Universitat de Girona
Escola Politècnica Superior

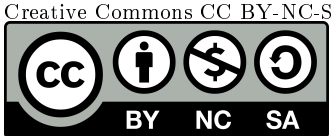
Paradigmes i llenguatges de programació

Curs 2024/25

GEINF

PAC4

Juan José Gómez Villegas, u1987338@campus.udg.edu
Guillem Pozo Sebastián, u1972840@campus.udg.edu



(28 de maig de 2025)

Índex

1 λ-càlcul clàssic

1.1 Tipus

El tipus de dades **LT** ens permet representar λ-termes, i la definició és exactament la gramàtica que diu el que és un λ-terme, és a dir, un λ-terme és: una variable, una aplicació i una abstracció, res més és un λ-terme.

```
data LT = Va String | Ap LT LT | Ab String LT

-- definim la forma de mostrar un lambda terme, els lambda termes es mostraran com (\x. (x y))
instance Show LT where
    show (Va a) = a
    show (Ap t1 t2) = "(" ++ show t1 ++ " " ++ show t2 ++ ")"
    show (Ab a t1) = "(\\" ++ a ++ ". " ++ show t1 ++ ")"

-- definim la idea d'equivalencia de dos lambda termes
instance Eq LT where
    (==) (Va a) (Va b) = a == b
    (==) (Ap t1 t2) (Ap t1' t2') = (&&) (t1 == t1') (t2 == t2')
    (==) (Ab _ t1) (Ab _ t2) = t1 == t2
    (==) _ _ = False
```

El **tipus LT** ha de ser instància de la classe **Show**, per tal que es pugui mostrar com nosaltres volguem, *com diu a l'enunciat*, i ha de ser instància de la classe **Eq**, per tal de definir la idea d'equivalència de dos λ-termes, és a dir, dos λ-termes seran el mateix quan, siguin dues abstraccions i independentment de la variable, els seus λ-termes interns siguin iguals, quan siguin dues aplicacions i aquestes siguin iguals, quan siguin dos variables iguals.

També hem definit el tipus de dades **Substitució**, que **ens ha permès implementar la funció subst** tal com diu la teoria, és a dir, **com un operador que ens permet reemplaçar ocurrències de variables per termes, evitant captura de variables lliures**. Per tant, el tipus **Substitució** serà una variable v que serà reemplaçada per un terme M', sobre un terme M, definit a la funció **subst**, tot junt representarà: $M[v->M']$.

```
data Substitucio = Sub String LT
```

1.2 Funcions auxiliars

Les següents funcions ens permeten implementar les funcions principals com funcions d'ordre superior.

La funció **es_redex** ens diu si un λ-terme té la forma d'un redex, o no.

```
-- es_redex, funcio que donat un LT, retorna True si es un redex, False altrament
es_redex :: LT -> Bool
es_redex (Ap (Ab _ _) _) = True
es_redex _ = False
```

I per buscar redex en un λ-terme T, la funció **conte_redex** ens diu si un terme qualsevol conté un redex a dins, **fent servir la funció es_redex**, i funciona de la manera següent: Una variable no conté cap redex (retorna fals, cas base), una abstracció conté un redex, si i només si, el terme que conté a dins **és un redex**, i una aplicació conté un redex, si qualsevol dels dos termes de dins **són o contenen un redex**. Aquesta funció retorna cert si conté un redex, amb això ja podem simplificar el codi de les funcions principals.

```
-- conte_redex, funcio que determina si un terme conte qualsevol redex (en qualsevol nivell,
    equivalent a fer esRedex sobre tot el terme)
conte_redex :: LT -> Bool
conte_redex (Va _) = False
conte_redex (Ab _ t) | es_redex t = True
                     | otherwise = conte_redex t
conte_redex (Ap t1 t2) | es_redex (Ap t1 t2) = True
                       | es_redex t1 = True
                       | es_redex t2 = True
                       | otherwise = conte_redex t1 || conte_redex t2
```

Les següents funcions ajuden a implementar la funció **subst**.

```
-- is_var, funcio que diu si un LT es una variable
is_var :: LT -> Bool
is_var (Va _) = True
is_var _ = False

-- get_var, donat una variable d'un lambda terme retorna la variable com un string
get_var :: LT -> String
get_var (Va a) = a
get_var (Ab a _) = a
get_var (Ap t1 _) = get_var t1
```

I les següents funcions ajuden a implementar la funció auxiliar que retorna les variables lliures i lligades d'un terme qualsevol. La funció **eliminar_duplicats**, donada una llista de variables, retorna la mateixa llista sense repetits, això soluciona un error trobar implementant la funció **freeAndboundVarsAux** en què les variables sortien repetides. I la funció **concat_tuples** s'utilitza en la mateixa funció, en el cas que el terme entrat sigui una aplicació.

```
-- eliminar_duplicats, funcio que elimina els elements duplicats d'una llista
eliminar_duplicats :: Eq a => [a] -> [a]
eliminar_duplicats = foldr (\x xs -> if x `elem` xs then xs else x:xs) []

-- concat_tuples, operador que concatena o intercalar dues llistes que son a dins d'una tupla
concat_tuples :: Eq a => ([a],[a]) -> ([a],[a]) -> ([a],[a])
concat_tuples t1 t2 = (eliminar_duplicats (fst t1 ++ fst t2),eliminar_duplicats (snd t1 ++ snd
    t2))
```

La següent funció a partir d'un λ-terme qualsevol, retorna una tupla amb dues llistes: primer una llista amb les variables lliures, segon una llista amb les variables lligades.

```
-- freeAndboundVarsAux, funcio que construeix una tupla amb dues llistes que continguin les
    variables lliures (first) i lligades (second)
freeAndboundVarsAux :: [String] -> [String] -> LT -> ([String],[String])
freeAndboundVarsAux freeVars boundVars (Va a) | a `elem` boundVars = (freeVars,boundVars)
                                              | otherwise = (a:freeVars,boundVars)
freeAndboundVarsAux freeVars boundVars (Ab a t1) = freeAndboundVarsAux freeVars (a:boundVars)
    t1
freeAndboundVarsAux freeVars boundVars (Ap t1 t2) = freeAndboundVarsAux freeVars boundVars t1
    `concat_tuples` freeAndboundVarsAux freeVars boundVars t2
```

1.3 Funcions principals

La funció **freeAndboundVars**, crida a la funció **freeAndboundVarsAux** com una funció d'ordre superior, inicialitzant dues llistes buides.

```
-- freeAndboundVars, donat un LT retorna una tupla amb una llista de freeVars i una llista de
    boundVars
freeAndboundVars :: LT -> ([String],[String])
freeAndboundVars = freeAndboundVarsAux [] []
```

La subst **freeAndboundVars**,

```
-- subst, donat un LT i una Substitucio, retorna el mateix LT al que se li ha aplicat la
    Substitucio
subst :: LT -> Substitucio -> LT
subst (Va a) (Sub v t') | a == v = t'
                        | otherwise = Va a
subst (Ap t1 t2) (Sub v t') = Ap (subst t1 (Sub v t')) (subst t2 (Sub v t'))
subst (Ab a t1) (Sub v t') | a == v = if is_var t' then Ab (get_var t') (subst t1 (Sub v t'))
    else Ab a t1
                           | a /= v && a `notElem` fst (freeAndboundVars t') = Ab a (subst t1
                               (Sub v t'))
                           | a /= v && a `elem` fst (freeAndboundVars t') = subst (alfa_conv
                               t1 t' a (a ++ "\'")) (Sub v t')
                           | otherwise = Ab a t1
    where alfa_conv t1 t' v' v'' | v'' `notElem` fst (freeAndboundVars t') && v'' `notElem`
        fst (freeAndboundVars t1) = Ab v'' (subst t1 (Sub v' (Va v'')))
                                   | otherwise = alfa_conv t1 t' v' (v'' ++ "\'")
```

```
-- esta_normal, diu si LT ja esta en forma normal
esta_normal :: LT -> Bool
esta_normal = not . conte_redex
```

```
-- beta_redueix, rep un LT que sigui un redex, i el resol
beta_redueix :: LT -> LT
beta_redueix (Ap (Ab a t1) t2) = subst t1 (Sub a t2)
beta_redueix t = t
```

```
-- redueix_un_n, rep un LT, i retorna el LT resultant d'aplicar la primera beta-reduccio
    segons l'ordre normal
redueix_un_n :: LT -> LT
redueix_un_n (Ap m n) | conte_redex (Ap m n) = beta_redueix (Ap m n)
                      | conte_redex m = Ap (redueix_un_n m) n
                      | conte_redex n = Ap m (redueix_un_n n)
                      | otherwise = Ap m n
redueix_un_n (Ab x t) = Ab x (redueix_un_n t)
```

```
-- redueix_un_a, rep un LT, i retorna el LT resultant d'aplicar la primera beta-reduccio
    segons l'ordre aplicatiu
redueix_un_a :: LT -> LT
redueix_un_a (Ap m n) | conte_redex m = Ap (redueix_un_a m) n
                      | conte_redex n = Ap m (redueix_un_a n)
                      | conte_redex (Ap m n) = beta_redueix (Ap m n)
                      | otherwise = Ap m n
redueix_un_a (Ab x t) = Ab x (redueix_un_a t)
```

```
-- l_normalitza_n, rep un LT, i retorna una llista de LT's que sigui una sequencia de beta-
    reduccions, segons l'ordre normal
l_normalitza_n :: LT -> [LT]
l_normalitza_n t | esta_normal t = t:[]
                 | otherwise = t:l_normalitza_n t'
    where t' = redueix_un_n t
```

```
-- l_normalitza_a, rep un LT, i retorna una llista de LT's que sigui una sequencia de beta-
    reduccions, segons l'ordre aplicatiu
l_normalitza_a :: LT -> [LT]
l_normalitza_a t | esta_normal t = t:[]
                 | otherwise = t:l_normalitza_a t'
    where t' = redueix_un_a t
```

```
-- normalitza_n, rep un LT, i retorna una tupla amb el nombre de passos, mes el LT en forma
    normal, seguint l'ordre normal
normalitza_n :: LT -> (Int,LT)
normalitza_n = normalitza_n_aux 0
```

```
-- normalitza_n_aux, funcio auxiliar cridada per normalitza_n
normalitza_n_aux :: Int -> LT -> (Int,LT)
normalitza_n_aux n t | esta_normal t = (n,t)
                     | otherwise = (n+1,snd (normalitza_n_aux (n+1) t'))
    where t' = redueix_un_n t
```

```
-- normalitza_a, rep un LT, i retorna una tupla amb el nombre de passos, mes el LT en forma
    normal, seguint l'ordre aplicatiu
normalitza_a :: LT -> (Int,LT)
```

2 Extra: Notació *de Bruijn*

2.1 Tipus

Per **la notació de Bruijn** hem definit un altre tipus de dades, que també és instància de les classes **Show** i **Eq**, igual que abans, l'únic que canvia en aquest tipus, és que ja no tenim variables en les λ-abstraccions, i les variables que abans eren cadenes de text, ara són enters, més concretament és la distància de cada variable en lambdes.

```
data LTdB = VadB Int | ApdB LTdB LTdB | AbdB LTdB

-- que tambe sera instancia de la classe Show i Eq
instance Show LTdB where
    show (VadB a) = show a
    show (ApdB t1 t2) = "(" ++ show t1 ++ "␣" ++ show t2 ++ ")"
    show (AbdB t1) = "(\\" ++ "._" ++ show t1 ++ ")"

-- definim la mateixa idea d'equivalencia que teniem en els lambda termes pels lambda termes
    amb notacio de Bruijn
instance Eq LTdB where
    (==) (VadB a) (VadB b) = a == b
    (==) (ApdB t1 t2) (ApdB t1' t2') = (&&) (t1 == t1') (t2 == t2')
    (==) (AbdB t1) (AbdB t2) = t1 == t2
    (==) _ _ = False
```

També hem hagut de definir un Context.

```
type Context = String
```

2.2 Funcions auxiliars

2.3 Funcions principals