

# Programación funcional con Haskell

## Tipos

Juan Manuel Rabasedas



# Declaración de Tipos

- En Haskell podemos definir un nuevo nombre para un tipo existente usando una declaración **type**.

```
type String = [Char]
```

String es un sinónimo del tipo [Char].

- Los **sinónimos de tipo** hace que ciertas declaraciones de tipos sean más fáciles de leer.

```
type Pos = (Int, Int)
```

```
origin    :: Pos
```

```
origin    = (0, 0)
```

```
left      :: Pos → Pos
```

```
left (x, y) = (x - 1, y)
```

# Declaración de Tipos

- Como en las funciones, la declaración de tipos puede tener parámetros:

```
type Par a = (a, a)
mult :: Par Int → Int
mult (x, y) = x * y
copy  :: a → Par a
copy x = (x, x)
```

- Las declaraciones de tipo pueden anidarse

```
type Pos = (Int, Int)
type Trans = Pos → Pos ✓
```

pero no pueden ser recursivos

```
type Tree = (Int, [Tree]) ✕
```

- Un nuevo tipo completo puede ser definido dando sus valores con la declaración **data**.

```
data Bool = False | True
```

declara un nuevo tipo Bool con dos nuevos valores False y True.

- True y False son los llamados constructores del tipo Bool
- Los nombres de los constructores deben empezar con mayúsculas.
- Los constructores definen los distintos elementos del tipo.

- Los valores de un nuevo tipo se usan igual que los predefinidos

```
data Answer = Yes | No | Unknown
```

```
Answers :: [Answer]
```

```
Answers = [Yes, No, Unknown]
```

```
flip :: Answer → Answer
```

```
flip Yes      = No
```

```
flip No       = Yes
```

```
flip Unknown = Unknown
```

# Declaraciones **data**

- Los constructores en las declaraciones **data** pueden tener parámetros

```
data Shape = Circle Float | Rect Float Float
```

```
square    :: Float → Shape
```

```
square n = Rect n n
```

```
area      :: Shape → Float
```

```
area (Circle r) =  $\pi * r^2$ 
```

```
area (Rect x y) =  $x * y$ 
```

- Los constructores pueden ser vistos como funciones

```
Circle :: Float → Shape
```

```
Rect :: Float → Float → Shape
```

# Declaraciones **data**

No es sorpresa que las declaraciones **data** pueden también tener parámetros de tipos .

**data** Maybe  $a = \text{Nothing} \mid \text{Just } a$

$\text{safediv} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Maybe Int}$

$\text{safediv } _ 0 = \text{Nothing}$

$\text{safediv } m n = \text{Just } (m \text{ 'div' } n)$

$\text{safehead} :: [a] \rightarrow \text{Maybe } a$

$\text{safehead } [] = \text{Nothing}$

$\text{safehead } xs = \text{Just } (\text{head } xs)$

Maybe es un constructor de tipos ya que dado un tipo  $a$ , construye el tipo Maybe  $a$ .

# Declaraciones **data**

- Las declaraciones **data** pueden ser recursivas:

**data** Nat = Zero | Succ Nat

*add* n Zero = n

*add* n (Succ m) = Succ (*add* m n)

- Nat es un nuevo tipo con constructores Zero :: Nat y Succ :: Nat → Nat
- Un valor de tipo Nat puede ser Zero o de la forma Succ n donde n :: Nat  
Zero, Succ Zero, Succ (Succ Zero), Succ (Succ (Succ Zero)), ...
- Podemos pensar a los valores del tipo Nat como números naturales donde:
  - Zero representa a 0 y Succ representa la función sucesor (1+)
  - Succ (Succ (Succ Zero)) = (1 + 1 + 1 + 0) = 3



Usando recursión es fácil definir funciones que conviertan los valores entre los tipos `Nat` e `Int`

$$\text{nat2int} :: \text{Nat} \rightarrow \text{Int}$$
$$\text{nat2int Zero} = 0$$
$$\text{nat2int (Succ } n) = 1 + \text{nat2int } n$$
$$\text{int2nat} :: \text{Int} \rightarrow \text{Nat}$$
$$\text{int2nat } 0 = \text{Zero}$$
$$\text{int2nat } n = \text{Succ } (\text{int2nat } (n - 1))$$

# Declaraciones **data**

- Dos naturales pueden ser sumados convirtiéndolos en enteros, sumados, y convertidos nuevamente a naturales

$$\begin{aligned} \text{add} &:: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\ \text{add } m \ n &= \text{int2nat } (\text{nat2int } m + \text{nat2int } n) \end{aligned}$$

- Sin embargo usando recursión la misma función se puede definir sin la necesidad de conversión:

$$\begin{aligned} \text{add } \text{Zero } n &= n \\ \text{add } (\text{Succ } m) \ n &= \text{Succ } (\text{add } m \ n) \end{aligned}$$

- Ejercicio: definir la multiplicación para Nat
- Ejercicio: definir la exponenciación para Nat.

# Declaraciones **data**

- Las declaraciones **data** pueden ser recursivas y con parámetros  
**data** List  $a$  = Nill | Cons  $a$  (List  $a$ )
- Usando recursión es facil definir funciones que conviertan los valores entre los tipos List  $a$  y  $[a]$

$to :: \text{List } a \rightarrow [a]$

$to \text{ Nil} = []$

$to (\text{Cons } x \text{ } xs) = x : (to \text{ } xs)$

$from :: [a] \rightarrow \text{List } a$

$from [] = \text{Nil}$

$from (x : xs) = \text{Cons } x (from \text{ } xs)$

# Sintaxis de registro (Record)

Podemos crear un tipo que describa a una persona:

- Con los campos: nombre, apellidos, edad, altura, número de teléfono y el sabor de su helado favorito.

```
data Person =  
Person String String Int Float String String deriving (Show)
```

- creamos una persona ( una instancia ):

```
> let guy =  
  Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"  
> guy  
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

# Sintaxis de registro (Record)

Para obtener la información guardada en un tipo `Person` debemos definir funciones para cada campo:

*firstName*      (`Person firstName _ _ _ _`) = *firstName*

*lastName*      (`Person _ lastname _ _ _`) = *lastName*

*age*            (`Person _ _ age _ _`)            = *age*

*height*        (`Person _ _ _ height _`)        = *height*

*phoneNumber* (`Person _ _ _ _ number _`) = *number*

*flavor*        (`Person _ _ _ _ _ flavor`)        = *flavor*

*Debe de haber un método mejor.*

# Sintaxis de registro (Record)

Podemos declarar el mismo tipo de la siguiente manera:

```
data Person = Person { firstName :: String  
                      , lastName :: String  
                      , age :: Int  
                      , height :: Float  
                      , phoneNumber :: String  
                      , flavor :: String  
                      } deriving (Show)
```

Se crean automáticamente las funciones para obtener los campos: *firstName*, *lastName*, *age*, *height*, *phoneNumber* y *flavor*.

# Sintaxis de registro (Record)

Ahora podemos definir:

```
Person { lastName      = "Finklestein"  
        , firstName    = "Buddy"  
        , height       = 184.2  
        , age          = 43  
        , flavor       = "Chocolate"  
        , phoneNumbre = "526-2928"  
        }
```

Notar que no es necesario respetar el orden en que fueron declarado los campos.

- Además de pattern matching en el lado izq. de una definición, podemos usar una expresión **case**

```
esCero :: Nat → Bool  
esCero n = case n of  
    Zero → True  
    _    → False
```

- Los patrones de los diferentes casos son intentados en orden
- Se usa la indentación para marcar un bloque de casos



- Programming in Haskell. Graham Hutton, CUP 2007.
- Introducción a la Programación Funcional con Haskell. Richard Bird, Prentice Hall 1997.
- ¡Aprende Haskell por el bien de todos! - Capitulo 8  
Creando nuestros propios tipos y clases de tipos