Tema 6. Tipos algebraicos

data <tipo> = <C_1> <tipo(s)> | <C_2> <tipo(s)> | .....

- *Se definen mediante sus constructoras* C_1 , C_2 , ...
- *Pueden ser polimórficos y recursivos*

Ejs: **data** Dia = Lu | Ma | Mi | Ju | Vi | Sa | Do

data Bool = False | True

- *Funciones definidas mediante ajuste de patrones*

```
festivo :: Dia –> Bool
festivo Do = True
festivo d = False
```

```
not :: Bool –> Bool
not True = False
not False = True
```

newtype

- A **newtype** declaration creates a new type in much the same way as **data**.

- You can replace the **newtype** keyword with **data** and it'll still compile and works
- The converse is not true:

  data can only be replaced with **newtype**

  if

  the type has **exactly one constructor**

  with **exactly one field** (type argument) inside it.

Ejemplos

newtype Fecha = F (Int,Int,Int)

newtype Natural a = Nat a

data Racional = Int :/ Int

newtype RacionalN = C (Int,Int)

Enumeraciones

Ej: **data** Dia = Lu | Ma | Mi | Ju | Vi | Sa | Do

Para definir Dia *como tipo enumerado:*

a) Lo declaramos instancia de Enum:

```
instance Enum Dia where
  fromEnum Lu = 0
  ----------------
  fromEnum Do = 6
  toEnum 0 = Lu
  ----------------
  toEnum 6 = Do
```

b) o derivamos la instancia automáticamente("deriving")

Clausula “deriving” (1)

```
data Dia =  Lu | Ma | Mi | Ju | Vi | Sa | Do
            deriving (Eq, Ord, Enum, Show)
```

Por ser instancia de Eq, Ord, Enum:

? Lu == Ma	? Ju < Sa	? pred Sa == succ Mi
False	True	False

Por ser instancia de Enum, Show:

? succ Vi	? pred Lu	? enumFromTo Lu Do
Sa	Error	[Lu,Ma,Mi,Ju,Vi,Sa,Do]

Clausula “deriving” (2)

data Dia = Lu | Ma | Mi | Ju | Vi | Sa | Do
deriving (Eq, Ord, Enum, Show)

Funciones que usan operaciones heredadas:

festivo, laborable :: Dia –> Bool
festivo d = (d==Do)
laborable d = (d >= Lu) && (d <=Vi)

siguiente, anterior :: Dia –> Dia
siguiente d = if (d==Do) then Lu else succ d
anterior d = if (d==Lu) then Do else pred d

Secuencias aritméticas

- `[n..]` es la lista `[n,n+1,n+2,...`

- `[n..m]` es la lista `[n,n+1,n+2,...,limit]`

  siendo `limit` el menor número (entero o real) mayor o igual que `m`.

  - `[5..8]` ⇒ `[5,6,7,8]`
  - `[5.7..8.2]` ⇒ `[5.7,6.7,7.7,8.7]`
  - `[2..(-3.1)]` ⇒ `[]`

- `[n,m..]` es la lista `[n,m,m+i,m+2*i,...` siendo `i= m-n`

  El incremento `i` que puede ser positivo, negativo o cero.

  - `[1.2,3.4..]` ⇒ `[1.2,3.4,5.6,7.8,10.0,...`
  - `[1.2,(-3.4)..]` ⇒ `[1.2,-3.4,-8.0,-12.6,-17.2,...`
  - `[1,1..]` ⇒ `[1,1,1,...`

- `[n,m..m']` es la lista `[n,m,m+i,m+2*i,...,limit]` siendo
  - `i= m-n`
  - `limit` = mayor número (entero/real) menor o igual que `m'`.

```
[1,3..8] ⇒[1,3,5,7]
[1,3..9] ⇒ [1,3,5,7,9]

[1,-3..(-14)] ⇒[1,-3,-7,-11]
[1,-3..(-11)] ⇒ [1,-3,-7,-11]

[1,1..8] ⇒ [1,1,1,...
[1.5,1.5..8.5] ⇒ [1.5,1.5,1.5,...
```

Secuencias de elementos de un tipo enumerado

- Todo tipo `T` que sea instancia de la clase `Enum` cuenta con los métodos `succ` y `pred`, que hacen las veces de `(+1)` y `(-1)` respectivamente para construir secuencias de elementos de tipo `T`

- ```
  data Prueba = A | B | C | D | E | F
                deriving (Show,Enum)
  ```

```
[C..] ⇒ [C,D,E,F]
[A..D] ⇒ [A,B,C,D]
[A,B..] ⇒ [A,B,C,D,E,F]
[F,D..] ⇒ [F,D,B]
```
```

Tipos recursivos

```
data Nat =  Cero | Suc Nat
                    deriving  (Eq, Ord, Show)
```

Constructores: Cero :: Nat *y* Suc :: Nat –> Nat

Funciones definidas sobre Nat:

```
suma :: Nat –> Nat –> Nat
suma Cero y = y
suma (Suc x) y = Suc(suma x y)
```

```
aInt :: Nat –> Int
aInt Cero = 0
aInt (Suc x) = (aInt x) +1
```

```
? suma (Suc (Suc Cero)) (Suc Cero)
  Suc (Suc (Suc Cero))
```

```
? aInt (Suc (Suc Cero))
  2
```

Ejemplo: instancia de la clase Eq

```
infix :/
data MiRacional = Integer :/ Integer
```

- *Si* "**deriving** Eq" *entonces la igualdad es la estructural:*

```
? (4:/5) == (12:/15)
False
```

- *Declarando la instancia con la siguiente definición de* (==):

```
instance Eq MiRacional where
(x:/y) == (x':/y')  =  (x*y'== y*x')
```

se obtiene como resultado:

```
? (4:/5) == (12:/15)
True
```

Ejemplo: instancia de la clase Show

```
data MiRacional = Integer :/ Integer
```

- *Si añadimos* "**deriving** Show": ? 14:/21
  14:/21

- *Declarando la instancia con la siguiente definición de* **show**

```
instance Show MiRacional where
    show (x:/y) =  show (div x z) ++ ":/"++ show (div y z)
                   where z = mcd x y
```

se obtiene como resultado: ? 14:/21 | ? 14:/7
2:/3 | 2:/1

Ejercicio: mcd (máximo común divisor)

Tipos polimórficos y recursivos

data Lista α = Vac | Cons α (Lista α)
deriving (Eq, Ord, Show)

Constructores:

Vac :: Lista α Vac ≈ []

Cons :: α –> Lista α –> Lista α Cons ≈ (:)

Funciones polimórficas definidas sobre Lista α:

longitud :: Lista α –> Int

longitud Vac = 0

longitud (Cons x s) = 1+ longitud s

Patrones

- Los patrones más sencillos son las variables y las constantes (o *constructoras sin argumentos de un tipo de datos*)

```
cond :: Bool -> a -> a -> a
```

- ```
  cond b x y = if b then x else y
  ```
- ```
  cond b x y
       | b     = x       -- b == True
       | not b = y       -- b == False
  ```

- ```
  cond True x y = x
  cond False x y = y
  ```

  Definición con patrones

- Una definición con patrones esta formada por (una o) varias ecuaciones. Un parámetro formal es substituido en cada ecuación por un patrón diferente.
  ```

¿Que son patrones?

- Un ***patrón*** es
  - ***una variable o***
  - ***una constructora*** de un tipo de datos ***aplicada a tantos patrones como argumentos tenga*** (en particular, las ***constructoras constantes***).

- Ejemplos:
  - `x, y, z, f,`
  - `[], True, False,`
  - `(x:xs), (x:[]), [x], F(x,y,z), (x :/ y)`

- Desde el punto de vista del uso de patrones no hay ninguna diferencia entre que las constructoras usadas sean de un tipo polimórfico o monomórfico

- Los patrones numéricos son especiales

“No es lo mismo patrón que expresión”

No son definiciones correctas:

- `final (lis1 ++ lis2) = lis2`
- `f (n+k) = n`

(nótese que no tienen forma única de ajuste)

- `mitad (x+x) = x`
- `raizcuad (x*x) = x`

(si tienen forma única, pero `+` y `*` no son constructoras)

Ejemplos

```
f:: Int -> Respuesta -> Int
f x Si = x
f x No = (-x)
f x NoSabe = 0
```

```
Data Respuesta = Si | No | NoSabe
```

```
data Figura =  Circ Float | Rect Float Float
               deriving Show
area:: Figura -> Float
area (Circ rad) = pi * rad^2
area (Rect horz vert) = horz * vert
```


```
Main> area (Circ 2)
12.5664 :: Float
Main> area (Rect 2 4)
8.0 :: Float
```

- La siguiente función no es una constructora del tipo `Figura`:

```
crearcirc :: Float -> Figura
crearcirc a = Circ (sqrt (a/pi))
```

```
Main> crearcirc 25
Circ 2.82095 :: Figura
```

por tanto, no es correcto definir una nueva funcion f

```
f (crearcirc a) = <expresión>
```


- ```
  data Sec a = Vacia | Cons a (Sec a)
                  deriving Show

  inversa :: Sec a -> Sec a
  inversa Vacia = Vacia
  inversa (Cons x xs)
     = ponerUlt x (inversa xs)
       where ponerUlt y Vacia = Cons y Vacia
             ponerUlt y (Cons x xs)
                 = Cons x (ponerUlt y xs)
  ```

- ```
  Main> inversa  (Cons 1 (Cons 2 (Cons 3 Vacia)))
  Cons 3 (Cons 2 (Cons 1 Vacia)) :: Sec Integer
  ```

Ajuste de patrones

- El mecanismo de evaluación de expresiones de la forma

  `f exp1 ... expn`

  - `f` es el nombre de una función n-ária definida mediante patrones
  - `exp1 ... expn` son expresiones

  se denomina *ajuste de patrones:*

  - buscar (por orden) la primera ecuación de la forma

    `f p1 ... pn` = <parte-derecha>

    tal que para ciertos valores de las variables de los patrones

    `p1 ... pn`

    se tenga que `p1` ≡ `exp1` ... `pn` ≡ `expn`

  - dichos valores de las variables se usarán para evaluar la expresión <parte-derecha> (que define el resultado para f en ese caso de ajuste)
  - si ninguna ecuación cumple tal propiedad , se produce `error`
- Ejemplo: `inversa (Cons 1 (Cons 2 Vacia))`

Patrones numéricos vs condicional

- ```
  -- fact :: (Eq p, Num p) => p -> p
  fact 0 = 1
  fact x = x * fact (x-1)
  ```
  -- aplicada a un número negativo produce *evaluación infinita*

- ```
  -- fact' :: (Eq p, Num p) => p -> p
  fact' x = if x <= 0 then 1 else x * fact' (x-1)
  ```
  -- aplicada a un número negativo da 1

- ```
  -- fact'' :: (Num p, Ord p) => p -> p
  fact'' x
      | x == 0 = 1
      | x > 0  = x * fact'' (x-1)
  ```
  -- aplicada a un número negativo produce un `error`
  ```

Patrón anónimo (o “comodín”)

- En Haskell se puede usar el guión-bajo “_” como patrón que ajusta siempre.

- Una variable también es un patrón que ajusta siempre
  - pero la variable toma el valor con el que ajusta para usarlo en la parte derecha de la definición
  - cuando la parte derecha de la ecuación no usa el valor de dicha variable, entonces es más claro y más cómodo usar “_”

- Definición de los operadores lógicos

```
(&&), (||):: Bool -> Bool -> Bool
False && _ = False
True && b = b
True || _ = True
False || b = b
```

El ajuste de patrones es no-estricto

- El ajuste es no-estricto y se intenta “de-izquierda-a-derecha”
  - tanto si hay varios patrones, como si un patrón consta de varios argumentos

- Ejemplos
  - `(Cons 1 ⊥)` no ajusta con `(Cons 2 xs)`

    es decir falla el ajuste sin producir evaluación infinita y ni error
  - El ajuste del patrón `(Cons 2 xs)` con
    - `(Cons 2 ⊥)`
    - `(Cons ⊥ Vacia)`

    produce evaluación infinita/error

- Cuestión: ¿Como afecta esto a los operadores lógicos definidos en la página anterior?

Patrones anidados y lineales

- Se permiten patrones anidados:

```
quitar :: -- tipo?
quitar (x,y) [] = []
quitar (x,y) ((u,v):ps)
      |(x,y)==(u,v)  = quitar (x,y) ps
      | otherwise    = (u,v) : quitar (x,y) ps
```

- En los patrones de una misma ecuación no pueden aparecer variables repetidas. La siguiente definición es incorrecta (en compilación)

```
quitar (x,y) [] = []
quitar (x,y) ((x,y):ps) = quitar (x,y) ps
quitar (x,y) ((u,v):ps) = (u,v) : quitar (x,y) ps
```

- Los patrones sin variables repetidas se llaman patrones lineales

Esquema general de una ecuación

```
fun pat1 ... patn
          [| cond1  = exp1
           | cond2  = exp2
             .
             .
             .
           | condm ] = expm
                     [ where
                         deflocal1
                         .
                         .
                         .
                         deflocalk ]
```

No se pueden anidar ni guardas, ni where

Equivalencia con expresiones `case`

- Una definición con patrones de la forma:

```
f p11 p12 ... p1k = exp1
f p21 p22 ... p2k = exp2
⋮
f pn1 pn2 ... pnk = expn
```

es equivalente a

```
f x1 x2 ... xk = case (x1,x2,...,xk) of
                      (p11, p12,..., p1k)  -> exp1
                      (p21, p22,..., p2k)  -> exp2
                      ⋮
                      (pn1, pn2,..., pnk)  -> expn
```

Arboles binarios

data Arbin α = Hoja α | Unir (Arbin α) (Arbin α)

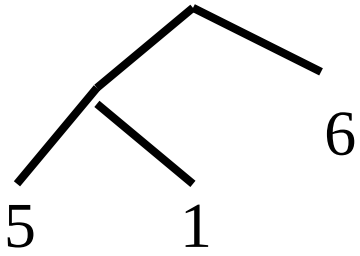
a1:: Arbin Int a2:: Arbin Char

a1 = Unir (Unir (Hoja 5) (Hoja 1)) (Hoja 6)

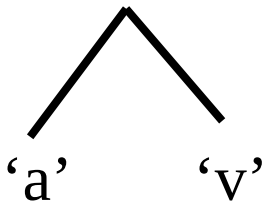
a2 = Unir (Hoja ‘a’) (Hoja ‘v’)

representan los árboles:

a1 =



a2 =



Funciones sencillas sobre Arbin α

prof :: Arbin α –> Int *(profundidad)*

prof (Hoja x) = 0

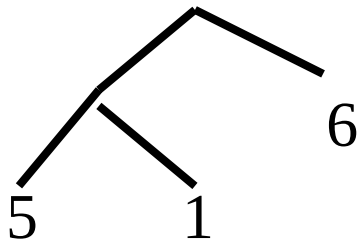
prof (Unir ai ad) = 1 + max (prof ai) (prof ad)

tamaño :: Arbin α –> Int *(número de hojas)*

tamaño (Hoja x) = 1

tamaño (Unir ai ad) = tamaño ai + tamaño ad

a1 =



? tamaño a1

3

? prof a1

2

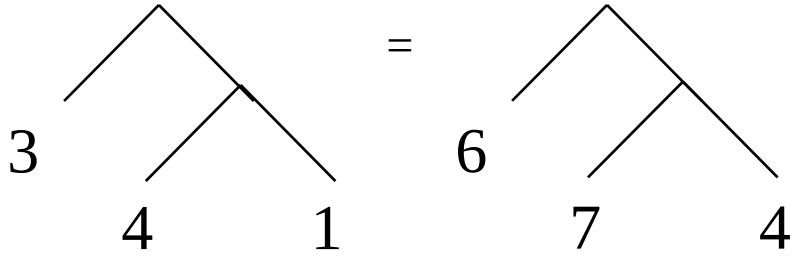
Función tipo “map” sobre Arbin α

maparbin :: (α–>β) –> Arbin α –> Arbin β

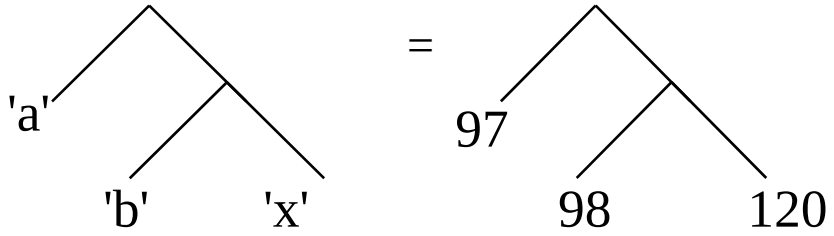
maparbin f (Hoja x) = Hoja (f x)

maparbin f (Unir ai ad) = Unir (maparbin f ai)(maparbin f ad)

maparbin (+3)



maparbin ord



Función tipo "fold" sobre Arbin α

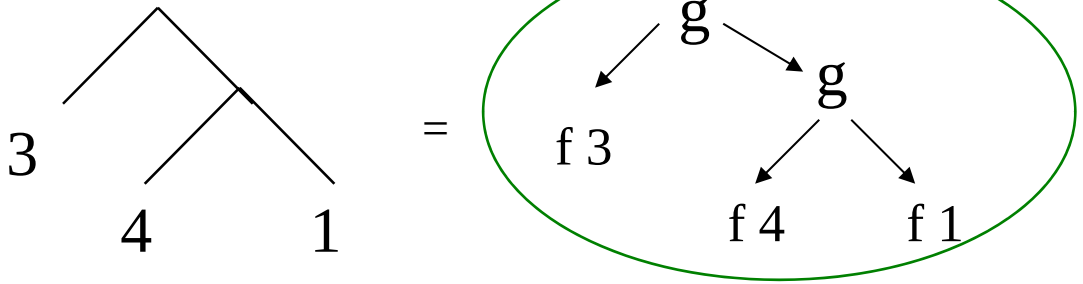
foldarbin :: (α–>β) –> (β–>β–>β) –> Arbin α –>β

foldarbin f g (Hoja x) = f x

foldarbin f g (Unir ai ad) = g (foldarbin f g ai) (foldarbin f g ad)

Idea:

foldarbin f g



= g (f 3) (g (f 4) (f 1))

tamaño = foldarbin (const 1) (+)

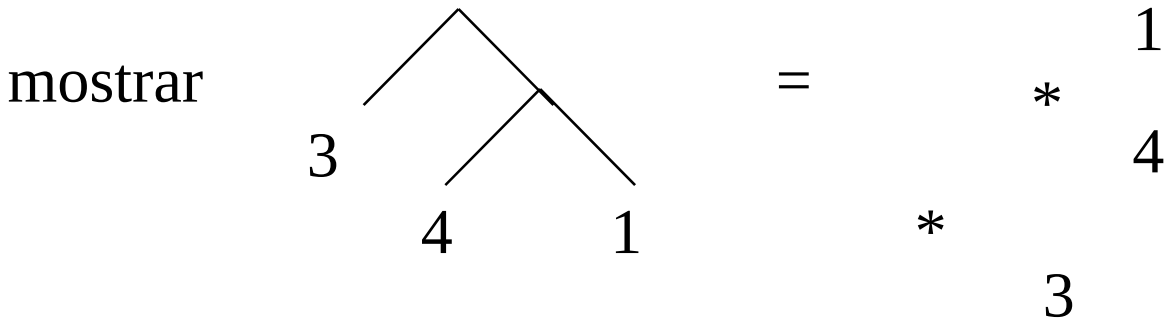
prof = foldarbin (const 0) g **where** g m n = 1+ max m n

Arbin α instancia de Eq y Show

data Arbin α = Hoja α | Unir (Arbin α) (Arbin α) **deriving** Eq

"La igualdad estructural es adecuada"

PERO si deseamos mostrar en pantalla los árboles de forma:



instance Show α => Show (Arbin α)
  where show = mostrar

en vez de **deriving** Show

Ejercicio: *definir la función* mostrar :: Show α => Arbin α –> String

Arboles binarios de búsqueda

data Arbus α = Vac | Nod (Arbus α) α (Arbus α)

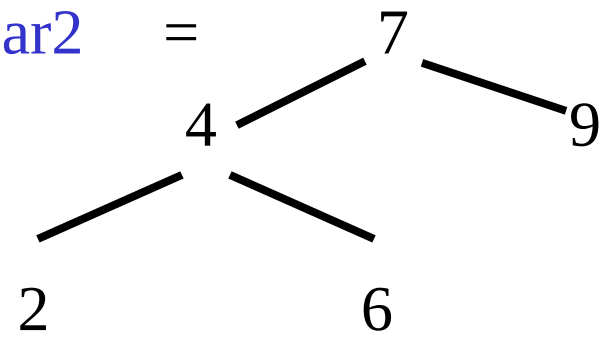
ar1, ar2:: Arbus Int ar3:: Arbus String

ar1 = Nod (Nod Vac 2 Vac) 4 (Nod Vac 6 Vac)

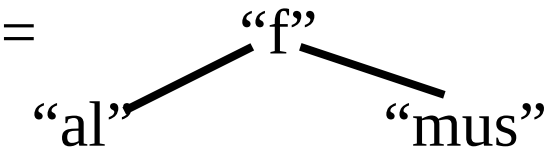
ar2 = Nod ar1 7 (Nod Vac 9 Vac)

ar3 = Nod (Nod Vac “al” Vac) “f” (Nod Vac “mus” Vac)

ar2 =



ar3 =



Funciones sobre Arbus α (1)

```
aplanar :: Arbus α –> [α]
aplanar Vac = []
aplanar (Nod ai r ad)   = aplanar ai ++ [r] ++ aplanar ad

estaOrd :: Ord α =>Arbus α –> Bool
estaOrd = ordenada • aplanar
```

? aplanar ar2	? estaOrd ar2
[2,4,6,7,9]	True

Ejercicio: definir la función ordenada :: Ord α => [α] –> Bool

Funciones sobre Arbus α (2)

```
esta :: Ord α => α –> Arbus α –> Bool
esta x Vac = False
esta x (Nod ai r ad)     | x<r = esta x ai
                         | x==r = True
                         | x>r = esta x ad

meter :: Ord α => α –> Arbus α –> Arbus α
meter x Vac = Nod Vac x Vac
meter x (Nod ai r ad)  | x<r  = Nod (meter x ai) r ad
                       | x==r = Nod ai r ad
                       | x>r   = Nod ai r (meter x ad)
```

Funciones sobre Arbus α (3)

```
borrar :: Ord α => α –> Arbus α –> Arbus α
borrar x Vac = Vac
borrar x (Nod ai r ad)      | x<r = Nod (borrar x ai) r ad
                            | x==r = une ai ad
                            | x>r = Nod ai r (borrar x ad)
```

donde la función une *debe cumplir la propiedad:*

aplanar (une ai ad) = aplanar ai ++ aplanar ad

(para ai, ad :: Arbus α *y con todos los nodos de* ai *menores que todos los de* ad*)*

Ejercicio: dar definiciones alternativas para la función une.

Tipo polimórfico predefinido “Maybe”

data Maybe α = Nothing | Just α
deriving (Eq, Ord, Read, Show)

Constructores: Nothing :: Maybe α
Just :: α –> Maybe α

Instancia (polimórfica) de las clases Eq, Ord, Show, Read

instance Eq α => **Eq** (Maybe α) .....

Ej: ? Just 3 == Just 5
False

? Nothing < Just 0
True

? Just 2
Just 2

Uso del tipo Maybe

Funciones predefinidas sobre Maybe :

```
maybe :: β –> (α –> β) –> Maybe α –> β
maybe n f Nothing  = n
maybe n f (Just x) = f x

lookup :: Eq α => α –> [(α,β)] –> Maybe β
lookup k []       = Nothing
lookup k ((x,y): res)
        | k==x            = Just y
        | otherwise       = lookup k res
```

Tipo polimórfico predefinido Either

data Either α β = Left α | Right β
deriving (Eq, Ord, Read, Show)

Constructores: Left :: α –> Either α β
Right :: β –> Either α β

Instancia (polimórfica) de las clases Eq, Ord, Show, Read

instance (Eq α, Eq β) => **Eq** (Either α β) .....

Ej: c, b:: Either Char Int c = Left 'j' b = Left 'x'

? c == b	? c < b	? c < Right 1
False	True	True

Uso del tipo Either

*Función **pre**definida sobre* Either :

either :: (α –> γ) –> (β –> γ) –> Either α β –> γ

either f g (Left x) = f x

either f g (Right y) = g y

Función definida usando la anterior

ord_long:: Either Char String –> Int

ord_long = either ord length

? ord_long (Left ‘a’)
  97

? ord_long (Right “ola marina”)
  10