Recitation - 04

Jahnavi - jp5867@nyu.edu

Lambda Calculus:

-> Introduced by the mathematician 'Alonzo Church' in the 1930s.

-> Lambda Calculus is a formal mathematical way to express computation.

-> It is based on function abstraction and application using variable binding and substitution.

-> It is a turing complete language.

The lambda calculus consists of a language of lambda terms, which is defined by a certain formal syntax, and a set of transformation rules, which allow manipulation of the lambda terms.

A valid lambda term:

- A variable, x , is itself a valid lambda term.
- If M is a lambda term, and x is a variable, then (λx.M) is a lambda term (called an abstraction).
- If M and N are lambda terms, then (MN) is a lambda term (called an application).

Precedence and Associativity:

Precedence: Application has higher precedence than lambda abstraction.

Associativity:

-> Function Applications are left associative - $x\ y\ z \Rightarrow (x\ y)\ z$

-> Abstractions are right associative - $\lambda x.x.\lambda y.y = \lambda x.(x.(\lambda y.y))$

Hence this expression - ʌx.ʌy.ʌz.x z (y z) is written as - (ʌx.(ʌy. (ʌz.((x z) (y z)))))

Free and Bound variables:

Bound Variable: a variable that is associated with some lambda.

Free Variable: a var that is *not* associated with any lambda.

1. In the expression x, variable x is free (no variable is bound).
2. In the expression λx.M, every x in M is bound; every variable other than x that is free in M is free in λx.M; every variable that is bound in M is bound in λx.M.
1. In the expression MN:
    1. The free variables of MN are the union of two sets: the free variables of M, and the free variables of N.
    2. The bound variables of MN are also the union of two sets: the bound variables of M and the bound variables of N.
    3. Free(λx. E) = Free(E) - { x }

Note that a variable may occur more than once in some lambda expression; some occurrences may be free and some may be bound, so the variable itself is *both* free and bound in the expression, but each individual *occurrence* is either free or bound (not both).

Free and Bound variables:

1. (ʎx.y)(ʎy.yx) : First y - free; Second y - bound; x - free   (The variables next to ʎ are bound to ʎ)

2. FV[ʎx.ʎy.((ʎz.ʎv.z(zv))(xy)(zu))] = {z,u}

2. Free(ʎx. x (ʎy. x y z)) = {z}

3. Free((ʎv.ʎy.v ʎu.ʎv.y) ʎu.ʎy.u) = {y}

4. Free((ʎu (ʎv. v u) ʎv. (y v)) = {y}

5. Free(ʎv.ʎy.v (ʎx.x (u y))) = {u,y}

6. ʎx. (x ʎy.x)   ʎy.(z (y ʎx.x)) - Bound

Reduction:

α - conversion: $\lambda x.M \rightarrow \lambda y.([y/x]M)$, if $y \notin FV(M)$.

-> (ʌx.x) is the same as (ʌy.y)

-> (ʌx.(x*x)) is the same as (ʌu.(u*u))

-> All we have done is change the parameter name (bound variable) next to the ʌ as well as in the body of the function.

-> Renaming the bound variable does not change the abstraction.

-> Formally, (ʌx.M) = $_{\alpha}$ (ʌy.M{x $\leftarrow$ y}) where

y is a "brand new" variable not appearing in M, and

M{x y} is M with all occurrences of x replaced by y.

Substitution:

$\lambda x. (x\ y))\ [y = 5] = (\lambda x. (x\ 5))$

$(\lambda x. (x\ y))\ [y = (u\ v)] = (\lambda x. (x\ (u\ v)))$

Substitution must be done carefully so as not to alter the meaning of the λ-term!

$(\lambda x. (x\ y))\ [y = x]\ != (\lambda x. (x\ x))$

As can be seen, y was a free-variable before, but after the substitution y's value has become bound! Such a case is called a "capture" case.

$(\lambda x. (x\ y))\ [y = x] => (\lambda x'. (x'\ y))\ [y = x] = (\lambda x'. (x'\ x))$

Another "capture" example: $(\lambda x. (y\ x))\ [y = (\lambda z.(x\ z))]\ != (\lambda x. ((\lambda z.(x\ z))\ x))\ (\lambda x. (y\ x))$

$[y = (\lambda z.(x\ z))] = (\lambda x'. (y\ x'))\ [y = (\lambda z.(x\ z))] = (\lambda x'. ((\lambda z.(x\ z))\ x'))$

Substitution:

1. $x\ [x = P] = P$
2. $y\ [x = P] = y$ if $x\ != y$
3. $(M\ N)\ [x = P] = (M[x = P]\ N[x = P])$
4. $(\lambda x.M)\ [x = P] = (\lambda x.M)$
5. $(\lambda y.M)\ [x = P] = (\lambda y.M[x = P])$ if $x!=y$ and y not belongs to $FV[P]$
6. $(\lambda y.M)\ [x = P] = (\lambda y'.(M\{y\ y'\}[x = P]))$ if $x\ y$ and y belongs to $FV[P]$ and y' is brand new

Substitution example:

$(\lambda y.\ (((\lambda x.\ x)\ y)\ x))\ [x = (y\ (\lambda x.\ x))]$

$= (\lambda y'.\ (((\lambda x.\ x)\ y')\ x))\ [x = (y\ (\lambda x.\ x))]$

$= (\lambda y'.\ (((\lambda x.\ x)[x = (y\ (\lambda x.\ x))]\ y'[x = (y\ (\lambda x.\ x))])\ x[x = (y\ (\lambda x.\ x))]))$

$= (\lambda y'.\ (((\lambda x.\ x)\ y')\ (y\ (\lambda x.\ x))))$

B - reduction:

->The process of evaluating lambda terms by “plugging arguments into functions” is called β-reduction.

->A term of the form (λx.M)N, which consists of a lambda abstraction applied to another term, is called a β-redex.

->We say that it reduces to M[N/x], and we call the latter term the reduct.

->We reduce lambda terms by finding a subterm that is a redex, and then replacing that redex by its reduct.

->We repeat this as many times as we like, or until there are no more redexes left to reduce.

->A lambda term without any β-redexes is said to be in β-normal form.

B reduction:

Example:

1. (ʌx.y)((ʌz.zz)(ʌw.w)) →β→ (ʌx.y)((ʌw.w)(ʌw.w)) →β→ (ʌx.y)(ʌw.w) →β→ y.
2. (ʌz.ʌu.ʌx.x (x ((v v) ʌu.u))) -> ʌu.ʌx.x
3. (ʌz.(z (z ʌy.z)) w) -> (w (w ʌy.w))

*The order of applying β-reductions is not significant. The end result is the same, especially if it terminates.

Not every term evaluates to something; some terms can be reduced forever without reaching a normal form. The following is an example:

(ʌx.xx)(ʌy.yyy) →β (ʌy.yyy)(ʌy.yyy) →β (ʌy.yyy)(ʌy.yyy)(ʌy.yyy) →β . . .

Evaluation strategy:

- **Normal order**: reduce the outermost "redex" first.

$(\lambda x.(\lambda y.xy))((\lambda x.x)z) = \lambda y.((\lambda x.x)z)y = \lambda y.zy$

- **Applicative order**: arguments to a function application are evaluated first, from left to right before the function application itself is evaluated.

$(\lambda x.(\lambda y.xy))((\lambda x.x)z) = (\lambda x.(\lambda y.xy))z = \lambda y.zy$

-> An expression that can't be β-reduced any further is a normal form.

-> Not everything has a normal form.

-> If a lambda reduction terminates, it terminates to the same reduced expression regardless of reduction order.

-> If a terminating lambda reduction exists, normal order evaluation will terminate.

The number of β-reductions performed in the evaluation of this expression are not same with the applicative order strategy or the normal order strategy is used.

Example:

iszero = (λ n. n (λ x. false) true)

0 = (λ s z. z)

1 = (λ s z. s z)

true=(λxy.x)

false=(λxy.y)

Question: How do we compute iszero 1 to get false via beta reduction?

```
    iszero 1                              #|Evaluate by normal order|#
=> iszero 1                               ; by def of iszero
=> (λ n. n (λ x. false) true) 1           ; do one step reduction for λ n
=> 1 (λ x. false) true                    ; by def of 1
=> (λ s z. s z) (λ x. false) true         ; application are left associative
=> ((λ s z. s z) (λ x. false)) true       ; do one step reduction for λ s
=> (λ z. (λ x. false) z) true             ; do one step reduction for λ z
=> (λ x. false) true                      ; do one step reduction for λ x
=> false
```

Numbers:

$0 :\Leftrightarrow \lambda sz.z$
$1 = \lambda sz.s(z)$
$2 = \lambda sz.s(s(z))$
$3 = \lambda sz.s(s(s(z)))$
$4 = \lambda sz.s(s(s(s(z))))$
$S :\Leftrightarrow \lambda abc.b(abc)$

Let us calculate the successor of 0 with it:

$S0 = (\lambda abc.b(abc))\ (\lambda sz.z)$
$\quad = \lambda bc.b((\lambda sz.z)\ bc)$
$\quad = \lambda bc.b((\lambda z.z)\ c)$
$\quad = \lambda bc.b(c)$

$\lambda bc.b(c) = \lambda sz.s(z) = 1$

Β-reduction with α-conversion

(λxyz.xyz)(λx.xx)(λx.x)x

= (((λxyz.xyz)(λx.xx))(λx.x))x

= ((**(λxyz.xyz)(λx.xx)**)(λx.x))x

(λxyz.xyz)(λx.xx)

= (λx.λyz.xyz)(λx.xx)

= (λx.λyz.xyz)(λx'.x'x')

= (λyz.xyz)[x := λx'.x'x']

= (λyz.(λx'.x'x')yz)

= (λyz. (**(λx'.x'x')y)** z)

= (λyz. **((x'x')[x' := y])** z)

= (λyz. **(yy)** z)

Add this back into the original expression:

((**(λxyz.xyz)(λx.xx)**)(λx.x))x

= ((λyz.(yy)z)(λx.x))x

= (**(λyz.(yy)z)(λx.x)**)x

(λyz.(yy)z)(λx.x)

= (λy.λz.(yy)z)(λx.x)

= (λz.(yy)z)[y := (λx.x)]

Example Contd...

= $(\lambda z.(yy)z)[y := (\lambda x.x)]$

= $(\lambda z.(\mathbf{(\lambda x.x)(\lambda x.x)})z)$

= $(\lambda z.(\mathbf{(x)[x := \lambda x.x]})z)$

= $(\lambda z.(\mathbf{(\lambda x.x)})z)$

= $(\lambda z.\mathbf{(\lambda x.x)z})$

= $(\lambda z.(x)[x:=z])$

= $(\lambda z.(z))$

= $(\lambda z.z)$

Put it back into the main expression:

$(\mathbf{(\lambda yz.(yy)z)(\lambda x.x)})x$

= $(\mathbf{(\lambda z.z)})x$

= x

Recursion:

FACT = ʎn. if n = 0 then 1 else n × FACT (n – 1)

Remove recursion: Way 1

FACT' = ʎf. ʎn. if n = 0 then 1 else n × (f f (n – 1))

FACT = FACT' FACT'

FACT 3 = (FACT' FACT') 3 Definition of FACT

= ((ʎf. ʎn. if n = 0 then 1 else n × (f f (n – 1))) FACT0) 3

= (ʎn. if n = 0 then 1 else n × (FACT0 FACT0 (n – 1))) 3

= if 3 = 0 then 1 else 3 × (FACT0 FACT0 (3 – 1))

= 3 × (FACT0 FACT0 (3 – 1))

= .....

= 3 × 2 × 1 × 1 = 6

Fixed point combinator:

$FIX(f) \equiv f(FIX(f))$

One such combinator is Y-combinator

$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

$YM = M(YM)$

$YM = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))M$

$\rightarrow (\lambda x.M(xx))(\lambda x.M(xx))$

$\rightarrow M((\lambda x.M(xx))(\lambda x.M(xx)))$

$M(YM) = M((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))M)$

$\rightarrow M((\lambda x.M(xx))(\lambda x.M(xx)))$

Fixed point combinator:

$G = \lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f\ (n - 1))$

$FACT = Y\ G$

$= \ (\ \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))\ )\ G$

$= \lambda x.G(x\ x))\ (\lambda x.G(x\ x))$

$= G(\ (\lambda x.G(x\ x))\ \ (\lambda x.G(x\ x))\ )$

$= (\ \lambda f.\ \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times (f\ (n - 1))\ \ )\ \ ((\lambda x.G(x\ x))\ \ (\lambda x.G(x\ x)))$

$= \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times (\ (\lambda x.G(x\ x))\ \ (\lambda x.G(x\ x))\ (n - 1))$

$= \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times (\ Y\ G\ (n - 1))$

$= \lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n \times (\ FACT\ (n - 1))$