Today's Lecture Notes for cs173

sk and dbtucker

September 25, 2000

Topic: The Big Bang!

A language with state

In Scheme, you can change the value of a variable using the function `set!`. We will study how to add such a feature to our language. First, we introduce the variant `setE` in our datatype (now called `AFun!Exp`):

```
(define-datatype AFun!Exp AFun!Exp?
                 [varE (v symbol?)]
                 [numE (n number?)]
                 [addE (lhs AFun!Exp?) (rhs AFun!Exp?)]
                 [setE (var symbol?) (val AFun!Exp?)]
                 [funE (var symbol?) (body AFun!Exp?)]
                 [appE (fun AFun!Exp?) (arg AFun!Exp?)])
```

We will also need the following operations (provided by Scheme) to mutate variables:

```
;; box : 'a -> Box ('a)
;; unbox : Box ('a) -> 'a
;; set-box! : Box 'a * 'b -> ()
```

These functions do the obvious: `box` puts a value in a box, `unbox` takes a value out of a box, and `set-box!` changes the value in the box.

For example, consider the following Scheme code:

```
(let ([x (box 5)])
  (set-box! x 6)
  (unbox x))
```

This expression evaluates to `6`. Note that:

```
(let ([x (box 5)])
  (set-box! x 6)
  x)
```

evaluates to `(box 6)`.

Now we will update our interpreter to handle `setE`. First of all, our environment now maps symbols to boxes of values, rather than symbols to values:

```
;; DSub = symbol -> Box (Value)
```

When we add variable to the environment (we only do so in `apply-fun`), we want to throw it in a box:

```
(define (apply-fun fv av)
  (cases AFunVal fv
         [funV (param body env)
               (interp body
                       (new-sub param (box av) env))]
         [else (error 'apply-fun "~s not an applicable value" fv)]))
```

When we substitute for a variable (the `varE` case of `interp`), we need to take it out of its box:

```
[varE (v) (unbox (get-sub v d))]
```

Now we just need to handle the `setE` case. Since every expression must compute to some value, we'll say the value of a `setE` expression is whatever the right-hand side computes to.

```
[setE (var val) (let ([box-v (get-sub var d)]
                      [val-v (interp val d)])
                  (set-box! box-v val-v)
                  val-v)]
```

The whole interpreter looks like this:

```
(define (interp a d)
  (cases AFun!Exp a
         [varE (v) (unbox (get-sub v d))]
         [numE (n) (numV n)]
         [addE (le re) (numV+ (interp le d) (interp re d))]
         [setE (var val) (let ([box-v (get-sub var d)]
                               [val-v (interp val d)])
                           (set-box! box-v val-v)
                           val-v)]
         [funE (var body) (funV var body d)]
         [appE (fe ae) (apply-fun (interp fe d) (interp ae d))]))
```

Order of evaluation

Now that we have state, order of evaluation matters. For example:

```
(let (x 3)
  (x + (set x 4)))
```

evaluates to 7, whereas:

```
(let (x 3)
  ((set x 4) + x))
```

evaluates to 8, assuming that the arguments are evaluate from left to right. Is this a fair assumption? Our interpreter says:

```
[addE (le re) (numV+ (interp le d) (interp re d))]
```

So our order of evaluation is going to be whatever Scheme's order is. Since Scheme evaluates arguments from left to right, so does our interpreter.

Lather, rinse, repeat

Let's try to write an infinite loop using our new language[1]. Here's one shot at it:

```
(let (f (fun (n) (f (n))))
  (f (1)))
```

This expression won't work because it's not even closed—the `f` in the body of the `fun` is unbound. We can fix this using `set`:

```
(let (f 0)
  ((set f (fun (n) (f (n))))
   1))
```

Here we first bind `f` to a dummy variable (`0`), then use `set` to bind it to a function. Now the `f` in the function refers to itself! So, the function applied to any argument (here, `1`) won't terminate.

[1] In fact, it's possible to write an infinite loop using just `AFunExp` (i.e. without state).