Programación Funcional

Curso 2019-20

λ-EXPRESIONES

Funciones anónimas: λ-expresiones

$\lambda x.e \equiv$ función que aplicada a x devuelve e

- $\lambda x.e$ es una función sin nombre *λ-abstracción*
- x variable ligada en $\lambda x.e$. La ligadura acaba con e.
- Regla de evaluación *β-reducción*:
  $(\lambda x.e)e' = e[x/e']$ donde $e[x/e']$ indica sustitución en e de las apariciones libres de x en e por e'
  $(\lambda x.x + 1)\ 2 = 2 + 1 = 3$ $(\lambda y.y + 1)\ 2 = 2 + 1 = 3$
  $(\lambda x.3)\ 2 = 3$ $(\lambda x.\lambda x.x)\ 3\ 2 = ???$
- Regla del renombramiento *α-conversión*
  $\lambda x.e \equiv \lambda y.e[x/y]$ $\lambda x.x + 1 \equiv \lambda y.y + 1$
- Atención al ámbito de la ligadura
  $(\lambda x.x + 1)((\lambda x.2 * x)\ 3) = 7$
- $\lambda x.\lambda y.e \equiv \lambda x.(\lambda y.e) \leadsto$ función de x, y (currificada)
  $(\lambda x.\lambda y.x + y)\ 3\ 2 = (\lambda y.3 + y)\ 2 = 3 + 2 = 5$

λ-cálculo

(Church,1936)

- Cálculo formal para reflejar cómputos mecanizables
- De potencia equivalente a las máquinas de Turing
- El material base son las λ-expresiones
- Cálculo original muy simple: sin tipos, sin valores primitivos, solo variables, λ's y aplicaciones.
- Gran variedad de variantes y extensiones
- Muy vigente en las ciencias de la computación

λ-cálculo soporta currificación, OS, aplicaciones parciales

- $(\lambda x.\lambda y.x + y)\ 3 = (\lambda y.3 + y)$
- $\lambda f.\lambda g.\lambda x.f(g\ x)$ *composición de funciones*
  $(\lambda f.\lambda g.\lambda x.f(g\ x))\ (\lambda x.2 * x)\ (\lambda x.x + 1)\ 3\ =\ ???$
- $(\lambda x.\lambda y.\lambda z.y\ x\ (z\ x))\ 2\ (\lambda x.\lambda y.y * x) =$
  $(\lambda y.\lambda z.y\ 2\ (z\ 2))(\lambda x.\lambda y.y * x) =$
  $\lambda z.((\lambda x.\lambda y.y * x)\ 2)(z\ 2) =$ $\lambda z.(((\lambda y.y * 2)(z\ 2))) =$
  $\lambda z.((z\ 2) * 2)$

λ-expresiones en Haskell

$\lambda x.e$ se escribe `\x -> e` no confundir con la `->` de los tipos

$\lambda x.\lambda y.e$ se escribe `\x y -> e` $\equiv$ `\x->(\y -> e)`

- `\x y -> e` azúcar para `\x -> \y -> e`
- $[\![$`(\x->x+1) 2`$]\!]$ `= 3`   `\x->x+1` $\equiv$ `\y->y+1`

  $[\![$`(\x->x+1)((\x->2*x) 3)`$]\!]$ `= 7` $[\![$`(\x y->x+y) 3 2`$]\!]$ `= 5`

  $[\![$`(\x x->x) 3 2 `$]\!]$`= ???` $[\![$`(\x -> \x->x) 3 2`$]\!]$ `= ???`
- Se integra bien con los tipos:

  `\x->x+1::Int->Int`   `\x.x::`$\forall$`a.a->a`

Más ejemplos de uso

$[\![$`map (\x->x+1) [1,2,3]`$]\!]$ `= [2,3,4]`

$[\![$`filter (\x -> x^2<7) [1,3,-2,-3]`$]\!]$ `= [1,-2]`

```
id = \x -> x
```

```
const x = \_ -> x
```

```
incList = map (\x -> x+1)
```

```
zip = zipWith (\x y-> (x,y))
```

λ-expresiones en Haskell soportan ajuste de patrones

$[\![$`(\(x,y) -> x) (3,4)`$]\!]$ `= 3` -- expresa `fst (3,4)`

$[\![$`(\(x:xs) -> x) [1,2]`$]\!]$ `= 1` -- expresa `head []`

$[\![$`(\(x:xs) -> x) []`$]\!]$ `=` $\perp$

Distinción de casos de ajuste vía *case*

```
\x y ->  case x of
             True  -> True
             _     -> y
```

-- expresa `||`