Lecture 19

Lectured by Prof. Caldwell and scribed by Sunil Kothari

October 30, 2008

1 Review - Use of the variables in the let construct

Q: What is the use of variables - seems like extra computation ?
A: consider the following example $let\ x = (z * y) + (3 * y) + (4 * y)$ in $x + x + y$

The evaluation of a let construct is given as:
$eval\ m\ (Let\ x\ e1\ e2) = eval\ m'\ e2$
where $m'\ z =$ if $z == x$ then $eval\ m\ e1$ else $m\ z$

Recall the datatype for the expression language:

```
data Exp= N Int | V String | Add Exp Exp |  Let String Exp Exp
```

Suppose $m\ z = 100$, then what is eval m (Let "x" (N 1) (Add (V "x")(V "x")))
?

$$eval\ m\ (Let\ "x"\ (N\ 1)\ (Add\ (V\ "x")(V\ "x"))) \rightsquigarrow eval\ m'\ (Add(V"x")(V"x")))$$

$$\begin{aligned} \text{where } m'\ "x" &= N\ 1 \\ m'\ \_ &= 100 \end{aligned}$$

$\rightsquigarrow (eval\ m'\ (V\ "x")) + (eval\ m'\ (V\ "y"))$
$\rightsquigarrow (m'\ "x") + (m'\ "y")$
$\rightsquigarrow 1 + 100$
$\rightsquigarrow 101$

Suppose we have a function $f : a \rightarrow b$. We can make a function which behaves like f but differs on one of the inputs. The function *update* does this job
$update :: (a \rightarrow b) \rightarrow (a, b) \rightarrow (a \rightarrow b)$
$update\ f\ (x, y) = \backslash w \rightarrow$ if $w == x$ then y else $f\ x$

So, if $f\ x = x$ then, $g = update\ f\ (0, 1)$ is a function that behaves just like the identity function except that on input 0 it returns 1.

Let's do a computation with

$$\begin{aligned} g\,0 &= (update f(0,1))0 \\ &\leadsto (\lambda w \to \text{if } w == 0 \text{ then } 1 \text{ else } f\ w)0 \\ &\leadsto \text{if } 0 == 0 \text{ then } 1 \text{ else } f\ 0 \\ &\leadsto \text{if } true \text{ then } 1\text{else } f\ 0 \\ &\leadsto 1 \end{aligned}$$

$evalm(Add(V"x")(Let"x"(N2)(V"x"))$
...
$\leadsto 100 + 2$
$\leadsto 102$

This is same as $\forall x : Int.P(x)$ where x in $P(x)$ is a binding of x which is quantified at the start of the expression.

2 Capture avoiding substitutions

Consider the lambda terms given by the following datatype:

```
data Lam = V String | Ap Lam Lam | Fun String Lam deriving (Eq, Show)
```

$Ap(Fun"x"(V"x"))(V"y") \leadsto y$

How does this evaluation happens ?
$Ap(Fun"x"(V"x"))["x" := (V"y")]$
where $e1[x := y]$ replace all x by y in $e1$
$\leadsto V"y"$

```
Main> :t subst
subst :: ([Char],Lam) -> Lam -> Lam
Main> subst ("x", V "y") (V "x")
V "y"
Main> subst ("x", V "y") (V "z")
V "z"
Main> subst ("x", V "w") (Fun  "z" (Ap ( V "z") (V "x")))
Fun "z" (Ap (V "z") (V "w"))
Main>
```

The following functions are equal
$(\lambda x \to x) = (\lambda y \to y)$
$(\lambda x \to x\ y) = (\lambda z \to z\ y)$

But $(\lambda x \to x\ y)$ not equal to $(\lambda y \to y\ z)$ nor $(\lambda y \to y\ y)$.

```
Main> subst ("x", V "w") (Fun  "x"  (V "x"))
Fun "x" (V "w")
```

Look what happened !!!
$(\lambda x \rightarrow x)[x := w] \rightsquigarrow (\lambda x \rightarrow w)$

In the body of the lambda x is getting replaced by w even though x is bound.

In capture avoiding substitutions, we want to substitute only free varaibles.

Here's an example :
$(\lambda x \rightarrow x\ y)[y := x\ z]$
$\rightsquigarrow \lambda x \rightarrow x\ (x\ z)$

As mentioned earlier, $(\lambda x \rightarrow x) = (\lambda y \rightarrow y)$
In general, $(\lambda x \rightarrow m) = (\lambda z \rightarrow m[x := z])$ and $z \in FV(m)$
So,

$$\begin{aligned} (\lambda x \rightarrow x\ y) &= (\lambda z \rightarrow z\ y) \\ &= (\lambda w \rightarrow w\ y) \end{aligned}$$

So we will define this notion of *free variables* (fv) of a term.

$$\begin{aligned} fv\ (V\ s) &= [s] \\ fv\ (Ap\ m\ n) &= fv\ m + +fv\ n \\ fv\ (Fun\ s\ m) &= filter\ (/ = s)\ (fv\ m) \end{aligned}$$

Once we have this we can use this to avoid capturing bound variables in the *subst* function and is defined as:

```
subst (x,n) (V s) = if x == s then n else (V s)
subst (x,n) (Ap m k ) = Ap (subst (x,n) m) (subst (x,n) k)
subst (x,n) (Fun y m) =
   if x == y
   then Fun y m
   else if y `elem` (fv n)
    then Fun z  (subst (x,n) (subst (y, V z) m))
    else Fun y (subst (x,n) m)
  where  z = fresh "z" vars
               where vars = x:y:((fv m) ++ (fv n)) -- what we need is a total
```

And a show function for the lambda terms as:

```
instance Show Lam where
  show (V x) = x
  show (Ap m n) = "(" ++ show m ++ ")(" ++ show n ++ ")"
  show (Fun x m) = "
" ++ x ++"->"++ show m
```

Another helper function is *test_subst*, which pretty prints the substitution.

```
test_subst (x,n) t = show t ++ " ---> " ++ show (subst (x,n) t)
```

Now we can test our substitution function:

```
Main> test_subst ("x", V "w") (Fun  "z" (Ap ( V "z") (V "x")))
"\\z->(z)(x) ---> \\z->(z)(w)"
Main> test_subst ("x", V "w") (Fun  "w" (Ap ( V "w") (V "x")))
"\\w->(w)(x) ---> \\z->(z)(w)"
Main> test_subst ("x", V "w") (Fun  "w" (Ap ( V "w") (V "z")))
"\\w->(w)(z) ---> \\zz->(zz)(z)"
Main>
```