

# Haskell: Continuació

Albert Rubio

Llenguatges de Programació, FIB, UPC

# Continguts

- ①  $\lambda$ -termes a Haskell
- ② Currificació
- ③ Ordre superior
- ④ Llistes en Haskell
- ⑤ Lazy evaluation

# Continguts

- 1  $\lambda$ -termes a Haskell
- 2 Currificació
- 3 Ordre superior
- 4 Llistes en Haskell
- 5 Lazy evaluation

# $\lambda$ -termes a Haskell

Sintaxi:

```
(\x -> (x+3))
```

Semàntica:

```
(\x -> (x+3)) 4
```

Avaluació per beta-reducció.

Permet crear funcions i també passar resultats via un “binder”.

```
mes2 = \x -> (x+2)
```

```
plus = \x y -> (x+y)
```

També es pot escriure

```
mes2 = (+ 2)
```

```
plus = (+)
```

# Continguts

- ①  $\lambda$ -termes a Haskell
- ② Currificació
- ③ Ordre superior
- ④ Llistes en Haskell
- ⑤ Lazy evaluation

## curry/uncurry

- Les funcions “standard” tenen un nombre fix de paràmetres i un resultat d'un tipus fixat.
- Les funcions “currificades” tenen un nombre de paràmetres variable i un resultat d'un tipus que varia segons el nombre de paràmetres aplicats.

Les funcions en Haskell són, per defecte, currificades.

Per exemple:  $(+)$ ,  $(+3)$  o  $(2 + 3)$

```
curry          :: ((a,b) -> c) -> a -> b -> c
curry f x y    = f ( x , y )
```

```
uncurry        :: (a -> b -> c) -> (a,b) -> c
uncurry f ( p , s ) = f p s
```

# curry/uncurry

Noteu que en Haskell una funció “uncurried” és

- una funció amb un sol paràmetre.
- el tipus del paràmetre és tupla.

Per això, s'escriu  $f(2, True, [3, 5, 6])$ , ja que és  $f$  aplicat a la tupla  $(2, True, [3, 5, 6])$ .

# Continguts

- ① λ-terms a Haskell
- ② Currificació
- ③ Ordre superior**
- ④ Llistes en Haskell
- ⑤ Lazy evaluation



# Funcions d'ordre superior

Funcions que

- 1 prenen una funció com a argument
- 2 retornen una funció com a resultat

En Haskell totes les funcions poden ser vistes com a funcions d'ordre superior, ja que

```
mesk :: Int -> Int -> Int
```

```
mesk :: Int -> (Int -> Int)
```

són equivalents.

**Punt clau:** les funcions són objectes de primera classe.

# Composició y aplicació de funcions

- Per a composar funcions tenim l'operador '.'

Definició:

$$f \ . \ g \ = \ \backslash \ x \ -> \ f \ (g \ x)$$

Exemple:

```
dqsort = reverse . qsort
```

- Per aplicar funcions tenim l'operador '\$' Definició:

$$f \ \$ \ x \ = \ f \ x$$

Permet evitar posar parèntesis:

```
reverse $ 3:xs
```

La podem usar en funcions d'ordre superior.

## Altres funcions d'ordre superior predefinides

Ja hem vist `curry` i `uncurry`.

```
curry      :: ((a,b) -> c) -> a -> b -> c
curry f x y = f ( x , y )
```

```
uncurry    :: (a -> b -> c) -> (a,b) -> c
uncurry f ( p , s ) = f p s
```

Un altre exemple és el `flip`

```
flip :: (a -> b -> c) -> b -> a -> c
```

## Altres funcions d'ordre superior predefinides

Ja hem vist `curry` i `uncurry`.

```
curry      :: ((a,b) -> c) -> a -> b -> c
curry f x y = f ( x , y )
```

```
uncurry    :: (a -> b -> c) -> (a,b) -> c
uncurry f ( p , s ) = f p s
```

Un altre exemple és el `flip`

```
flip :: (a -> b -> c) -> b -> a -> c
```

```
flip f y x = f x y
```

## Altres funcions d'ordre superior predefinides

Ja hem vist `curry` i `uncurry`.

```
curry      :: ((a,b) -> c) -> a -> b -> c
curry f x y = f ( x , y )
```

```
uncurry    :: (a -> b -> c) -> (a,b) -> c
uncurry f ( p , s ) = f p s
```

Un altre exemple és el `flip`

```
flip :: (a -> b -> c) -> b -> a -> c
```

```
flip f = g
  where g y x = f x y
```

# Funcions d'ordre superior predefinides amb llistes

Primer exemple: `map`

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Aplica una funció a tots els element d'una llista.

```
map even [2,4,5,6,9]
```

```
[True,True,False,True,False]
```

# Funcions d'ordre superior predefinides llistes

❶ foldr i foldl

❷ iterate

❸ all i any

❹ filter

❺ dropWhile i takeWhile

❻ zipWith

i d'altres similars, com ara, scanf, scanr, concatMap, span, ...

# Continguts

- ①  $\lambda$ -termes a Haskell
- ② Currificació
- ③ Ordre superior
- ④ Llistes en Haskell
- ⑤ Lazy evaluation



# Generació de llistes en Haskell

Seqüències aritmètiques:

```
[1..5], ['a'..'z']
```

```
[1..] [1,3..]
```

Llistes per comprensió: [ <exp> | <q1>, ..., <qn> ]

Qualifiers q1,...,qn:

Generadors: x <- [ elems ]

Filtres: expressió booleana

- Exemples:

```
[ x*x | x <- [1..10], even x ]
```

```
[ ( x, y ) | x <- [ 1, 2, 3 ], y <- [ 9, 10, 11 ] ]
```

```
[ ( x, y ) | x <- [ 1, 2, 3 ], y <- [ 1 ..x ] ]
```

```
[ x | x <- [10..], let s = show x, s == reverse s ]
```

- pot ser més eficient un canvi d'ordre:

```
[ ( x, y ) | x <- [ 1 .. 3 ], y <- [ 1 ..2 ], even x ]
```

```
[ ( x, y ) | x <- [ 1 .. 3 ], even x, y <- [ 1 ..2 ] ]
```

# Funcions predefinides amb llistes

① `head`   `i`   `tail`

② `init`   `i`   `last`

③ `reverse`

④ `length` , `!!`   `i`   `elem`

⑤ `maximum`   `i`   `minimum`

⑥ `and`   `i`   `or`

⑦ `sum`   `i`   `product`

⑧ `take`   `i`   `drop`   `==`   `splitAt`

⑨ `takeWhile`   `i`   `dropWhile`   `==`   `span`

# Continguts

- ①  $\lambda$ -termes a Haskell
- ② Currificació
- ③ Ordre superior
- ④ Llistes en Haskell
- ⑤ Lazy evaluation

# Lazy evaluation

- Només avalua el que cal.
- Un valor que encara no han estat avaluats s'anomena "thunk"
- Podem avaluar una funció que té "thunks" com arguments
- Provoca cert indeterminisme, sobre com s'executa.
- Ineficiència(?). Depen del compilador i depen del cas.
- Permet tractar estructures potencialment molt grans o "infinites"

# Aplicacions de la lazy evaluation

## Exemples

- Càlcul del Fibonacci (eficient).
- Generar primers.
- Obtenir l'enèsim en ordre.
- Càlcul de  $e^x$  per Taylor.

# Avaluació eager/lazy

En Haskell es pot forçar cert nivell d'avaluació eager usant l'operador infix `$!`

`f$!x` avalua primer `x` i després `fx`

però només avalua fins que troba un constructor.