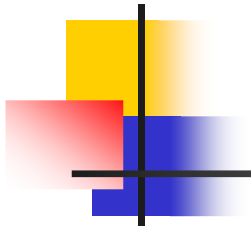


Tema 3. Constructores de tipos

Para definir tipos compuestos a partir de otros tipos

Tupla: $(T_1, T_2, ..., T_n) \equiv T_1 \times T_2 \times ... \times T_n$

- Estructura de datos heterogénea

Lista: $[T]$

- Estructura de datos homogénea

Función: $T_1 \rightarrow T_2$

Tuplas $(T_1, T_2, \ldots, T_n)$

Ejs: ('a', (7,8)) :: (Char, (Int,Int))

("hola", 8.7, True) :: (String, Float, Bool)

- Patrones de tuplas:

  (x,y) ó (x,y,z) ó (x,(y,z)) .....

- Definición de funciones por ajuste de patrones:

  fst (x,y) = x

  snd (x,y) = y

  (fst y snd predefinidas para los pares)

Ejemplo con tuplas

```
raices :: (Float,Float,Float) –> (Float, Float)
raices (a,b,c)
        | a == 0        = error “no es de 2 grado”
        | e < 0         = error “raices complejas”
        | otherwise     = ((-b-s)/d, (-b+s)/d)
                            where       e  = b*b - 4*a*c
                                        s = sqrt e
                                        d = 2*a
```

? raices (1,3,2)

(-2.0,-1.0)

Ejemplo con tuplas / Currificación

Definimos `minimo :: (Int, Int) –> Int`

`minimo (x,y) = if x<y then x else y`

su versión currificada es:

`minimoc :: Int –> Int –> Int`

`minimoc x y = if x<y then x else y`

- *Expresiones correctas:* `minimo (5,2)` `minimoc 5 2`
- *Expresiones incorrectas:* `minimo 5 2` `minimoc (5,2)`

Funciones “curry” y “uncurry”

Funciones predefinidas en Haskell:

curry f x y = f (x,y)

uncurry g (x,y) = g x y

curry f es la versión currificada de f:: (T_1,T_2) –>T

uncurry g es la versión descurrificada de g:: T_1–>T_2 –>T

Ej: Para la función minimo :: (Int, Int) –> Int

curry minimo es equivalente a minimoc :: Int –> Int –> Int

- *Exp. correctas:* minimo (5,2) (curry minimo) 5 2

Listas [T]

Ejs: [3,4,5,9] :: [Int] [3,4,5,9] == 3:4:5:9:[]

['a','4','+'] :: [Char]

- Patrones de listas:

  [] lista vacía

  (x:s) lista con cabeza x y resto s

- Definición de funciones por ajuste de patrones:

  head (x:s) = x (head [] = error)

  tail (x:s) = s (tail [] = error)

Ejemplo con listas

```
-- Elementos positivos de una lista
positLista :: [Int] –> [Int]
positLista [] = []
positLista (x:s) =  if  x>0   then  x: positLista s
                              else  positLista s
```

script

```
? positLista [3,-2,9,-5,0]
  [3,9]
```

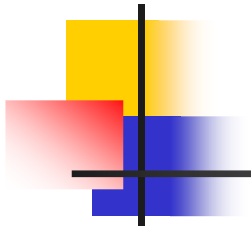
NOTA: script con comentarios, declaraciones y definiciones

Tipos sinónimos

type <tipo> = <expresión de tipo>

- Nuevo nombre para un tipo => Mayor legibilidad
- **type** String = [Char] "hola" == ['h','o','l','a']

Ejs:
```
type Biblioteca = [Libro]
type Libro = (Titulo, FechaPub, Autor)
type Autor = (Nombre, Apellido)
type FechaPub = Int
type Nombre = String
type Apellido = String
type Titulo  = String
```

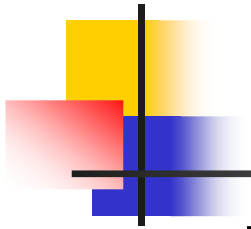


Ejemplo “biblioteca”

```
libro1 = (“Haskell”, 1998, (“Richard”, “Bird”)) :: Libro
libro2 = (“Miranda”, 1995, (“Alan”,”Turner”)) :: Libro
.....
mibib = [libro1, libro2, ...] :: Biblioteca

darAutor :: Libro –> Autor
darAutor (t,f,a) = a

obtenerTitulos :: Biblioteca –> [Titulo]
obtenerTitulos [] = []
obtenerTitulos ((t,f,a):res) = t: obtenerTitulos res
```



Polimorfía

- Tipo de las listas es polimórfico: $[\alpha]$
  con funciones polimórficas como head:: $[\alpha] \rightarrow \alpha$
  (donde α es una variable de tipo)
- Tipo de las tuplas es polimórfico: (α, β) (α, β, γ) .....
  con funciones polimórficas como fst:: $(\alpha, \beta) \rightarrow \alpha$
  (donde α, β son variables de tipo)
- La función error es polimórfica: error:: String $\rightarrow \alpha$

NOTA: Usaremos letras griegas α, β, χ, ... para las variables de tipo aunque en Haskell son a,b,c,....

Instanciación de tipos

Se obtiene al sustituir de *forma consistente* las variables (de tipo) por tipos.

Ejs:

(Char,Char) es instancia de (α, β)

(Int,[Int]) es instancia de (α, β)

[Char] –> Char es instancia de $[\alpha]\text{–>}\alpha$

pero

[Char] –> Int NO es instancia de $[\alpha]\text{–>}\alpha$

(*) todas las apariciones de una variable por el mismo tipo

Inferencia de tipos (1)

- Se infiere el tipo más general (si existe) para las funciones definidas por el usuario.

  Ej: Para f x y = (head x, y) se infiere como tipo

  f :: [α] –> β –> (α, β)

- Preguntas para cualquier instanciación del tipo de f

  ? f [‘a’,’e’,’u’] “ko”

  (’a’, “ko”) :: (Char, [Char])

  donde α β

  ? f [‘4’] True

  (‘4’, True) :: (Char, Bool)

  α β

Inferencia de tipos (2)

? f ‘a’ 3 f :: [α]–>β–>(α, β)

ERROR: No concuerda el tipo esperado con el real

* Expresión : f ‘a’ 3
* Término : ‘a’
* Tipo real : Char
* Tipo esperado : [α]

? head [‘c’, “mesa”] head:: [α]–>α

ERROR: No concuerda el tipo esperado con el real

* Subexpresión : [‘c’, “mesa”]
* Término : “mesa”
* Tipo real : String
* Tipo esperado : Char

Comprobación de tipos (1)

- Se comprueba si el tipo declarado es una instancia del tipo inferido.

Ej: f :: [Char] –> Integer –> (Char, Integer)

f x y = (head x, y)

Comprobación correcta:

[Char] –> Integer –> (Char, Integer) es instancia de

[α]–>β–>(α, β) para α = Char y β = Integer

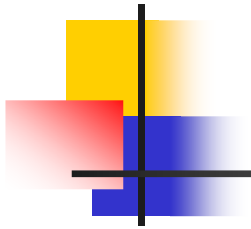
? f [‘a’,’e’,’u’] 3

(’a’,3) :: (Char, Integer)

? f [4,5] True

ERROR: No concuerda el tipo esperado con el real .....

Comprobación de tipos (2)



- Se obtiene un error si el tipo declarado no es una instancia del inferido por el sistema

```
f :: [Char] –> Integer –> (Bool, Integer)
f x y = (head x, y)
```

Comprobación: error

- Se obtiene un error si no existe tipo inferido para una definición dada (sin declaración)

```
g x y = (head x, x + y)
```

Comprobación: error