Tema 6. Definiciones de tipos

6.1 Sinónimos de tipo

✓ Introducen un nuevo nombre para un tipo existente.

✓ Se usa la palabra clave **type** :

Ejemplos:

$$\textbf{type}\ \underbrace{\mathit{Entero}}_{\mathit{Nuevo\ Nombre}} = \underbrace{\mathit{Integer}}_{\mathit{Tipo\ Existente}}$$

$$\begin{aligned} &\mathit{uno} :: \mathit{Entero} \\ &\mathit{uno} = 1 \end{aligned}$$

$$\textbf{type}\ \mathit{DeEnteroEnEntero} = \mathit{Entero} \to \mathit{Entero}$$

$$\begin{aligned} &\mathit{sucesor} :: \mathit{DeEnteroEnEntero} \\ &\mathit{sucesor}\ x = x + 1 \end{aligned}$$

$$\textbf{type}\ \mathit{ParFlotantes} = (\mathit{Float},\ \mathit{Float})$$

$$\begin{aligned} &\mathit{parCeros} :: \mathit{ParFlotantes} \\ &\mathit{parCeros} = (0.0,\ 0.0) \end{aligned}$$

$$\textbf{type}\ \mathit{String} = [\mathit{Char}] \quad -- \text{ Predefinido en Prelude}$$

✓ 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:

$$\begin{array}{l} \textbf{data}\ \mathit{DíaSemana} = \mathit{Lunes} \mid \mathit{Martes} \mid \mathit{Miércoles} \mid \mathit{Jueves} \mid \mathit{Viernes} \\ \qquad\qquad\qquad\qquad \mid \mathit{Sábado} \mid \mathit{Domingo}\ \ \textbf{deriving}\ \mathit{Show} \end{array}$$

$$\begin{array}{l} \mathit{unDía} :: \mathit{DíaSemana} \\ \mathit{unDía} = \mathit{Lunes} \end{array}$$

$$\begin{array}{l} \mathit{laborables} :: [\mathit{DíaSemana}] \\ \mathit{laborables} = [\mathit{Lunes}, \mathit{Martes}, \mathit{Miércoles}, \mathit{Jueves}, \mathit{Viernes}] \end{array}$$

✓ *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.

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:

$$\textbf{data}\ LetraOEntero\ =\ Letra\ Char \mid Entero\ Integer\ \ \textbf{deriving}\ Show$$

✓ 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

$$\textbf{data}\ Racional = Par\ \underbrace{Integer}_{Numerador}\ \underbrace{Integer}_{Denom.}\ \textbf{deriving}\ Show$$

✓ 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)

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

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

$$\begin{array}{l} \mathit{unMedio} :: \mathit{Racional} \\ \mathit{unMedio} = 1 :/ 2 \end{array}$$

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

$$(:/) :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Racional}$$

Ejemplos:

$$\begin{array}{l} \mathit{numerador} :: \mathit{Racional} \to \mathit{Integer} \\ \mathit{numerador}\ (x :/ \_) = x \end{array}$$

$$\begin{array}{l} \mathit{denominador} :: \mathit{Racional} \to \mathit{Integer} \\ \mathit{denominador}\ (\_ :/ y) = y \end{array}$$

$$\begin{array}{l} \textbf{infixl}\ 7\ >\!*\!< \\ (>\!*\!<) :: \mathit{Racional} \to \mathit{Racional} \to \mathit{Racional} \\ a :/ b >\!*\!< c :/ d = (a * c) :/ (b * d) \end{array}$$

? $\mathit{numerador}\ (1 :/ 3)$
$1 :: \mathit{Integer}$

? $1 :/ 2 >\!*\!< 1 :/ 3$
$1 :/ 6 :: \mathit{Racional}$

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

$$\textbf{data}\ Nat\ =\ Cero \mid Suc\ Nat\ \ \textbf{deriving}\ Show$$

✓ 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 $Cero$:

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

✓ Incluso se puede definir ∞

$$\begin{array}{lcl} inf & :: & Nat \\ inf & = & Suc\ inf \end{array}$$

? inf
$Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ \ldots$

Tipos recursivos (2)

✓ La recursión permite definir funciones que actúen sobre el tipo Nat de un modo elegante.

Ejemplos:

$$
\begin{array}{lcl}
esPar & :: & Nat \to Bool \\
esPar\ Cero & = & True \\
esPar\ (Suc\ n) & = & not\ (esPar\ n)
\end{array}
$$

infixl 6 $<+>$

$$
\begin{array}{lcl}
(<+>) & :: & Nat \to Nat \to Nat \\
Cero\ <+>\ m & = & m \\
(Suc\ n)\ <+>\ m & = & Suc\ (n <+>\ m) \\
 & & \quad -- \text{ ya que } (n+1)+m = (n+m)+1
\end{array}
$$

infixl 7 $<*>$

$$
\begin{array}{lcl}
(<*>) & :: & Nat \to Nat \to Nat \\
Cero\ <*>\ m & = & Cero \\
(Suc\ n)\ <*>\ m & = & n\ <*>\ m\ \ <+>\ m \\
 & & \quad -- \text{ ya que } (n+1)*m = n*m+m
\end{array}
$$

Ejemplo de evaluación:

$$
\begin{array}{l}
\underline{Suc\ (Suc\ Cero)\ <+>\ Suc\ Cero} \\
\Longrightarrow \{\text{segunda ecuación de } (<+>)\} \\
Suc\ (\underline{(Suc\ Cero)\ <+>\ Suc\ Cero}) \\
\Longrightarrow \{\text{segunda ecuación de } (<+>)\} \\
Suc\ (Suc\ (\underline{Cero\ <+>\ Suc\ Cero})) \\
\Longrightarrow \{\text{primera ecuación de } (<+>)\} \\
Suc\ (Suc\ (Suc\ Cero))
\end{array}
$$

Plegado para el tipo Nat

✓ Muchas funciones sobre el tipo Nat siguen el mismo esquema:

$$
\begin{array}{lcl}
esPar & :: & Nat \to Bool \\
esPar\ Cero & = & True \\
esPar\ (Suc\ n) & = & not\ (esPar\ n)
\end{array}
$$

$$
\begin{array}{lcl}
aInteger & :: & Nat \to Integer \\
aInteger\ Cero & = & 0 \\
aInteger\ (Suc\ n) & = & 1 + (aInteger\ n)
\end{array}
$$

✓ El esquema común es:

$$
\begin{array}{lcl}
fun & :: & Nat \to a \\
fun\ Cero & = & \boxed{e} \\
fun\ (Suc\ n) & = & \boxed{f}\ (fun\ n)
\end{array}
$$

✓ Una función de orden superior para este esquema

$$
\begin{array}{l}
foldNat :: (a \to a) \to a \to (Nat \to a) \\
foldNat\ f\ e = fun \\
\quad \textbf{where} \\
\qquad \begin{array}{lcl}
fun\ Cero & = & e \\
fun\ (Suc\ n) & = & f\ (fun\ n)
\end{array}
\end{array}
$$

✓ O equivalentemente (ya que $fun \equiv foldNat\ f\ e$)

$$
\begin{array}{lcl}
foldNat & :: & (a \to a) \to a \to Nat \to a \\
foldNat\ f\ e\ Cero & = & e \\
foldNat\ f\ e\ (Suc\ n) & = & f\ (foldNat\ f\ e\ n)
\end{array}
$$

✓ Las funciones originales como concreción de $foldNat$:

$$
\begin{array}{lcl}
esPar & :: & Nat \to Bool \\
esPar & = & foldNat\ not\ True
\end{array}
$$

$$
\begin{array}{lcl}
aInteger & :: & Nat \to Integer \\
aInteger & = & foldNat\ (1+)\ 0
\end{array}
$$

6.3 Tipos Polimórficos

Los tipos también pueden ser polimórficos.

Either

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

$$\textbf{data}\ \ Either\ a\ b\ =\ \ Left\ a \mid Right\ b\ \ \textbf{deriving}\ Show$$

✓ 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:

$$\begin{array}{lll} l1 & :: & [Either\ Integer\ Bool] \\ l1 & = & [Left\ 1,\ Right\ True,\ Left\ 3,\ Left\ 5] \\ l2 & :: & [Either\ Bool\ Integer] \\ l2 & = & [Rigth\ 2,\ Left\ False,\ Right\ 5] \end{array}$$

Maybe

Tipo predefinido para representar valores parciales:

$$\textbf{data}\ Maybe\ a\ =\ \ Nothing \mid Just\ a\ \ \textbf{deriving}\ Show$$

✓ 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*:

$$\begin{array}{lll} recíproco & :: & Float \rightarrow Maybe\ Float \\ recíproco\ 0 & = & Nothing \\ recíproco\ x & = & Just\ (1/x) \end{array}$$

Listas

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

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

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:

$$\begin{aligned} l3\ &::\ \ Lista\ Integer \\ l3\ &=\ \ Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Vacía)) \\ l4\ &::\ \ Lista\ Bool \\ l4\ &=\ \ Cons\ True\ (Cons\ True\ (Cons\ False\ Vacía)) \end{aligned}$$

Para estructuras lineales es mejor usar un constructor simbólico:

$$\begin{aligned} &\textbf{infixr}\ 5\ :> \\ &\textbf{data}\ \ Lista\ a\ =\ \ Vacía \mid \underbrace{a}_{cabeza}\ :>\ \underbrace{(Lista\ a)}_{cola}\ \ \textbf{deriving}\ Show \\ &l3\ ::\ \ Lista\ Integer \\ &l3\ =\ \ 1\ :>\ 2\ :>\ 3\ :>\ Vacía \end{aligned}$$

Ejemplo poco práctico, ya que las listas están predefinidas

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