Environments and Local State 13

George Wang
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

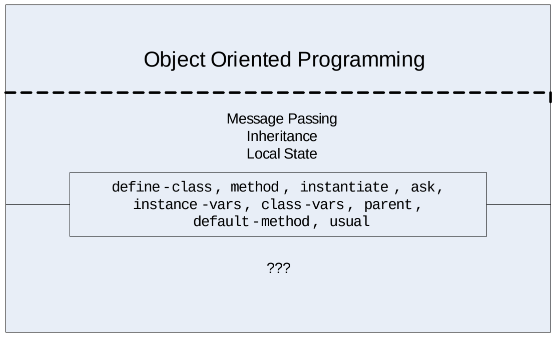
July 14, 2010

1 Above the Line: Review

Things we talked about:

`define-class, instantiate, method, ask, instance-vars, instantiation variables, class-vars, parent, default-method, usual, initialize`

2 The Line


Object Oriented Programming
Message Passing
Inheritance
Local State
define-class, method, instantiate, ask,
instance-vars, class-vars, parent,
default-method, usual
???

This is essentially the line we're talking about when we talk about above the line OOP. We want to talk about how it is that we implement message passing and inheritance and local state, and that's what we'll be attacking today and tomorrow.

A *local* variable is one that is only available in a specific small portion of the program. In Scheme, we'll generally be talking about variables that are only available within a specific procedure. As an example of local variable, we've seen that `let` is able make them. A *state* variable is one that remembers its value from one invocation to the next, and that's not something we've seen before.

3 From Functional to Local State

Before we can attack local state, let's try to remember some information in a global variable:

```
;;;;; In file cs61a/lectures/3.1/count1.scm
(define counter 0)
(define (count)
  (set! counter (+ counter 1))
  counter)
> (count)
1
> (count)
2
```

We can see that we've created a *state* variable. The variable `counter` is able to remember its value from one call of `count` to the next.

What's new here is the special form `set!` that allows us to change the value of a variable. This is not like `let`, which creates a temporary, local binding. This makes a permanent change in some variable that already existed. This is something we saw earlier this week with OOP, and the syntax is just like `define` (but not the abbreviation for defining a function): it takes an unevaluated name and an expression whose value provides the new value.

Again, we can't just use the substitution model of evaluation. If you think about the substitution model where we substitute variable names with values, we end up with:

```
(set! 0 (+ 0 1))
0
```

Instead, a better way to think about variables is that they're like addresses written on postcards. If you read an address, it tells you where to go to find somebody. In the same way, you can think that there are metaphorical city streets and house addresses inside the computer, and knowing the address of something allows you to find the piece of information there.

Another new thing is that a procedure body can include more than one expression. In functional programming, the expressions don't do anything except compute a value, and a function can only return one value, so it doesn't make sense to have more than one expression in it. But when we invoke `set!` there is an effect that lasts beyond the computation of that expression, so now it makes sense to have that expression and then another expression that does something else. When a body has more than one expression, the expressions are evaluated from top to bottom and the value returned by the procedure is the value computed by **the last expression**. All but the last are just for effect.

Alright, since we have global state, let's try to implement local state. Let's look at an attempt to do this, that *doesn't work*:

```
;;;;; In file cs61a/lectures/3.1/count.lose
(define (count)
  (let ((counter 0))                  > (count)
    (set! counter (+ counter 1))      1
    counter))                         > (count)
                                      1
                                      > (count)
                                      1
```

We are on the right track with using let, because we know that we can create a *local* variable with let. The problem here, is that this isn't a state variable! The let is executed every single time we call count, so it creates a new counter with the value of 0.

Now, let's finally look at something that works:

```
;;;;; In file cs61a/lectures/3.1/count2.scm
(define count
  (let ((result 0))
    (lambda ()
      (set! result (+ result 1))
      result)))
```

Notice that there are no parentheses around the word `count` on the first line! Instead of

```
(define count (lambda () (let ...)))
```

which is what the earlier version means, we have essentially interchanged the `lambda` and the `let` so that the former is inside the latter:

```
(define count (let ... (lambda () ...)))
```

But wait, why does this work? We have to start looking at the environment model of evaluation to fully answer that question. But a short intuitive explanation is that `let` creates local variables that are only accessible to things within the body of the let. Furthermore, the lambda is something that hangs around, which gives us the state half of the equation. With both of these, we have the local state that we wanted.

Using this, we can create the equivalent of objects. Instead of a single count that does what we want it to do, let's define a make-count procedure:

```
;;;;; In file cs61a/lectures/3.1/count3.scm
(define (make-count)                     > (define dracula (make-count))
  (let ((result 0))                      > (dracula)
    (lambda ()                           1
      (set! result (+ result 1))         > (dracula)
      result)))                          2
                                         > (define monte-cristo (make-count))
                                         > (monte-cristo)
                                         1
                                         > (dracula)
                                         3
```

Each of `dracula` and `monte-cristo` is the result of evaluating the expression `(lambda () ...)` to produce a procedure. Each of those procedures has access to its own local state variable called `result`. `Result` is temporary with respect to `make-count` but permanent with respect to `dracula` or `monte-cristo`, because the `let` is inside the `lambda` for the former but outside the `lambda` for the latter.

4 The Environment Model

Let's introduce the central issues about environments.

The question is, what happens when you invoke a procedure? For example, suppose we've said

```
(define (square x) (* x x))
```

and now we say `(square 7)`; what happens? The substitution model says:

1. Substitute the actual argument value(s) for the formal parameter(s) in the body of the function.
2. Evaluate the resulting expression.

In this example, the substitution of 7 for x in `(* x x)` gives `(* 7 7)`. In step 2 we evaluate that expression to get the result 49. We now forget about the substitution model and replace it with the environment model:

1. Create a frame with the formal parameter(s) bound to the actual argument values.

2. Use this frame to extend the lexical environment.

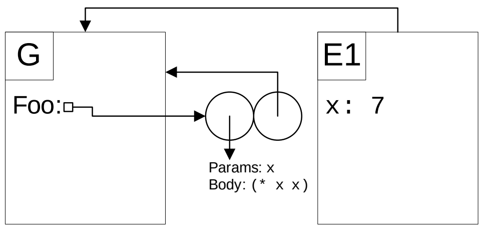
3. Evaluate the body (without substitution!) in the resulting environment.

A FRAME is a collection of name-value associations or bindings. In our example, the frame has one binding, from x to 7. Skip step 2 for a moment and think about step 3. The idea is that we are going to evaluate the expression `(* x x)` but we are refining our notion of what it means to evaluate an expression. Expressions are no longer evaluated in a vacuum, but instead, every evaluation must be done with respect to some environment—that is, some collection of bindings between names and values. When we are evaluating `(* x x)` and we see the symbol x, we want to be able to look up x in our collection of bindings and find the value 7.

Looking up the value bound to a symbol is something we've done before with global variables. What's new is that instead of one central collection of bindings we now have the possibility of *local* environments. The symbol x isn't always 7, only during this one invocation of `square`. So, step 3 means to evaluate the expression in the way that we've always understood, but looking up names in a particular place.

What's step 2 about? The point is that we can't evaluate `(* x x)` in an environment with nothing but the x=7 binding, because we also have to look up a value for the symbol * (namely, the multiplication function). So, we create a new frame in step 1, but that frame isn't an environment by itself. Instead we use the new frame to extend an environment that already existed. That's what step 2 says.

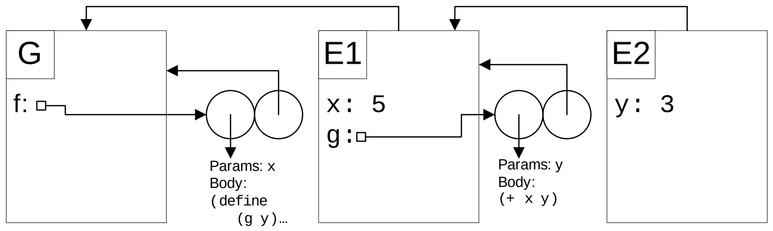
Thus, our complete environment diagram is:



Which old environment do we extend? In the `square` example there is only one candidate, the global environment. But in more complicated situations there may be several environments available. For example:

```
(define (f x)
  (define (g y)
    (+ x y))
    (g 3))
> (f 5)
```

When we invoke `f`, we create a frame (call it E1) in which x is bound to 5. We use that frame to extend the global environment (call it G). Now we evaluate the body of `f`, which contains the internal definition for `g` and the expression `(g 3)`. To invoke `g`, we create a frame in which `y` is bound to 3. Call this frame E2. We are going to use E2 to extend some old environment, but which? G or E1? The body of g is the expression `(+ x y)`. To evaluate that, we need an envoironment in which we can look up all of + (in G), `x` (in E1), and `y` (in E2). So we'd better make our new environment by extending E1, not by extending G:



Note that this doesn't show us something that in fact is very important, which is that we should not always extend the current environment. Instead, we should always extend the environment that the right bubble points to. This is what allows us to have our local state. Let's look at an example where this doesn't apply:

4.1 Exercise: `make-adder`

```
(define (make-adder n)
  (lambda (x) (+ x n)))
(define 3+ (make-adder 3))
(define n 7)
> (3+ n)
```

Make sure you are able to come up with this on your own!