



## Tema 5. Operadores sobrecargados

*Motivación de las clases de tipos*

? 3 == 4

False

? 'a' == 'a'

True

? doble == doble

Error

*¿tipo del operador (==)? ¿(==) ::  $\alpha \rightarrow \alpha \rightarrow \text{Bool}$  ?* **NO**

→ Hay una familia de tipos que usan (==) para describir la igualdad → (==) operador sobrecargado

*Lo mismo ocurre con otros operadores como (+), (>), ...*

? 3+4

7 :: Integer

? 3.0+4.6

7.6 :: Double

? 'a'+'b'

Error



## Clase de tipos Eq

*Colección de tipos que usan los operadores (==), (/=)*

<b>class Eq <math>\alpha</math> where</b> $(==), (/=) :: \alpha \rightarrow \alpha \rightarrow \text{Bool}$
$x /= y = \text{not } (x == y)$

→ Clase con dos funciones miembro (ó **métodos**)

$(==), (/=) :: \text{Eq } \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{Bool}$

 *restricción*

→ Definición por defecto de (/=) en términos de (==)



## Declarando instancias de la clase Eq

*Cada instancia de Eq define su operador (==)*

- Bool declarado como instancia de la clase Eq

```
instance Eq Bool where  
x == y = (x && y) || (not x && not y)
```

- $[\alpha]$  declarado como instancia (polimórfica) de Eq

```
instance Eq  $\alpha \rightarrow$  Eq  $[\alpha]$  where  
[] == []           = True  
(x:s) == (y:r)     = x==y && s==r  
_ == _            = False
```



## Instancias de la clase Eq

---

- Bool, Char, Int, Integer, Float, Double son instancias de Eq
- $[\alpha]$  es instancia de Eq siempre que lo sea  $\alpha$ 
  - [Bool] es instancia de Eq
  - String es instancia de Eq      **type** String = [Char]
  - [Int], [[Char]], ....
- $(\alpha, \beta)$  es instancia de Eq siempre que lo sean  $\alpha$  y  $\beta$ 
  - (Int, Bool) es instancia de Eq
- $(\alpha, \beta, \gamma)$  es instancia de Eq siempre que lo sean  $\alpha$ ,  $\beta$  y  $\gamma$ 
  - (Char, String, Int) es instancia de Eq
- etc .....  $([(Char, String)], Int)$  es instancia de Eq



## Operadores de orden

---

? 3 > 4	? 'a' >= 'a'	? "hola" < "k"	? True > False
False	True	True	True


- Hay una familia de tipos que usan los operadores de orden
  - operadores sobrecargados
  - Clase de tipos: **Ord**
- Los tipos instancias de Ord deben ser ya instancias de Eq  
(los tipos con orden deben tener igualdad)
  - Ord se declara como subclase de Eq



## Clase de tipos Ord (1)

*Colección de tipos que usan los operadores de orden*

*Clase de tipos **Ord** declarada como subclase de **Eq***



```
class Eq  $\alpha \Rightarrow$  Ord  $\alpha$  where  
  ( $\geq$ ), ( $\leq$ ), ( $>$ ), ( $<$ ) ::  $\alpha \rightarrow \alpha \rightarrow$  Bool  
  max, min ::  $\alpha \rightarrow \alpha \rightarrow \alpha$ 
```

Clase con varias funciones miembro

```
( $\geq$ ), ( $\leq$ ), ( $>$ ), ( $<$ ) :: Ord  $\alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow$  Bool  
max, min :: Ord  $\alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ 
```



## Clase de tipos Ord (2)

```
class Eq  $\alpha \Rightarrow$  Ord  $\alpha$  where  
  ( $\geq$ ), ( $\leq$ ), ( $>$ ), ( $<$ ) ::  $\alpha \rightarrow \alpha \rightarrow$  Bool  
  max, min ::  $\alpha \rightarrow \alpha \rightarrow \alpha$   
  
   $x \leq y = (x < y) \parallel (x == y)$   
   $x \geq y = (y \leq x)$   
   $x > y = (y < x)$   
  max x y = if  $y < x$  then x else y  
  min x y = if  $x \leq y$  then x else y
```

→ Definiciones por defecto de ( $\leq$ ), ( $\geq$ ), ( $>$ ), max, y min en términos de ( $<$ ) y de ( $==$ )



## Declarando instancias de la clase Ord

- Bool declarado como instancia de la clase Ord

```
instance Ord Bool where
```

```
False < True      = True
```

```
_ < _             = False
```

- [ $\alpha$ ] declarado como instancia polimórfica de Ord

```
instance Ord  $\alpha \rightarrow$  Ord [ $\alpha$ ] where
```

```
[] < (x:s)         = True
```

```
(x:s) < (y:r)       = (x<y) || ((x==y) && (s<r))
```

```
_ < _              = False
```





## Uso de operadores sobrecargados

Ej:  $f\ x\ y = \text{if } x \leq y \text{ then "si" else "no"}$

↓ *tipo inferido*

$f :: \text{Ord } \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow [\text{Char}]$

? f 3 4

“si”

? f “am” “adg”

“no”

? f 3 ‘a’

Error ....

? f head last

**ERROR**

\*\*\* El tipo de **head** (como el de **last**) es:  $[\beta] \rightarrow \beta$

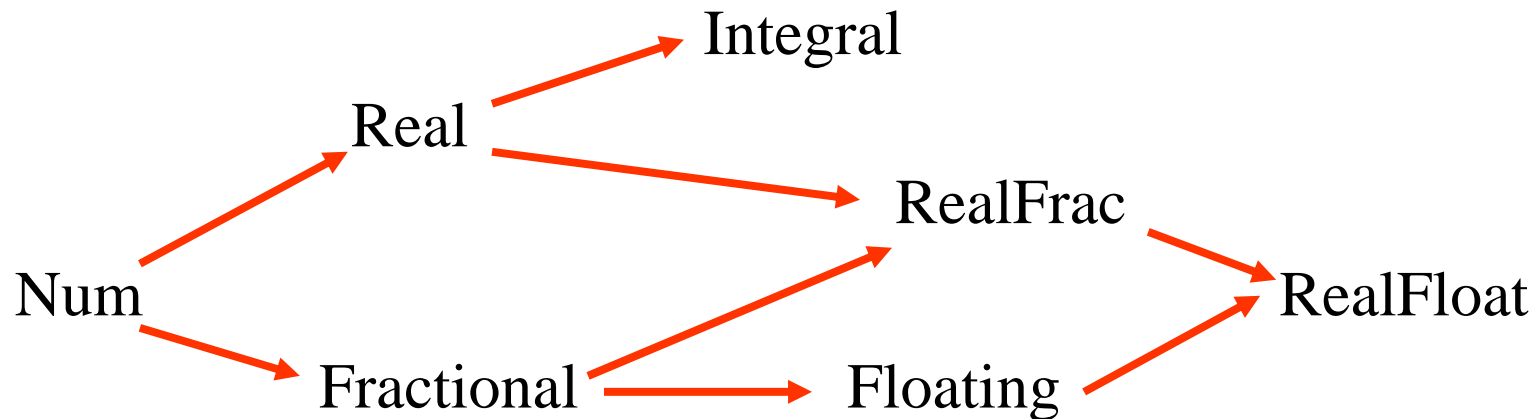
\*\*\* y este tipo no es una instancia de la clase **Ord**



## Clases para tipos numéricos

Los 4 tipos numéricos son `Int`, `Integer`, `Float` y `Double`

- `Int`, `Integer` son instancias de `Integral` ..... `Num`
- `Float`, `Double` son instancias de `RealFloat` ..... `Num`
- Jerarquía de las clases numéricas:





## La clase Num

---

**class** Num  $\alpha$  **where**

$(+)$  ::  $\alpha \rightarrow \alpha \rightarrow \alpha$

$(-)$  ::  $\alpha \rightarrow \alpha \rightarrow \alpha$

$(*)$  ::  $\alpha \rightarrow \alpha \rightarrow \alpha$

negate ::  $\alpha \rightarrow \alpha$

abs ::  $\alpha \rightarrow \alpha$

signum ::  $\alpha \rightarrow \alpha$

fromInteger :: Integer  $\rightarrow \alpha$

Para pedir información  
Prelude> :i Num

-- instancias: Int, Integer, Float, Double,



## La clase Integral

---

**class** (**Real**  $\alpha$ , **Enum**  $\alpha$ )  $\Rightarrow$  **Integral**  $\alpha$  **where**

$\text{quot} :: \alpha \rightarrow \alpha \rightarrow \alpha$

$\text{rem} :: \alpha \rightarrow \alpha \rightarrow \alpha$

$\text{div} :: \alpha \rightarrow \alpha \rightarrow \alpha$

$\text{mod} :: \alpha \rightarrow \alpha \rightarrow \alpha$

$\text{quotRem} :: \alpha \rightarrow \alpha \rightarrow (\alpha, \alpha)$

$\text{divMod} :: \alpha \rightarrow \alpha \rightarrow (\alpha, \alpha)$

$\text{toInteger} :: \alpha \rightarrow \text{Integer}$

-- instancias: Int, Integer

*En Prelude:*       $\text{even}, \text{odd} :: \text{Integral } \alpha \rightarrow \alpha \rightarrow \text{Bool}$       (*even – par*)



## Otras clases de tipos

---

- Enum para tipos enumerados
- Show para tipos con elementos “mostrables”
- Read para tipos con elementos “leibles”
- Bounded para tipos con valores acotados

*Son instancias (predefinidas) de*

- **Enum:** Char, Bool, Int, Integer, Float, Double
- **Show:** todos los tipos salvo las funciones ( $\rightarrow$ )
- **Read:** todos salvo funciones y tipo IO  $\alpha$
- **Bounded:** Char, Bool, Int



## La clase Enum

---

**class** Enum  $\alpha$  **where**

succ, pred ::  $\alpha \rightarrow \alpha$

toEnum :: Int  $\rightarrow \alpha$

fromEnum ::  $\alpha \rightarrow$  Int

enumFrom ::  $\alpha \rightarrow [\alpha]$  -- [n..]

enumFromThen ::  $\alpha \rightarrow \alpha \rightarrow [\alpha]$  -- [n,m..]

enumFromTo ::  $\alpha \rightarrow \alpha \rightarrow [\alpha]$  -- [n..m]

enumFromThenTo ::  $\alpha \rightarrow \alpha \rightarrow \alpha \rightarrow [\alpha]$  -- [n,n'..m]

-- *Minimal complete definition:* toEnum, fromEnum

-- de forma que **toEnum(fromEnum x) = x**



## La clase Show

---

**class** Show  $\alpha$  where

show ::  $\alpha \rightarrow$  String

showsPrec :: Int  $\rightarrow$   $\alpha \rightarrow$  ShowS

showList :: [ $\alpha$ ]  $\rightarrow$  ShowS      *type ShowS = String  $\rightarrow$  String*

*-- Minimal complete definition: show ó showsPrec*

- show 4 == ['4'] == "4"      show 'a' == ["','a','"] == "'a'"  
show "hola" == ["'",'h','o','l','a','"'] == "\"hola\""
- *El sistema aplica automáticamente putStr . show para mostrar en pantalla los resultados de las expresiones.*



## Ejemplo de uso de Show

---

```
type Direccion = (String, Int)
```

```
listaDir = [(“Aldapa”, 5), (“Nueva”, 14), ...] :: [Direccion]
```

*Definimos funciones para imprimir listas en pantalla:*

```
escribirDir :: Direccion -> String
```

```
escribirDir (calle, num) = calle ++ “, ” ++ show num ++ “\n”
```

```
escribirLista :: [Direccion] -> String
```

```
escribirLista = concat . map escribirDir
```

*Pregunta adecuada para imprimir el string en pantalla:*

```
? putStrLn (escribirLista listaDir)
```