**PART 1**

Per definir el tipus LT hem utilitzat tres constructors:

type Var = String

data LT  = Va Var | La Var LT | AP LT LT

Va per crear variables, La per crear lambda-abstraccions i AP per crear aplicacions.

Per crear el mètode subst hem utilitzat la funció freeAndBoundVars que retorna una tupla on el primer element és la llista de variables lliures i el segon element les variables lligades. També hem necessitat una llista de possibles variables amb lletres de la “a” a la “z”. Està definida a l’inici com a possible\_vars. Quan s’ha de canviar el nom d’una variable per l’altre s’ha d’agafar la primera variable lliure de la llista, per això hem creat el mètode firstNonBoundVar. Quan es substitueix un nom de variable per una altra, aquella deixa d’estar lligada i la que està lligada és la substituta, per tant a la llista de variables lligades s’ha de substituir el nom d’aquella variable per la nova escollida, això ho fem amb replaceFirst.

possible\_vars = ["a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"]

Funció que retorna una tupla amb les variables lliures i lligades del lambda-terme donat.

Param 1: Lambda-terme a avaluar.

Retorna: Tupla (<freeVars>,<boundVars>) amb dos llistes, la primera representant les variables lliures de <param 1> i la segona representant les variables lligades.

freeAndBoundVars :: LT -> ([Var],[Var])

freeAndBoundVars (Va v) = ([v],[]) -- La variable s'afegeix a la llista de lliures

freeAndBoundVars (La v lt) = (

    delete v (fst (freeAndBoundVars lt)), -- S'elimina <v> de la llista de variables lliures perquè ara estarà lligada

    if v `elem` fst (freeAndBoundVars lt) -- Si <v> està a la llista de variables lliures de <lt>

    then (snd (freeAndBoundVars lt))++[v] -- Llavors s'afegeix a la llista de lligades

    else snd (freeAndBoundVars lt)) -- Sino es torna a cridar a la funció amb <lt>

freeAndBoundVars (AP a b) = (

    fst (freeAndBoundVars a) `union` fst (freeAndBoundVars b),

    snd (freeAndBoundVars a) `union` snd (freeAndBoundVars b)) -- S'uneixen els resultats d'aplicar la funció als dos lambda-termes

De la llista de possibles noms de variables, retorna la primera que no estigui lligada

Param 1: Llista de variables lligades

Retorna: Primera variable que estigui lliure, és a dir, que no estigui a la llista passada

firstNonBoundVar :: [Var] -> Var

firstNonBoundVar bVars = (possible\_vars \\ bVars)!!0 -- Es fa la diferència entre les llistes i s'agafa el primer element

Reemplaça la primera instància de <param 1> per <param 2> a la llista donada mirant d'esquerra a dreta.

Param 1: Paràmetre a buscar i substituir

Param 2: Nou paràmetre substitut de <param 1>

Param 3: Llista on realitzar la cerca i substitució

Retorna: La llista original però la primera instància de <param 1> substituida per <param 2>

replaceFirst :: (Eq a) => a -> a -> [a] -> [a]

replaceFirst \_ \_ [] = []

replaceFirst a x (b:bc) | a == b    = x:bc

                        | otherwise = b:(replaceFirst a x bc)

La funció subst és el següent:

Substitueix a un lambda-terme una variable lliure per el lambda-terme indicat sense realitzar caputra de variables.

Param 1: Lambda-terme on realitzar la substitució.

Param 2: Variable a substituir.

Param 3: Nou lamda-terme substitut.

Retorna: Retorna el lambda-terme <param 1> on s'han substituit les instàncies lliures de <param 2> per <param 3> sense realizar captura.

subst :: LT -> Var -> LT -> LT

subst x@(Va v) old\_val new\_val = iSubst x old\_val new\_val []

subst x@(La v lt) old\_val new\_val = iSubst x old\_val new\_val [v] -- Es crida a iSubst afegint la variable v com a lligada

subst (AP lt1 lt2) old\_val new\_val = (AP (iSubst lt1 old\_val new\_val []) (iSubst lt2 old\_val new\_val []))

Finalment també per al mètode subst hem necessitat una funció d’immersió anomenada iSubst on es passa un altre paràmetre amb una llista de les variables que hi ha lligades fins aquell moment:

Funció d'immersió de subst. Realitza el mateix però s'ha de passar un paràmetre adicional indicant quines són les variables lligades en el moment de cridar a la funció.

Param 1: Lambda-terme on realitzar la substitució.

Param 2: Variable a substituir.

Param 3: Nou lamda-terme substitut.

Param 4: Llista de variables que estan lligades en el moment de cridar la funció.

Retorna: Retorna el lambda-terme <param 1> on s'han substituit les instàncies lliures de <param 2> per <param 3> sense realitzar captura.

iSubst :: LT -> Var -> LT -> [Var] -> LT

iSubst (Va v) old\_val new\_val bVars =

    if (v == old\_val) && not (elem v bVars)

    then new\_val

    else Va v -- Si és la variable que es vol canviar i no està lligada, llavors substituir per el nou valor

iSubst x@(La v lt) old\_val new\_val bVars =

    if elem v (fst (freeAndBoundVars new\_val)) -- Si la variable que la abstracció está lligant está dins de les variables lliures del nou lamda-terme

    then iSubst (La fnbv (subst lt v (Va fnbv))) old\_val new\_val nbVars -- Llavors s'ha de fer una alpha-conversió canvinat les <v> per un altre nom per evitar la captura

    else (La v (iSubst lt old\_val new\_val (v:bVars))) -- Sinó es pot cridar altre cop a iSubst amb el <lt> i afegint <v> a les variables lligades

    where

        fnbv = firstNonBoundVar (bVars++(fst (freeAndBoundVars new\_val))) -- Primera variable lliure que no estigui dins els variables lliures del nou valor, es fa aixi perquè sino dona problemes de recursivitat

        nbVars = replaceFirst v fnbv bVars -- Les noves variables lligades canviant la que estaba abans lligada per la nova

iSubst (AP lt1 lt2) old\_val new\_val bVars = (AP (iSubst lt1 old\_val new\_val bVars) (iSubst lt2 old\_val new\_val bVars))

En el cas de la funció esta\_normal només es retornarà False si hi ha un redex en el lambda-terme. El seu tipus és el següent:

Indica si un lambda-terme està en forma normal o no.

Param 1: El lambda-terme a comprovar.

Retorna: Cert si està en forma normal, fals altrament.

esta\_normal :: LT -> Bool

esta\_normal (Va v) = True

esta\_normal (La v lt) = esta\_normal lt

esta\_normal (AP (La v lt) \_) = False

esta\_normal (AP a b) = esta\_normal a && esta\_normal b

Tipus i definició de beta\_redueix:

Funció que rep un lambda-terme que sigui un redex i el resol.

Si el lambda-terme passat no és un redex, retorna el mateix lambda-terme.

Param 1: Lambda-terme sobre el que realitzar la reducció.

Retorna: El lambda-terme <param 1> amb la substitució feta, si no és un redex retorna el mateix lambda-terme.

beta\_redueix :: LT -> LT

beta\_redueix x@(Va v) = x

beta\_redueix x@(La v lt) = x

beta\_redueix (AP (La v lt1) lt2) = subst lt1 v lt2

beta\_redueix x@(AP lt1 lt2) = x

Per la funció redueix\_un\_n també s’ha utilitzat una funció d’immersió iRedueix\_un\_n on es passa el mateix lambda-terme però et retorna una tupla on el primer element indica si s’ha realitzat alguna beta-reducció en el lamda-terme passat, així si s’ha realitzat una reducció a la part esquerra, no fa falta mirar la part dreta.

Funció on es passa un lambda-terme i s'aplica la primera beta-reducció en ordre normal.

Si el lambda-terme passat no té cap redex, retorna el mateix lambda-terme.

Param 1: Lambda-terme sobre el que realitzar la reducció.

Retorna: El lambda-terme <param 1> amb la primera beta-reducció en ordre normal feta, si no és un redex retorna el mateix lambda-terme.

redueix\_un\_n :: LT -> LT

redueix\_un\_n x@(Va v) = x

redueix\_un\_n x@(La v lt) = La v (snd (iRedueix\_un\_n lt))

redueix\_un\_n x@(AP (La v lt1) lt2) = beta\_redueix x

redueix\_un\_n x@(AP lt1 lt2) =

    if (fst new\_lt1)

    then AP (snd new\_lt1) (lt2)

    else AP (snd new\_lt1) (snd new\_lt2)

    where

        new\_lt1 = iRedueix\_un\_n lt1

        new\_lt2 = iRedueix\_un\_n lt2

Funció d'immersió de redueix\_un\_n on es retorna un paràmetre addicional indicant si s'ha realitzat alguna beta-reducció o no.

Param 1: Lambda-terme sobre el que realitzar la reducció

Retorna: Tupla on el primer valor indica si s'ha realitzat una beta-reducció i el segon valor és el lambda-terme resultant d'aplicar la funció

iRedueix\_un\_n :: LT -> (Bool, LT)

iRedueix\_un\_n x@(Va v) = (False, x) -- S'indica que no s'ha realitzat cap beta-reducció

iRedueix\_un\_n x@(La v lt) = (fst red, La v (snd red))

    where

        red = iRedueix\_un\_n lt

iRedueix\_un\_n x@(AP (La v lt1) lt2) = (True, beta\_redueix x) -- S'indica que ja s'ha realitzar una beta-reducció

iRedueix\_un\_n x@(AP lt1 lt2) =

    if (fst new\_lt1) -- Si ja s'ha realitzat una beta-reducció a l'esquerra no fa falta fer-ho a la dreta

    then (fst new\_lt1, (AP (snd new\_lt1) (lt2)))

    else (fst new\_lt2, (AP (snd new\_lt1) (snd new\_lt2))) -- Si no s'ha fet cap beta-reducció, llavors s'intenta fer a la dreta

    where

        new\_lt1 = iRedueix\_un\_n lt1

        new\_lt2 = iRedueix\_un\_n lt2

Funció on es passa un lambda-terme i s'aplica la primera beta-reducció en ordre aplicatiu.

Si el lambda-terme passat no té cap redex, retorna el mateix lambda-terme.

Param 1: Lambda-terme sobre el que realitzar la reducció.

Retorna: El lambda-terme <param 1> amb la primera beta-reducció en ordre aplicatiu feta, si no és un redex retorna el mateix lambda-terme.

redueix\_un\_a :: LT -> LT

redueix\_un\_a :: LT -> LT

redueix\_un\_a x@(Va v) = x

redueix\_un\_a x@(La v lt) | not (esta\_normal lt) = La v (redueix\_un\_a lt)

                         | otherwise = x

redueix\_un\_a x@(AP y@(La v lt1) lt2) = redueix\_un\_n $

    (AP  (

            if not(esta\_normal y)

            then (redueix\_un\_a y)

            else y

        )

        (

            if not(esta\_normal lt2)

            then (redueix\_un\_a lt2)

            else lt2

        )

    )

redueix\_un\_a x@(AP lt1 lt2) = AP (redueix\_un\_n lt1) (redueix\_un\_n lt2)

Per l\_normalitza\_n el que s’ha fet és mirar si ja està en forma normal i si no ho està aplicar un pas de reducció.

Normalitza un lambda-terme, retorna la llista de passes fetes fins arribar a la forma normal seguint l'ordre normal.

Param 1: Lambda-terme sobre el que buscar la forma normal.

Retorna: Llista de lambda-termes, seqüència de reduccions, des del lambda-terme original <param 1> fins el lambda-terme en forma normal.

l\_normalitza\_n :: LT -> [LT]

l\_normalitza\_n x@(Va v) = [x]

l\_normalitza\_n x = if esta\_normal x then [x] else [x] ++ l\_normalitza\_n (redueix\_un\_n x) -- S'agrupen els dos casos, s'ha de fer el mateix tant si és una abstracció com si és una aplicació

Normalitza un lambda-terme, retorna la llista de passes fetes fins arribar a la forma normal seguint l'ordre aplicatiu.

Param 1: Lambda-terme sobre el que buscar la forma normal.

Retorna: Llista de lambda-termes, seqüència de reduccions, des del lambda-terme original <param 1> fins el lambda-terme en forma normal.

l\_normalitza\_a :: LT -> [LT]

l\_normalitza\_a ap@(Va a) = [ap]

l\_normalitza\_a x = if esta\_normal x then [x] else [x] ++ l\_normalitza\_a (redueix\_un\_a x)

Per normalitza\_n i normalitza\_a el que s’ha fet és crear una funció d’ordre superior iNormalitza on es passa la funció que es vol que s’apliqui al lambda-terme, ja sigui redueix\_un\_n o redueix\_un\_a:

Normalitza un lambda-terme seguint l'ordre normal, retorna una tupla amb el nombre de passos necessàris per arribar a la forma normal i la forma normal del lambda-terme.

Param 1: Lambda-terme sobre el que buscar la forma normal.

Retorna: Tupla amb el primer valor com el nombre de passos a realitzar per arribar a la forma normal i el segon valor el lambda-terme en forma normal.

normalitza\_n :: LT -> (Integer, LT)

normalitza\_n x = iNormalitza redueix\_un\_n x 0

Normalitza un lambda-terme seguint l'ordre aplicatiu, retorna una tupla amb el nombre de passos necessàris per arribar a la forma normal i la forma normal del lambda-terme.

Param 1: Lambda-terme sobre el que buscar la forma normal.

Retorna: Tupla amb el primer valor com el nombre de passos a realitzar per arribar a la forma normal i el segon valor el lambda-terme en forma normal.

normalitza\_a :: LT -> (Integer, LT)

normalitza\_a x = iNormalitza redueix\_un\_a x 0

Immersió de la funció normalitza\_n i normalitza\_a.

Realitza la mateixa funció excepte que a aqueste se li passa dos paràmetres extres.

Param 1: Funció a aplicar sobre el <param 2>

Param 2: Lambda-terme sobre el que buscar la forma normal.

Param 3: Nombre de passes que s'han realitzat fins al moment de cridar la funció.

Retorna: El mateix que normalitza\_n i normalitza\_a

iNormalitza :: (LT -> LT) -> LT -> Integer -> (Integer, LT)

iNormalitza \_ x@(Va v) n = (n,x)

iNormalitza f x n = if esta\_normal x then (n,x) else iNormalitza f (f x) (n+1)

Les definicions del meta-llenguatge es poden trobar totes seguides a l’apartat “META LLENGUATGE” del fitxer adjunt. El que s’ha fet és convertir les definicions dels apunts a lambda-termes utilitzant els constructors esmentats al principi. Finalment s’ha definit el mètode fact per poder calcular el factorial d’un nombre. Hi ha definits els nombres del 0 al 5, tot i que amb el factorial només hem aconseguit calcular fins al 3 en 1 minut de temps. Es pot trobar un exemple de l’execució a l’apartat “TERMES PER PROVAR EL FUNCIONAMENT” del fitxer adjunt. Allà hi ha uns lambda-termes definits juntament amb unes sumes, productes i factorials.

Actualment no se’ns acudeix cap forma de realitzar les funcions de beta-reducció més eficients.

Per derivar LT d’Ord el que hem utilitzat és el total de variables, tant lliures com lligades, és a dir, un LT lt1 serà més gran que un altre lt2 si té, en total, més variables que lt2.

Les derivacions són les següents:

instance Show LT where

    show (Va x) = x

    show (La v lt) = "(\\"++v++". "++ show lt ++ ")"

    show (AP lt lv) = "("++ show lt++" "++ show lv ++ ")"

Aquí el que s’ha fet es aprofitar la definició d’igualtat de Bruijn així el mètode surt molt més simple i evitem la repetició de codi.

El que fem primer és passar els dos termes en format de Bruijn i llavors aplicar la igualtat.

instance Eq LT where

    (==) lt1 lt2 = (a\_deBruijn lt1) == (a\_deBruijn lt2)

-- Per ordenar es compten el nombre total de variables que hi ha, tant lliures com lligades

instance Ord LT where

    lt1 <= lt2 = (length (fst (freeAndBoundVars lt1))) + ((length (snd (freeAndBoundVars lt1)))) <= (length (fst (freeAndBoundVars lt2))) + ((length (snd (freeAndBoundVars lt2))))

**PART 2**

Per definir el tipus de dades LTdB també hem utilitzat tres constructors similar amb LT, l'única diferència és que aquest últim te menys paràmetres en un dels constructors, precisament el constructor de l'abstracció.

També difereixen en el tipus de paràmetres del constructor de les variables, hem passat de String a Int.

type Nombre = Int -- Tipus de les variables

type Context = [Var] -- Llista de variables per el Context

data LTdB = Nat Nombre | Ap LTdB LTdB | L LTdB -- Tipus de dades per representar els lambdes termes en format debruijn

Amb el format de Bruijn, l’operador d'igualtat resultat molt mes fàcil d'implementar degut a que ara no tenim el problema d’alfa equivalència, només cal fer una cerca, simplement és anar buscant diferències entre els dos termes, a la que trobem una diferència (variables ,constructors) ja podem concloure que són diferents.

Instancia d'igualitat <Eq> per comparar dues termes en format Debruijn

Param 1: Lambda Terme en format Debruijn <LTdB>

Param 2: Lambda Terme en format Debruijn <LTdB>

Retorna: True si les dues termes son iguals altrament Fals

instance Eq LTdB where

    (==) (Nat x) (Nat x') = x==x'

    (==) (L l1) (L l2) = l1 == l2

    (==) (Ap l1 l2) (Ap l1' l2') = l1==l1' && l2==l2'

    (==) \_  \_ = False

L’Instancia show també és similar a LT només canvia el format de la sortida.

Instància  <show> per poder mostrar per pantalla els termes en format de Bruijn

Param 1: Lambda Terme en format de Bruijn <LTdB>

Param 2: Lambda Terme en format de Bruijn <LTdB>

Retorna: Mostra per pantalla el terme amb el següent format ex --> \.0 equival a \x.x

instance Show LTdB where

    show (Nat x) = show x

    show (L lt) = "(\\."++ show lt ++ ")"

    show (Ap lt lv) = "("++ show lt++" "++ show lv ++ ")"

Funció que rep un <LT> i retorna aquest mateix terme pero en format Debruijn <LTdB>

Param 1: Lambda terme que volem transformar en format de Bruijn

Retorna: El lambda terme en forma de Bruijn <LTdB>

a\_deBruijn :: LT -> LTdB

a\_deBruijn lt = i\_deBruijn lt ["x","y","z","a","b","c","q","s","f","n"] -- li passem el lamda terme i la llista del context (mapeig de possibles noms de variables lliures amb un índex)

Funció inmersiva que rep un  LT i retorna aquest mateix terme pero en format Debruijn

Param 1: Lambda terme que volem transformar en format Debruijn

Param 2: Llista variables del context

Retorna: El lambda terme en forma Debruijn <LTdB>

i\_deBruijn :: LT -> Context -> LTdB

-- Busquem l’índex de la variable afegit anteriorment a l’abstracció en cas que la variable sigui lligada altrament (variable lliure ) agafem l’índex del context.

i\_deBruijn va@(Va x) xs  = Nat (index xs x)

i\_deBruijn la@(La x lt) xs = L (i\_deBruijn lt (x:xs))

i\_deBruijn ap@(AP a b) xs = Ap (i\_deBruijn a xs) (i\_deBruijn b xs)

Funció que rep un <LTdB> i retorna aquest mateix terme pero en format Debruijn <LT>

Param 1: Lambda terme que volem transformar en format LTdB

Retorna: El lambda terme en forma Debruijn <LT>

de\_deBruijn :: LTdB -> LT

de\_deBruijn ltd = i\_de\_deBruijn ltd ["x","y","z","a","b","c","q","s","f","n"]

Funció inmersiva que rep un  LTdB i retorna aquest mateix terme pero en format LT

Param 1: Lambda terme que volem transformar en format LT

Param 2: Llista variables del context (mapeig de possibles variables lliures amb un index )

Retorna: El lambda terme en forma Debruijn LT

i\_de\_deBruijn :: LTdB -> Context -> LT

i\_de\_deBruijn va@(Nat v) xs = Va (xs!!v) -- S’associa un nom amb v que correspon a l’integer del terme <va>

i\_de\_deBruijn la@(L lt) xs = La t' (i\_de\_deBruijn lt (t':xs)) -- Es crea un nou lambda terme utilitzant la variable t’

                            where

                              v = charToString ['a'..'z'] -- Generem possibles noms de variables

-- Triem un nom de variables tenint en compte el context (Que no sigui una variable lliure que hem mapejat anteriorment com a possible nom de variable lliure)

                              t' = last (reverse [x | x<-v, not (x `elem` xs)])

i\_de\_deBruijn ap@(Ap l1 l2) xs = AP (i\_de\_deBruijn l1 xs) (i\_de\_deBruijn l2 xs)

**PART 3**

**EXEMPLES D’EXECUCIÓ**

**normalitza\_n (AP (AP (suma) (un)) (dos))**

(6,(\f. (\x. (f (f (f x))))))

**normalitza\_n (AP (AP (prod) (quatre)) (cinc))**

(12,(\f. (\x. (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f (f x)))))))))))))))))))))))

**normalitza\_n (AP fact zero)**

(9,(\f. (\x. (f x))))

**normalitza\_n (AP fact tres)**

(3212,(\f. (\x. (f (f (f (f (f (f x)))))))))

**normalitza\_n (AP suc tres)**

(3,(\f. (\x. (f (f (f (f x)))))))

**normalitza\_n (AP (prec) (tres))**

(61,(\f. (\x. (f (f x)))))

**normalitza\_n (AP prec cinc)**

(97,(\f. (\x. (f (f (f (f x)))))))

**normalitza\_n (AP prec zero)**

(9,(\f. (\x. x)))

**lt1 = AP (AP (La "z" (Va "z")) (La "z" (AP (Va "z") (Va "z")))) (La "z" (AP (Va "z") (Va "q")))**

**lt1**

(((\z. z) (\z. (z z))) (\z. (z q)))

**l\_normalitza\_n lt1**

[(((\z. z) (\z. (z z))) (\z. (z q))),((\z. (z z)) (\z. (z q))),((\z. (z q)) (\z. (z q))),((\z. (z q)) q),(q q)]

**a\_deBruijn lt1**

(((\.0) (\.(0 0))) (\.(0 7)))

**de\_deBruijn (a\_deBruijn lt1)**

(((\d. d) (\d. (d d))) (\d. (d q)))

**a\_deBruijn (snd (normalitza\_n lt1))**

(6 6)

**normalitza\_a lt1**

(4,(q q))

**l\_normalitza\_a lt1**

[(((\z. z) (\z. (z z))) (\z. (z q))),((\z. (z z)) (\z. (z q))),((\z. (z q)) (\z. (z q))),((\z. (z q)) q),(q q)]