Tema 5. Operadores sobrecargados

Motivación de las clases de tipos

? “ca” == “cas” ? ‘a’ == ‘a’ ? doble == doble

False True Error (doble es una función)

¿tipo del operador (==)? ¿ (==) :: α –> α –> Bool ? **NO**

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

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

? 3+4 ? 3.0+4.6 ? ‘a’+‘b’

7 :: Integer 7.6 :: Double Error

Una clase es una colección de tipos que comparten (con sobrecarga) ciertas operaciones llamadas métodos de la clase

Clase de tipos Eq

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

```
class Eq α  where
(==), (/=) :: α –> α –> Bool

-- Minimal complete definition: either '==' or '/='
x /= y = not (x == y)
x == y = not (x /= y)
```

La clase Eq tiene dos funciones miembro (ó métodos) con tipo:

(==), (/=) :: Eq α => α –> α –> Bool

restricción de clase

Definición por defecto de (/=) en términos de (==) y viceversa

Declarando instancias de la clase Eq

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

- Bool se puede declarar instancia de la clase Eq así:

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

- [α] está declarado instancia (polimórfica) de Eq así:

```
instance  Eq α =>  Eq [α] 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
- [α] es instancia de Eq siempre que lo sea α
  - [Bool] es instancia de Eq
  - String es instancia de Eq **type** String = [Char]
  - [Int], [[Char]], ...
- (α,β) es instancia de Eq siempre que lo sean α y β
  - (Int,Bool) es instancia de Eq
- (α,β,γ) es instancia de Eq siempre que lo sean α, β y γ
  - (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 α => Ord α  where
(<=), (>=), (>), (<) :: α –> α –> Bool
max, min :: α –> α –> α
compare :: :: α –> α –> Ordering
```

La clase Ord tiene varias funciones miembro con tipos:

```
(<=), (>=), (>), (<) :: Ord α => α –> α –> Bool
max, min :: Ord α => α –> α –> α
compare :: Ord α => α –> α –> Ordering
```

donde

```
data Ordering = LT | EQ | GT  (tipo algebraico predefinido)
```

Clase de tipos Ord (2)

class Eq α => Ord α **where**

(<=), (>=), (>), (<) :: α –> α –> Bool

max, min :: α –> α –> α

compare :: :: α –> α –> Ordering

-- minimal complete definition: either 'compare' or '<='

Definiciones por defecto de todos los métodos de Ord en términos de **(<=)** o bien en términos de compare (y de los métodos de Eq)

Declarando instancias de la clase Ord

- Bool se puede declarar instancia de la clase Ord así:

```
instance  Ord  Bool  where
x <=  y    =   (x == False || y == True)
```

- [α] se puede declarar instancia polimórfica de Ord así:

```
instance  Ord α =>  Ord [α] where
[] <= _              =  True
(x:s) <= (y:r)       = (x < y) || ((x==y) && (s <= r))
 _ <=  _             =  False
```

Uso de operadores sobrecargados

Ej: f x y = if x<=y then “si” else “no”

tipo inferido

f :: Ord α => α –> α –> [Char]

? f 3 4 “si”

? f “am” “adg” “no”

? f 3 ‘a’ Error ....

? f head last

ERROR

*** **head** y **last** tienen ambos el mismo tipo: **[β] –> β** (instancia de α)

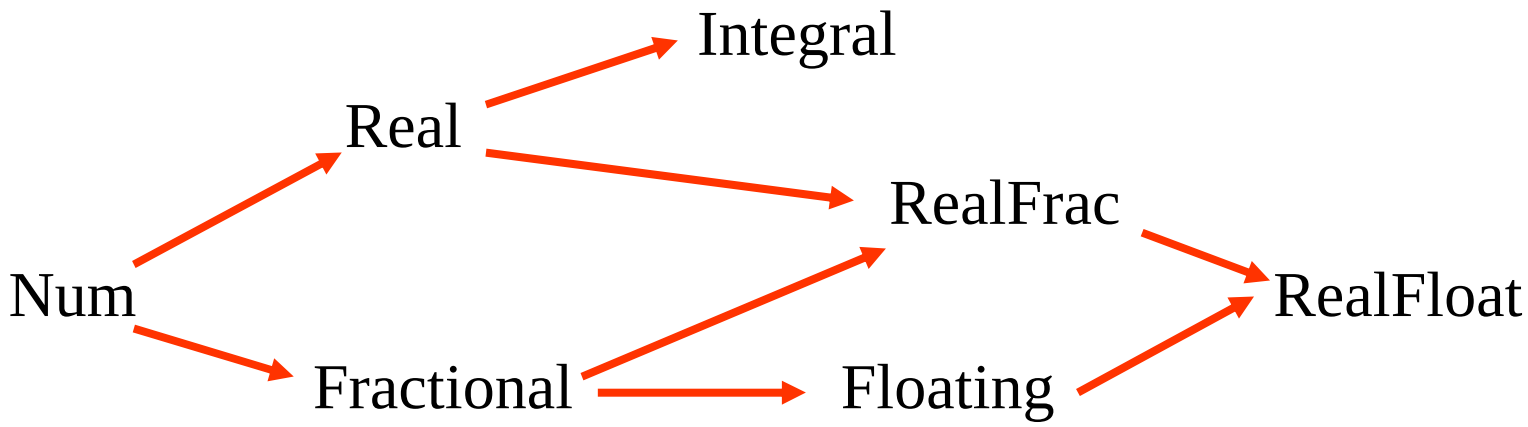
*** pero este tipo no pertenece a la clase **Ord**

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 α **where**

(+) :: α –> α –> α

(-) :: α –> α –> α

(*) :: α –> α –> α

negate :: α –> α

abs :: α –> α

signum :: α –> α

fromInteger :: Integer –> α

Para pedir información

Prelude> :i Num

-- instancias: Int, Integer, Float, Double

La clase Integral

class (Real α, Enum α) => Integral α **where**

quot :: α –> α –> α

rem :: α –> α –> α

div :: α –> α –> α

mod :: α –> α –> α

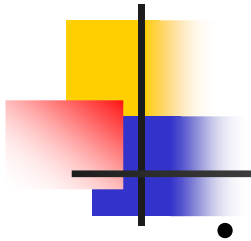
quotRem :: α –> α –> (α,α)

divMod :: α –> α –> (α,α)

toInteger :: α –> Integer

-- instancias: Int, Integer

En Prelude: even, odd :: Integral α => α –> 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 (–>)
- Read: todos salvo funciones y tipo IO α
- Bounded: Char, Bool, Int (con minBound, maxBound)

La clase Enum

```
class Enum α where
  succ, pred :: α –> α
  toEnum :: Int –> α
  fromEnum :: α –> Int
  enumFrom :: α –> [α]                          -- [n..]
  enumFromThen :: α –> α –> [α]                 -- [n,s..]
  enumFromTo :: α –> α –> [α]                   -- [n..m]
  enumFromThenTo :: α –> α –> α –> [α]          -- [n,s..m]
```

```
-- minimal complete definition: toEnum, fromEnum
--  de forma que   toEnum (fromEnum x) = x
```

La clase Show

class Show α **where**

show :: α –> String

showsPrec :: Int –> α –>ShowS

showList :: [α] –> ShowS type ShowS= String –> 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:

```
? putStr (escribirLista listaDir)
```