

# Tipos en Haskell

Mauro Jaskelioff

04/04/2017

# Tipos en Haskell

- ▶ Haskell posee un sistema de tipos expresivo.
- ▶ Haskell tiene tipado estático.
  - ▶ El tipado estático significa que los tipos son chequeados en tiempo de compilación.
- ▶ Haskell tiene inferencia de tipos
  - ▶ Sin embargo, dar el tipo de las funciones es ventajoso.
- ▶ Veremos varias formas de definir tipos de datos en Haskell
  - ▶ Sinónimos de tipos
  - ▶ Tipos Algebraicos
  - ▶ Constructores de tipo

# Sinónimos de tipos

- ▶ En Haskell, se puede 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 hacen que ciertas declaraciones de tipos sean más fáciles de leer.

**type** *Pos* = (*Int*, *Int*)

*origen*    :: *Pos*

*origen*    = (0, 0)

*izq*        :: *Pos* → *Pos*

*izq* (*x*, *y*) = (*x* - 1, *y*)

# Sinónimos de Tipos

- ▶ Los sinónimos de tipo pueden tener parámetros

**type** *Par* *a* = (*a*, *a*)

*copiar* :: *a* → *Par a*

*copiar* *x* = (*x*, *x*)

- ▶ Los sinónimos de tipo pueden anidarse

**type** *Punto* = (*Int*, *Int*)

**type** *Trans* = *Punto* → *Punto*

pero no pueden ser recursivos

**type** *Tree* = (*Int*, [*Tree*])

# Declaraciones **data**

- ▶ los **data** declaran un nuevo tipo cuyos valores se especifican en la declaración.

**data** *Bool* = *False* | *True*

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

- ▶ *True* y *False* son los *constructores* del tipo *Bool*
- ▶ Los nombres de los constructores deben empezar con mayúsculas.
- ▶ Dos constructores diferentes siempre construyen diferentes valores del tipo.

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

**data** *Respuesta* = *Si* | *No* | *Desconocida*

*respuestas* :: [*Respuesta*]

*respuestas* = [*Si*, *No*, *Desconocida*]

*invertir* :: *Respuesta* → *Respuesta*

*invertir Si* = *No*

*invertir No* = *Si*

*invertir Desconocida* = *Desconocida*

# Usando **data**

- ▶ Ejemplo en que los constructores tienen 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 son funciones

> :t *Circle*

*Circle* :: *Float* → *Shape*

> :t *Rect*

*Rect* :: *Float* → *Float* → *Shape*

¿Qué complejidad tienen estas funciones?

# Sintaxis para Records

- Definimos un tipo de datos para guardar datos de alumnos:

**data** *Alumno* = *A String String Int String deriving Show*

- Definimos un alumno

*juan* = *A "Juan" "Perez" 21 "jperez999999@gmail.com"*

- Para acceder a los datos es útil definir funciones:

*nombre :: Alumno → String*

*nombre (A n \_ \_ \_) = n*

*apellido :: Alumno → String*

*apellido (A \_ a \_ \_) = a*

*edad :: Alumno → Int*

*edad (A \_ \_ e \_) = e*

*email :: Alumno → String*

*email (A \_ \_ \_ m) = m*



## Sintaxis para Records (cont.)

- ▶ Haskell provee sintaxis para aliviarnos el trabajo:

```
data Alumno = A { nombre :: String  
                  , apellido :: String  
                  , edad    :: Int  
                  , email   :: String  
                  } deriving Show
```

- ▶ No tenemos que definir las proyecciones por separado.
- ▶ Cambia la instancia de *Show*.
- ▶ No tenemos que acordarnos el orden de los campos:

```
juan = A { apellido = "Perez"  
          , nombre = "Juan"  
          , email = "jperez999999@gmail.com"  
          , edad = 21 }
```

# Constructores de Tipos

- ▶ Las declaraciones **data** pueden tener parámetros de tipos.

**data** *Maybe* *a* = *Nothing* | *Just* *a*

*safehead* :: [*a*] → *Maybe* *a*

*safehead* [] = *Nothing*

*safehead* *xs* = *Just* (*head* *xs*)

- ▶ *Maybe* es un constructor de tipos ya que dado un tipo *a*, construye el tipo *Maybe a*.
- ▶ *Maybe* tiene *kind* \* → \*. En GHCi:

> :kind *Maybe*

*Maybe* :: \* → \*

# El constructor de Tipos *Either*

**data** *Either* a b = *Left* a | *Right* b

- ▶ Describe un tipo que puede tener elementos de dos tipos.
- ▶ En sus elementos está claro de qué tipo es el elemento almacenado.
- ▶ Ejercicio: Dar 4 elementos diferentes de tipo *Either Bool Bool*.
- ▶ En una interpretación naif de tipos como conjuntos, el tipo *Either* corresponde a la *unión disjunta*.
- ▶ Ejercicio: ¿Qué *kind* tiene *Either*?

- ▶ Las declaraciones **data** pueden ser recursivos

**data**  $Nat = Zero \mid Succ\ Nat$

$add\ n\ Zero = n$

$add\ n\ (Succ\ m) = Succ\ (add\ n\ m)$

- ▶ Ejercicio: definir la multiplicación para  $Nat$

# Tipos Recursivos con Parámetros

- ▶ Por supuesto, los tipos recursivos pueden tener parámetros.

**data** *List* *a* = *Nil* | *Cons* *a* (*List* *a*)

- ▶ El tipo *List* es *isomorfo* a las listas predefinidas

*to* :: *List* *a* → [*a*]

*to Nil* = []

*to* (*Cons* *x* *xs*) = *x* : (*to* *xs*)

*from* :: [*a*] → *List* *a*

*from* [] = *Nil*

*from* (*x* : *xs*) = *Cons* *x* (*from* *xs*)

- ▶ 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.