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

(27 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.

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

-- definim la forma de mostrar un lambda terme
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
```

En el codi també es veu com hem fet per tal que el **tipus LT** sigui una instància de les classes **Show** i **Eq**, definint així com es mostraran els λ-termes, i que vol dir que dos λ-termes siguin equivalents.

També hem definit el tipus de dades **Substitució**, que ens ha permet implementar la funció substitució tal i com diu la teòria, 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 per un terme M', i la funció **subst** una **Substitució** sobre un terme M, tot junt representarà: $M[v- > M']$.

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

1.2 Funcions auxiliars

1.3 Funcions principals

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

-- 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
```

```
-- 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') | [x |x <- (fst (freeAndboundVars t')), y <- (snd (freeAndboundVars
     (Ab a t1))), x == y] == [] =
                                if a == v then Ab (get_var t') (subst t1 (Sub v t')) else Ab a
                                    (subst t1 (Sub v t'))
                           | otherwise = Ab a t1
```

```
-- 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
```

```haskell
-- 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)
```

```haskell
-- 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)
```

```haskell
-- 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]
```

```haskell
-- 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]
```

```haskell
-- 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)
```

```haskell
-- 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*

Per **la notació de Bruijn** hem definit un altre tipus de dades, que tambe es instancia de les classes **Show** i **Eq**.

```haskell
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
```