Today's Lecture Notes for cs173

sk and dbtucker

September 13, 2000

Purpose: To understand environments and lambda.

Quote of the Day: "Save the environment. Create a closure today."—Cormac Flanagan

Interpreting let-expressions

Now that we have substitution nailed down, we can write an interpreter for our language with let-expressions. Recall the type for `subst`:

```
;; subst : ALetExp symbol ALetExp -> ALetExp
```

where the first arg is the body into which we're subst'ing, the second is the variable to subst for, and the third is the expression to subst.

At this point we will also correct the return type of `calc`—it should return a `numE`, not a `number`. The reason for this is that we will need to substitute the result of a calculation into another expression. We will also assume we have the following functions:

```
;; numE+ : numE numE -> numE
;; numE* : numE numE -> numE
```

which add and multiply numbers, respectively.

Now we can write the interpreter:

```
;; calc : ALetExp -> numE
(define (calc a)
  (cases ALetExp a
    [numE (n) a]
    [addE (le re)
      (numE+ (calc le) (calc re))]
    [multE (le re)
      (numE* (calc le) (calc re))]
    [letE (var val body)
      (calc (subst body
                   var
                   (calc val)))]
    [varE (v) (error ...)]))
```

There are a couple of points to note here:

1. In the `letE` case, we evaluate `val` before substituting it into `body`. This is exactly the meaning of eager evaluation. If we change `(calc val)` to `val`, then we have are doing lazy evaluation[1].

2. Observe that the `varE` case always results in an error. In a closed expression, all variables will be substituted for before the evaluator gets to them. If we have a dangling `varE`, it was free in the expression, which is an error.

Variable capture

We still had one sticking point on substitution. Consider this expression:

```
(let (x (y + 1))
  (let (y 2)
    x))
```

Note that `y` is free in the above. If we use the lazy version of the interpreter above, under our definition of substitution this expression reduces to:

```
(let (y 2)
  (y + 1))
```

Suddenly `y` is bound! We refer to this situation as *variable capture*—the unbound variable `y` in the original expression has been captured (bound) in the new expression.

This is clearly not what we want, since the original expression should not have computed to a value, but now can reduce to the value `3`. The real problem is that we defined substitution in terms of string manipulation, and so these two `y`'s ended up being related though name, even though they refer to different bindings.

In general, our substitution breaks in the case:

```
(subst (letE var val body) svar sval)
```

when `var` occurs free in `sval`. Therefore we will make the assumption that our interpreter need only work correctly on closed expressions.

Environments

Consider the following expression:

```
(let (x 3)
  (let (y (x + 2))
    (let (z (x + y))
      z)))
```

[1] Assuming the expression we are evaluating is closed—see the next section.

How many times is the final variable `z` traversed? Each let-expression needs to test whether to substitute into it, so it will be traversed 3 times (once each for the `x`, `y`, and `z` bindings). For a sequence of n such let-expressions, the final variable will be traversed n times.

We would prefer to visit each variable only once. To do this, we will delay the substitutions performed at each let-expression and keep track of them. Whenever we visit a variable, we will then perform the substitution. Take the above example:

1. Before we start evaluating, there are no delayed substitutions: $\texttt{d} = \{\}$.
2. After the first let-expression, we now have one delayed substitution: $\texttt{d} = \{x \mapsto 3\}$.
3. After the second let-expression, $\texttt{d} = \{x \mapsto 3, y \mapsto 5\}$.
4. After the third let-expression, $\texttt{d} = \{x \mapsto 3, y \mapsto 5, z \mapsto 8\}$.

When we encounter a variable `v`, we replace it with `d(v)` for the current `d`.

Say the type for delayed substitutions is `DSub`. We need the following three functions:

```
;; fresh-sub : () -> DSub
;; new-sub : sym * numE * DSub -> Sub
;; get-sub : sym * DSub -> numE
```

Now we can write an interpreter using delayed substitutions:

```
;; interp : ALetExp * DSub -> numE
(define (interp a d)
  (cases ALetExp a
    [numE (n) a]
    [addE (le re)
      (numE+ (interp le d) (interp re d))]
    [multE (le re)
      (numE* (interp le d) (interp re d))]
    [letE (var val body)
      (interp body
              (new-sub var
                       (interp val d)
                       d))]
    [varE (v)
      (get-sub v d)]))
```

We define `(calc a)` to conform to the previous interface:

```
(define (calc a) (interp a (fresh-sub)))
```

The last question is how to implement the functions for `DSubs`. One way is to define the appropriate datatypes:

```
(define-datatype DSub DSub?
  [fresh-sub]
  [new-sub (var symbol?)
           (val ALetExp?)
           (rest DSub?)])

(define (get-sub var d)
  (cases DSub d
    [fresh-sub () (error "variable not bound")]
    [new-sub (v val rest)
       (cond
         [(symbol=? var v) val]
         [else (get-sub var rest)])]))
```

Another method is to represent `DSub` as a function from `symbol`s to `ALetExp`s:

```
(define (fresh-sub)
  (lambda (y)
    (error "variable not bound")))

(define (get-sub var d)
  (d var))

(define (new-sub var val d)
  (lambda (v)
    (cond [(symbol=? v var) val]
          [else (d v)])))
```