Today's Lecture Notes for cs173

sk and dbtucker

September 22, 2000

Theme: Having fun (in your language) is harder than it seems.

Eager vs. lazy semantics

Question 2 of homework 2 asked for an expression that shows the difference between eager and lazy semantics. To recall: "The two computations may differ in any way, so long as if they both yield a value, the value must not be the same number or a closure that produces the same value for all values it is applied to."

There were 3 popular answers:

1. ```
   (let (x (1 / 0))
     0)
   ```

   Under eager semantics, `(1 / 0)` is evaluated, which produces an error. Under lazy semantics, `(1 / 0)` is never evaluated, since `x` is not present in the body of let, `0`. Thus the result is `0`.

2. ```
   (let (x <infinite loop>)
     3)
   ```

   If we can write an `<infinite loop>` in our language, then this expression will not terminate under eager semantics, but will evaluate to `3` under lazy semantics.

3. ```
   (let (y 5)
     (let (x y)
       (let (y 4)
         x)))
   ```

   WRONG! This expression does not meet the requirements—it evaluates to `5` under both the eager and lazy semantics. The point of confusion is that an incorrect lazy interpreter will do dynamic binding here, and thus bind the `x` to the second `y`. A *correct* lazy interpreter will compute this expression to `5`. (Consider how this expression is evaluated lazily using substitution.)
   ```

An eager interpreter using Scheme's `lambda`

Question 3b asked for an interpreter that (a) uses Scheme's closures to represent closures in our language and (b) uses Scheme's function application to do application in our language. So, most of the code writes itself; we know that the `appE` case is a simple application in Scheme, and that `funE` must return a closure (i.e. a `lambda` expression).

```
(define (interp a d)
  (cases AFunExp a
         [numE (n) n]
         [varE (v) (get-sub v d)]
         [addE (le re) (+ (interp le d) (interp re d))]
         [multE (le re) (* (interp le d) (interp re d))]
         [funE (param body)
               (lambda (argval)
                 (...))]
         [appE (fe ae) ((interp fe d) (interp ae d))]))
```

The only question is the body of the `lambda` that is returned in the `funE` case. Let's look at a concrete example:

```
(interp ((fun (x) (x + y)) (2 + 3)) d)
```

where `d` is $\{\texttt{y} \mapsto \texttt{1}\}$. So far we know that this expression computes to:

```
((lambda (argval) (...)) 5)
```

Now the next thing that will get evaluated is `(...)`, and Scheme will bind `argval` to `5` in this expression. What we really want next is to evaluate the body of the `fun`—that is, `(x + y)`—where the environment `d` is extended by $\{\texttt{x} \mapsto \texttt{5}\}$. In other words:

```
(interp (x + 1) (new-sub 'x 5 d))
```

In general, we want to evaluate the `body`, where `param` is bound to `argval`. Now the code is easy to see:

```
(define (interp a d)
  (cases AFunExp a
         [numE (n) n]
         [varE (v) (get-sub v d)]
         [addE (le re) (+ (interp le d) (interp re d))]
         [multE (le re) (* (interp le d) (interp re d))]
         [funE (param body)
               (lambda (argval)
                 (interp body (new-sub param argval d)))]
         [appE (fe ae) ((interp fe d) (interp ae d))]))
```

A couple of notes:

1. The key observation here is that Scheme itself uses static binding. Therefore, when we write `d` in the `lambda` expression of the `funE` case, we know that it will be bound to the value of `d` *when the* ***`funE`*** *was evaluated*.

2. Since we're using Scheme's closures as values, we decided to use Scheme's numbers as values too.

The story so far

There were 4 interpreters for part 3 of the homework. The two variables are evaluation strategy (eager vs. lazy) and closure representation (datatypes vs. Scheme's lambda).

	datatype	lambda
eager	done 9/15	done today
lazy	?	next up

A lazy interpreter using Scheme's `lambda`

One is tempted to modify the eager interpreter by simply changing the `varE` and `appE` cases as follows:

```
[varE (v) (interp (get-sub v d) d)]
[appE (fe ae) ((interp fe d) ae)]
```

and hoping that all is hunky-dory. Unfortunately, this solution doesn't quite work. Let's look at an example (due to Rob) that fails:

```
(let (x 3)
  (let (f (fun (x) (2 * x)))
    (f ((x + 4))))
```

This expression should evaluate to `14`. Following through the evaluation, we eventually reduce the expression to `(2 * (x + 4))` in the environment $\{$`x` $\mapsto$ `(x + 4)`$\}$. Now we have a problem—this computation will not terminate.

The current problem is the same one we had when we introduced delayed substitutions: some expressions are not being evaluated in the environment in which they were created. In the above, `(2 * (x + 4))` should be evaluated in the environment $\{$`x` $\mapsto$ `3`$\}$. Our solution before was to create closures for function values; here, we will create closures for *all* expressions being bound by `lambda` application.

The key is this: when we have a function application, we need to capture the environment for the argument expression (which isn't evaluated right away). We create a *thunk*—a function of no arguments—whose body contains the computation we will do lazily. Then when we encounter a variable, we look it up in the environment and evaluate the thunk that is returned.

```
(define (interp a d)
  (cases AFunExp a
         [numE (n) n]
         [varE (v) ((get-sub v d))]
         [addE (le re) (+ (interp le d) (interp re d))]
         [multE (le re) (* (interp le d) (interp re d))]
         [funE (param body)
               (lambda (argval)
                 (interp body (new-sub param argval d)))]
         [appE (fe ae) ((interp fe d)
                        (lambda () (interp ae d)))]))
```

Notes:

1. The thunk is created by `(lambda () (interp ae d))`.
2. The thunk is evaluated by `((get-sub v d))`.