer from the definition of foo goes.** The partially created environment now looks like this:



Functions are called by value so before the body can be executed *the argument is evaluated first*, thus `x` is looked up and the value 2 is returned. Then the 2 is bound to n and a new frame is created (this frame will be removed when the function evaluation is complete) and *placed on top of a copy of the environment in the closure as we indicated above.* The picture now looks like this:



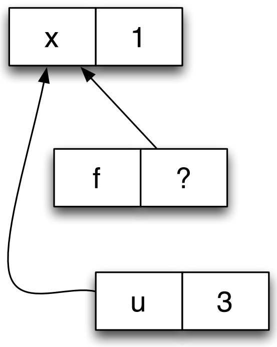
Now in evaluating the body `n + x`, two names have to be looked up. The `n` is found to be bound to 2 and the `x` is found to be bound to 1; the binding $(x, 2)$ will not be seen. The final picture looks like this:



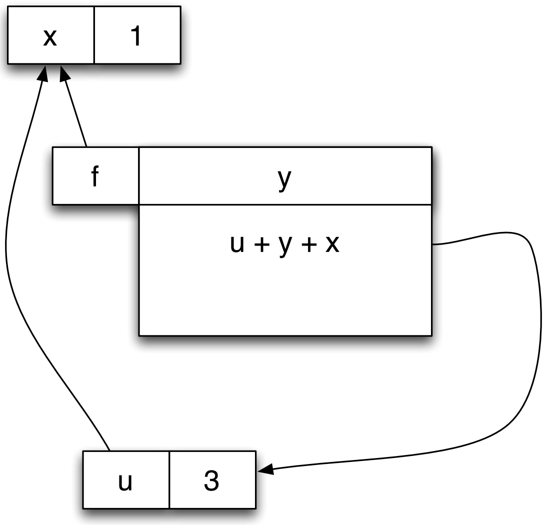
A final example combines the ideas we have seen so far. It is subtle so look at it carefully.

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

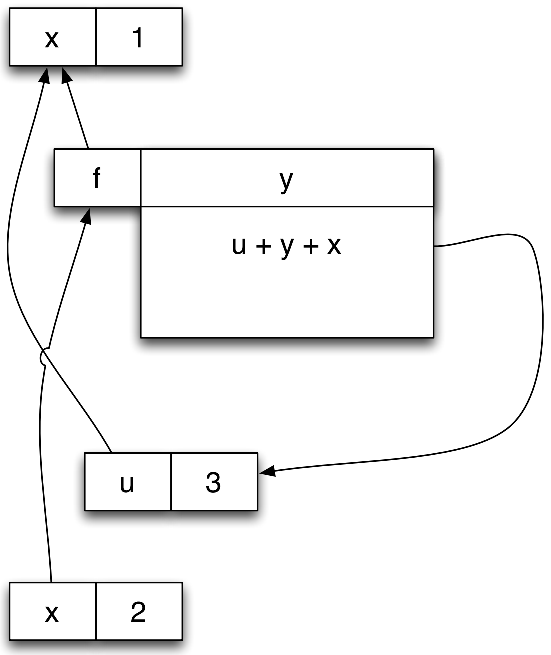
In the first picture below we see that the frame for the binding $(x, 1)$ is set up. The definition of f is not yet complete. We have to evaluate the expression on the right-hand-side of `let f = ...`, which starts with a `let u = ...`. Thus the frame for the binding $(u, 3)$ is set up. This points to the frame that exists at this time, namely to the top frame; the binding for f is not yet set up at this stage.



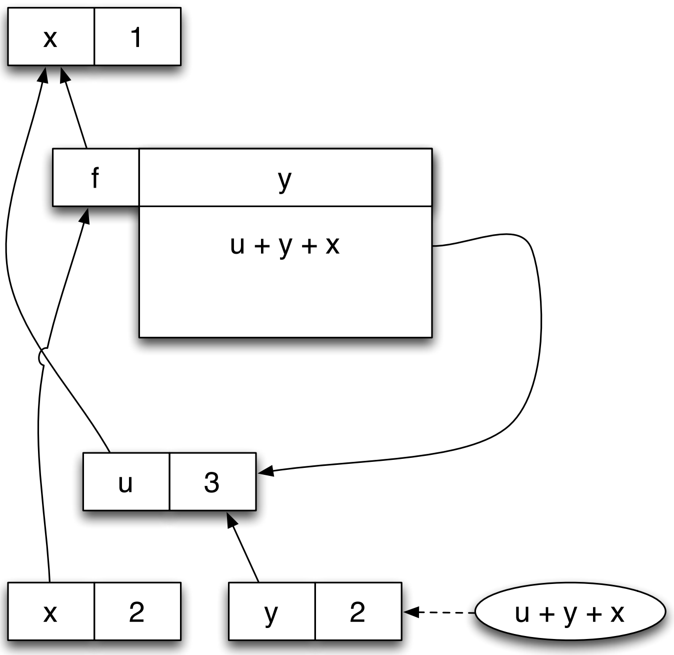
Then the closure for f can be constructed. It is shown below. Note carefully that frame $(u, 3)$ is alive and well and is the latest frame so the closure of f contains a pointer to this frame. *This traps the frame*; as long as f is alive, the u frame is also alive.



Lastly the frame for the inner x binding is set up; the `let` binding for u is closed (but it is still alive and trapped inside the closure for f) so the pointer from this frame goes to the frame for f.



Now we make the function call $f(x)$. The evaluation of x gives 2 so a new temporary frame binding $(y, 2)$ is set up. Where does it point? It points to whatever the closure of f points to f. The resulting diagram is shown below.



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 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 associated with a name with respect to an environment is the value given by the binding of the name in the first frame in the environment that contains a binding for that name. If no frame in the collection specifies a binding for the name, then the name is said to be unbound in the environment.

These notes were written originally for SML, then edited for OCaml and edited again for F# and then back to OCaml. This just shows that the concepts are all the same; there are only minor differences in syntax. These concepts are **not** valid for Python as far as I know[3].

[3] Which is not very far.