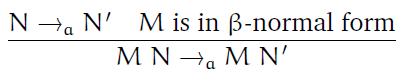
**Assignment 3 - Lambda Calculus**

**Name: Mingyuan Cui**

**StdentID: u5323288**

**1.** **Write down four rules for *strict* or *applicative*, left-to-right order evaluation of λ-calculus terms in the style of the rules given for normal order evaluation in the lectures. In other words, define a binary relation →a that ensures that all functions and arguments are evaluated before substitutions are performed. Here’s one rule for free:**



**This rule guarantees that evaluation moves from left to right because**

**N is not allowed to be evaluated before M has been reduced to normal**

**form. You need to write three more rules. [5 marks]**

**Answer:**

**2. Prove the following “inequalities” by giving a derivation of λ|-X = Y for arbitrary terms X and Y when the inequality is assumed to be an equation:**

**(a) K I # K [2 marks]**

**(b) x # y (with x and y distinct \_-calculus variables) [3 marks]**

**Answer:**

**(a) Assume K I=K**

Then K I I=K I

Then I=K I (by behaviour of K)

Then I=K (by assumption K I=K)

Then I X Y=K X Y (for arbitrary X, Y)

Then X Y=X (by behaviour of I and K)

Take X to be I.

Then Y=I

Then X=Y

Thus, K I # K

**(b) Assume x=y**

Then K= λxy.x= λxy.y (by assumption x=y)

Then K X Y= (λxy.x)X Y= (λy.X) Y =X (for arbitrary X, Y)

And K X Y= (λxy.y)X Y= (λy.y) Y =Y

Then X=Y

Thus, x # y

**3. Define the λ-term corresponding to the following recursive function f *without* using the Y (or any other recursion) combinator:**

**f(0) = 3**

**f(n + 1) = 2 × f(n) + 3**

**Use primitive recursion and the λ-terms corresponding to 2, 3, + and ×. *Don’t* solve the recurrence relation and define the function with a closed form using exponentiation. [5 marks]**

**Answer:**

we define the function f first,

assume that

f(0)=Pr<f`, g>(0, y)=f`(y)=3;

f(n)= Pr<f`, g>(n, y);

then,

f(n+1)=Pr<f`, g>(n+1, y)

=g(n, Pr<f`, g>(n, y), y)

=2 × f(n) + 3

=2× Pr<f`, g>(n, y)+3;

∴f`: 3

g: 2 ×+ 3= + (× 2 ) 3

Then we define the +, ×, 2, 3 and Suc,

Suc(λn. (λf x. f (n f x)))

+λx y.x Suc y

3 λf x.f (f (f x))

2 λf x. f (f x)))

× λx y. x ( + y) 0

Notation: My strategy of × operation was inspired by the primitive recursion of addition operation in the lecture.

**4. Again, using primitive recursion, define a function on lists that adds 1 to each element of a list. Thus:**

**f([ ]) = [ ]**

**f(h :: t) = (h + 1) :: f(t)**

**Recall that h :: t is the list consisting of the element h followed by the list t (a “cons” cell). [5 marks]**

**Answer:**

Assume that

f([ ]) = Pr<f`, g>([ ], y)=f`(y)=[ ]=nil;

f(h::t)

= Pr<f`, g>( l, t, h::t)

= g(l-1, Pr<f`, g>( l-1, y), t, h::t)

Notation: l is the length of the list, t is an index in the list and is the first argument in function Key(t, y), h::t is the list and is the second argument of function Key(t, y)

∴f`: nil

g : consKey cons

cons(λh t c n. c h (t c n))

nilλc n.n

Suc (λn. (λf x. f (n f x)))

+(λx y.x Suc y)

H (the head of list y)

(the elements except the head in list y)

Key (λx y. H (Pred(x) T y) (the value of mx from List y [m1,m2,m3…mn] )

Notation: My strategy of Key function was inspired by the of addition operation defined in the lecture. And H and T function was defined to help with Key function.