Today's Lecture Notes for cs173

sk and dbtucker

September 11, 2000

Purpose: To understand let and substitution.

Quote of the Day: "I am not a number, I am a free variable."—anonymous

Adding let-expressions

To add let-expressions to our language, we need to add two new variants, `letE` and `varE`:

```
(define-datatype ALetExp ALetExp?
  [numE (n number?)]
  [addE (lhs ALetExp?)
        (rhs ALetExp?)]
  [multE (lhs ALetExp?)
         (rhs ALetExp?)]
  [letE (var symbol?)
        (val ALetExp?)
        (body ALetExp?)]
  [varE (v symbol?)])
```

What should happen when we evaluate a let-expression? Intuitively, `(let (x 4) x)` should evaluate to 4, and `(let (x 4) (1 + x))` should be 5. We need to formulate a rule for let.

Substitution

Let's try to define substitution.

Attempt #1: Given `(let (var val) body)`, replace all instances of `var` in `body` with `val`:

- `(let (x 4) x)` $\longrightarrow$ `4`
- `(let (x 4) 3)` $\longrightarrow$ `3`
- `(let (x 3) (x + x))` $\longrightarrow$ `(3 + 3)`

- `(let (x 4) (x + (let (x 3) x)))` $\longrightarrow$ `(4 + (let (4 3) 4))`

The last expression is bogus—the expression `(let (4 3) 4)` doesn't make any sense because `3` is being bound to something that is not a variable. Note that this isn't even an `ALetExp` since `4` is not a symbol. Clearly we don't want to substitute into the variable binding position of a let-expression.

Attempt #2: Same as #1, but only perform the substitution on `varE`'s:

- `(let (x 4) (x + (let (x 3) x)))` $\longrightarrow$ `(4 + (let (x 3) 4))`

Now this expression is syntactically legal, but is it what we expect? Consider these two expressions:

- `(let (x 4) (let (x 3) x))`
- `(let (x 4) (let (y 3) y))`

We would expect these two expressions to evaluate to the same result, since renaming the innermost binding from `x` to `y` shouldn't affect the value. Under attempt #2, though, the former will give us 8 and the latter 7.

Where did we go wrong? Well, we expect the innermost `x` to be bound by the nearest enclosing let-expression, namely `(let (x 3) x)`. So, we shouldn't be allowed to substitute another value for `x` into that let-expression.

Attempt #3: Same as #2, except that when we come to a let-expression, we don't substitute when the binding variable of the let-expression is the same as the variable we are substituting for. That's a bit wordy, so hopefully the definition of `subst` will be clearer. The first arg is the expression we're subst'ing into, the second arg is the variable we're subst'ing for, and the third arg is the value we're subst'ing for the variable:

```
;; subst : ALetExp symbol ALetExp -> ALetExp
(define (subst sbody svar sval)
  (cases ALetExp sbody
    [numE (n) sbody]
    [addE (le re) (addE (subst le svar sval)
                        (subst re svar sval))]
    [multE (le re) (multE (subst le svar sval)
                          (subst le svar sval))]
    [varE (v)
      (cond
        [(symbol=? v svar) sval]
        [else sbody])]
    [letE (lvar lval lbody)
      (cond
        [(symbol=? lvar svar) sbody]
        [else (letE lvar lval (subst lbody svar sval))])]))
```

This is almost right now. The only problem is that expressions like these don't work:

- `(let (x 3) (let (y (x + 1)) y))` $\longrightarrow$ `(let (y (x + 1)) y)`
- `(let (x 3) (let (x (x + 1)) x))` $\longrightarrow$ `(let (x (x + 1)) x)`

In both cases we forgot to substitute into the rhs of the let-expression binding, so now our expressions aren't closed. This is easy to fix.

Attempt #4: Extend #3 by substituting into the the rhs of the let-expression:

```
;; subst : ALetExp symbol ALetExp -> ALetExp
(define (subst sbody svar sval)
  (cases ALetExp sbody
    [numE (n) sbody]
    [addE (le re) (addE (subst le svar sval)
                        (subst re svar sval))]
    [multE (le re) (multE (subst le svar sval)
                          (subst le svar sval))]
    [varE (v)
      (cond
        [(symbol=? v svar) sval]
        [else sbody])]
    [letE (lvar lval lbody)
      (cond
        [(symbol=? lvar svar) (letE lvar
                                    (subst lval svar sval)
                                    lbody)]
        [else (letE lvar
                    (subst lval svar sval)
                    (subst lbody svar sval))])]))
```

Phew! Finally we have it right. Or do we? This definition works fine if the value we're substituting is closed. But what happens if we have an expression like this?

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

This is left as an exercise to the reader. Try not to hurt yourself thinking about it.

Evaluation strategies

One final note on performing substitution. Given this expression:

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

you can evaluate `(1 + 2)` first, then substitute:

```
--> (let (x 3) (x + x))
--> (3 + 3)
--> 6
```

This is called *eager evaluation*, or *call-by-value*.

Alternatively, you can substitute first, then evaluate:

```
--> ((1 + 2) + (1 + 2))
--> (3 + (1 + 2))
--> (3 + 3)
--> 6
```

This is *lazy evaluation*, also called *call-by-name* or *call-by-need*.

In this course we will always use call-by-value, since almost every real programming language uses this strategy.