

Tema 6. Definiciones de tipos

6.1 Sinónimos de tipo

6.2 Definiciones de tipos de datos

Tipos enumerados

Uniones

Productos

Registros variantes

Tipos recursivos

6.3 Tipos Polimórficos

Either

Maybe

Listas

Árboles binarios

6.1 Sinónimos de tipo

- ✓ Introducen un nuevo nombre para un tipo existente.
- ✓ Se usa la palabra clave `type` :

Ejemplos:

```
type Entero = Integer  
      Nuevo Nombre      Tipo Existente
```

```
uno :: Entero  
uno = 1
```

```
type DeEnteroEnEntero = Entero → Entero
```

```
sucesor :: DeEnteroEnEntero  
sucesor x = x + 1
```

```
type ParFlotantes = (Float, Float)
```

```
parCeros :: ParFlotantes  
parCeros = (0.0, 0.0)
```

```
type String = [Char] -- Predefinido en Prelude
```

- ✓ El nuevo nombre del tipo debe comenzar con mayúscula.

6.2 Definiciones de tipos de datos

- ✓ El programador puede definir nuevos tipos (palabra reservada `data`)
- ✓ Todos los nombres de tipo deben comenzar con mayúscula

Tipos enumerados

- ✓ Constan de un número finito de valores que se enumeran en la definición.

Ejemplo:

```
data DíaSemana = Lunes | Martes | Miércoles | Jueves | Viernes  
              | Sábado | Domingo deriving Show
```

```
unDía :: DíaSemana  
unDía = Lunes
```

```
laborables :: [DíaSemana]  
laborables = [Lunes, Martes, Miércoles, Jueves, Viernes]
```

- ✓ `DíaSemana` es un *constructor de tipo* (nombre del tipo definido).
- ✓ Los valores que puede tomar una variable del tipo `DíaSemana` son `Lunes`, `Martes`, ... o `Domingo`.
- ✓ Son los *constructores de datos*. También deben empezar con mayúscula.
- ✓ La cláusula `deriving Show` es necesaria para poder mostrar por pantalla los valores del tipo.
- ✓ Un mismo constructor de datos no puede aparecer en dos tipos distintos en un mismo *ámbito*.

Tipos enumerados (2)

Los constructores de datos se pueden usar como patrones:

```
esFinSemana      :: DíaSemana → Bool
esFinSemana Sábado = True
esFinSemana Domingo = True
esFinSemana _     = False
```

Otro ejemplo (predefinido):

```
data Bool = False | True deriving (Show, ...)
```

```
infixr 3 &&
(&&)      :: Bool → Bool → Bool
False && x = False
True && x  = x
```

```
infixr 2 ||
(||)      :: Bool → Bool → Bool
False || x = x
True || x  = True
```

Uniones

- ✓ Unión de varios tipos existentes:

```
data LetraOEntero = Letra Char | Entero Integer deriving Show
```

- ✓ Los valores del tipo *LetraOEntero* son:

- ◇ Los valores del tipo *Char* precedidos del constructor *Letra*
- ◇ Los valores del tipo *Integer* precedidos del constructor *Entero*

```
unValor :: LetraOEntero
unValor = Letra 'x'
```

```
otroValor :: LetraOEntero
otroValor = Entero 15
```

```
listaMixta :: [LetraOEntero]
listaMixta = [ Letra 'a', Entero 10, Entero 12, Letra 'b' ]
```

- ✓ Los constructores los elige el programador pero **son obligatorios**
- ✓ Cada constructor introducido tiene un tipo (no hay que declararlo)

```
Letra    :: Char → LetraOEntero
Entero   :: Integer → LetraOEntero
```

- ✓ Los constructores de datos pueden actuar como patrones y funciones:

```
incLoE      :: LetraOEntero → LetraOEntero
incLoE (Entero n) = Entero (n + 1)
incLoE (Letra c)  = Letra (chr (1 + ord c))
```

```
? incLoE (Letra 'a')
Letra 'b' :: LetraOEntero
```

```
? incLoE (Entero 10)
Entero 11 :: LetraOEntero
```

Productos

- ✓ Tipos con un único constructor y varias componentes

```
data Racional = Par Integer Integer deriving Show
                   Numerador Denom.
```

- ✓ Los valores del tipo *Racional* son cualesquiera dos valores de tipo *Integer* precedidos del constructor *Par*:

```
unMedio :: Racional
unMedio = Par 1 2
```

- ✓ Tipo del constructor (no hay que declararlo):

```
Par :: Integer → Integer → Racional
```

Ejemplos:

```
numerador :: Racional → Integer
numerador (Par x _) = x
```

```
denominador :: Racional → Integer
denominador (Par _ y) = y
```

```
infixl 7 >*<
(>*<) :: Racional → Racional → Racional
(Par a b) >*< (Par c d) = Par (a * c) (b * d)
```

```
? numerador (Par 1 3)
1 :: Integer
```

```
? (Par 1 2) >*< (Par 1 3)
Par 1 6 :: Racional
```

Constructores simbólicos

- ✓ Si un constructor de datos es binario su nombre puede ser simbólico
- ✓ Pueden mejorar legibilidad
- ✓ Ha de comenzar por el carácter dos puntos (:)
- ✓ El constructor se escribe entre las dos componentes del tipo (infijo)

```
infix 9 :/  
data Racional =  $\underbrace{Integer}_{Num.} :/ \underbrace{Integer}_{Denom.}$  deriving Show
```

- ✓ Los valores del tipo *Racional* son dos valores de tipo *Integer* con el constructor (:/) infijo:

```
unMedio :: Racional  
unMedio = 1 :/ 2
```

- ✓ Tipo del constructor (no hay que declararlo):

```
(:/) :: Integer → Integer → Racional
```

Ejemplos:

```
numerador :: Racional → Integer  
numerador (x :/ _) = x
```

```
denominador :: Racional → Integer  
denominador (_ :/ y) = y
```

```
infixl 7 >*<  
(>*<) :: Racional → Racional → Racional  
a :/ b >*< c :/ d = (a * c) :/ (b * d)
```

```
? numerador (1 :/ 3)  
1 :: Integer
```

```
? 1 :/ 2 >*< 1 :/ 3  
1 :/ 6 :: Racional
```

Registros variantes

- ✓ Mezcla de Uniones, Productos y Enumerados
- ✓ Permiten expresar distintas formas para valores de un mismo tipo
- ✓ Cada forma puede tener un número distinto de componentes

Ejemplo: Tipo para representar cuatro clases de figuras

```
type Radio = Float
type Lado = Float
type Base = Float
type Altura = Float
```

```
data Figura = Círculo Radio
            | Cuadrado Lado
            | Rectángulo Base Altura
            | Punto
            deriving Show
```

```
unCírculo :: Figura
unCírculo = Círculo 25
```

```
unRectángulo :: Figura
unRectángulo = Rectángulo 10 15
```

```
listaFiguras :: [Figura]
listaFiguras = [Círculo 15, Cuadrado 3, Rectángulo 5 6]
```

```
área :: Figura → Float
área (Círculo r) = pi * r ^ 2
área (Cuadrado l) = l ^ 2
área (Rectángulo b h) = b * h
área Punto = 0
```

Tipos recursivos

- ✓ Alguna componente de un constructor puede tener el tipo que se está definiendo
- ✓ Permiten definir tipos con cardinalidad infinita

Ejemplo: Los naturales

```
data Nat = Cero | Suc Nat deriving Show
```

- ✓ Un valor de tipo *Nat* puede tener dos formas
 - ◇ La constante *Cero*
 - ◇ El constructor *Suc* seguido de otro valor de tipo *Nat*
- ✓ Iterando la segunda forma se generan los naturales distintos a *Cero*:

$$\underbrace{Suc\ Cero}_{uno} \quad \underbrace{Suc\ (Suc\ Cero)}_{dos} \quad \underbrace{Suc\ (Suc\ (Suc\ Cero))}_{tres} \quad \dots$$

- ✓ Incluso se puede definir ∞

```
inf :: Nat
inf = Suc inf
```

```
? inf
Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc ...
```


6.3 Tipos Polimórficos

Los tipos también pueden ser polimórficos.

Either

Tipo predefinido para representar la unión de otros dos tipos arbitrarios:

```
data Either a b = Left a | Right b deriving Show
```

- ✓ Los valores de tipo *Either a b* son los valores de tipo *a* precedidos del constructor *Left* y los valores de tipo *b* precedidos de *Right*.

Ejemplo: Listas con enteros y booleanos:

```
l1 :: [Either Integer Bool]
l1 = [Left 1, Right True, Left 3, Left 5]
l2 :: [Either Bool Integer]
l2 = [Right 2, Left False, Right 5]
```

Maybe

Tipo predefinido para representar valores parciales:

```
data Maybe a = Nothing | Just a deriving Show
```

- ✓ Los valores de tipo *Maybe a* son los valores de tipo *a* precedidos de *Just* y además un valor especial que se escribe *Nothing*
- ✓ *Nothing* se suele usar para representar *no definido*:

```
recíproco :: Float → Maybe Float
recíproco 0 = Nothing
recíproco x = Just (1/x)
```

Listas

Podemos definir una lista polimórfica homogénea (todos los elementos tienen el mismo tipo):

$$\text{data } Lista\ a = Vacía \mid Cons \underbrace{a}_{cabeza} \underbrace{(Lista\ a)}_{cola} \text{ deriving Show}$$

Las listas definidas tienen dos formas posibles:

- ✓ Puede ser la lista vacía, representada por *Vacía*
- ✓ Puede ser una lista no vacía, representada por *Cons cabeza cola* donde la cabeza ha de tener tipo *a* y la cola tipo *Lista a*

Ejemplos:

```
l3 :: Lista Integer
l3 = Cons 1 (Cons 2 (Cons 3 Vacía))

l4 :: Lista Bool
l4 = Cons True (Cons True (Cons False Vacía))
```

Para estructuras lineales es mejor usar un constructor simbólico:

```
infixr 5 :>
data Lista a = Vacía |  $\underbrace{a}_{cabeza} :> \underbrace{(Lista\ a)}_{cola}$  deriving Show

l3 :: Lista Integer
l3 = 1 :> 2 :> 3 :> Vacía
```

Ejemplo poco práctico, ya que las listas están predefinidas

Árboles binarios

Podemos definir un árbol binario polimórfico homogéneo con datos en nodos:

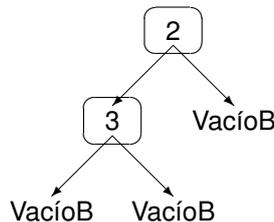
$$\text{data } \text{ÁrbolB } a = \text{VacíoB} \mid \text{NodoB } \overbrace{(\text{ÁrbolB } a)}^{\text{hijo izq}} \underbrace{a}_{\text{dato en nodo}} \overbrace{(\text{ÁrbolB } a)}^{\text{hijo der}}$$

deriving Show

Los árboles definidos tienen dos formas posibles:

- ✓ Puede ser un árbol binario vacío, representado por *VacíoB*
- ✓ Puede ser un árbol binario no vacío, representada por *Nodo izq dato der* donde *izq* y *der* (que representan el subárbol izquierdo y derecho) han de tener tipo *ÁrbolB a* y *dato*, que es el dato almacenado en el nodo, tiene tipo *a*

Ejemplos:



$a :: \text{ÁrbolB Integer}$

$a = \text{NodoB } ai \ 2 \ ad$

where

$ai = \text{NodoB } \text{VacíoB } 3 \ \text{VacíoB}$

$ad = \text{VacíoB}$

$sum \text{ÁrbolB} \quad \quad \quad :: \text{ÁrbolB Integer} \rightarrow \text{Integer}$

$sum \text{ÁrbolB } \text{VacíoB} \quad \quad = 0$

$sum \text{ÁrbolB } (\text{NodoB } i \ r \ d) = sum \text{ÁrbolB } i + r + sum \text{ÁrbolB } d$

Objetivos del tema

El alumno debe:

- ✓ Saber definir y utilizar sinónimos de tipos
- ✓ Saber definir tipos enumerados, uniones, productos, registros variantes y tipos recursivos
- ✓ Entender las definiciones de tipo
- ✓ Saber definir funciones sobre tipos definidos
- ✓ Saber reducir expresiones en las que aparezcan tipos definidos
- ✓ Entender la función de plegado *foldNat* y saber definir otras funciones como concreciones de ésta
- ✓ Saber definir y utilizar tipos polimórficos
- ✓ Conocer las reglas léxicas de Haskell