

Tema 7. El sistema de clases

7.1 Funciones Sobrecargadas

Clases e Instancias

7.2 Algunas Clases e Instancias predefinidas

La clase Eq

La clase Ord

Las clases Show y Read

Las clases Num y Fractional

7.3 Derivación de instancias

7.4 Tipos sobrecargados: Contextos

7.1 Funciones Sobrecargadas

- ✓ Tienen sentido para algunos tipos, pero no todos
- ✓ Pueden tener definiciones distintas para cada tipo

Ejemplo: Consideremos los tipos

```
type Lado      = Float
type Radio     = Float
type Área     = Float
data Cuadrado = UnCuadrado Lado deriving Show
data Círculo  = UnCírculo Radio deriving Show
```

Tiene sentido definir una función para calcular el área de un *Cuadrado*:

```
área          :: Cuadrado → Área
área (UnCuadrado l) = l * l
```

○ para un *Círculo*:

```
área          :: Círculo → Área
área (UnCírculo r) = pi * r ^ 2
```

¿ Es el tipo de *área* :: $a \rightarrow \text{Área}$?

NO, no tiene sentido, p. ej., calcular el área de un *Bool*

área tiene sentido para los tipos *Cuadrado* y *Círculo*, pero no para *Bool*, luego no es polimórfica

Clases e Instancias

✓ *área* tiene sentido para varios tipos pero *NO* para todos

✓ *área* tiene una definición *DISTINTA* para cada tipo

área es un ejemplo de función *Sobrecargada*

En Haskell, para definir función sobrecargada hay que crear una *clase* (conjunto de tipos que implementan la función):

```
class TieneÁrea t where
  área :: t → Área
```

✓ *TieneÁrea* es el nombre de la clase (empieza por mayúscula)

✓ *t* es una variable de tipo que representa los tipos de la clase

✓ El *método* *área* solo estará definido para los *t* que pertenezcan a la clase

Para incluir un tipo en una clase se realiza una *instancia*

```
instance TieneÁrea Cuadrado where
  área (UnCuadrado l) = l * l
```

```
instance TieneÁrea Círculo where
  área (UnCírculo r) = pi * r ^ 2
```

Uso:

```
? área (UnCuadrado 3)      -- Se usa primera instancia
9.0 :: Área
```

```
? área (UnCírculo 3)      -- Se usa segunda instancia
28.2743 :: Área
```

```
? área True               -- No existe instancia adecuada
ERROR ...
```

7.2 Algunas Clases e Instancias predefinidas

- ✓ Haskell organiza los tipos predefinidos en clases de tipos.

Clase: conjunto de tipos para los que tiene sentido una serie de operaciones sobrecargadas.

- ✓ Algunas de las clases predefinidas:
 - ◇ *Eq* tipos que definen igualdad: (`==`) y (`/=`)
 - ◇ *Ord* tipos que definen un orden: (`<=`), (`<`), (`>=`), ...
 - ◇ *Num* tipos numéricos: (`+`), (`-`), (`*`), ...

Instancias: conjunto de tipos pertenecientes a una clase.

- ✓ Algunas instancias predefinidas:
 - ◇ *Eq* tipos que definen igualdad: *Bool*, *Char*, *Int*, *Integer*, *Float*, *Double*, ...
 - ◇ *Ord* tipos que definen un orden: *Bool*, *Char*, *Int*, *Integer*, *Float*, *Double*, ...
 - ◇ *Num* tipos numéricos: *Int*, *Integer*, *Float* y *Double*

La clase Eq

✓ Tipos *igualables*

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

-- Mínimo a implementar: (==) o bien (/=)

```
x == y = not (x /= y)
x /= y = not (x == y)
```

- ✓ Las definiciones en la clase de (==) y (/=) son *Métodos por defecto*
- ✓ Se usan si no se definen en las instancias
- ✓ Basta con definir uno de los dos

La clase Ord

✓ Tipos ordenables

class *Ord* *a* **where**

$(<), (<=), (>=), (>) :: a \rightarrow a \rightarrow \text{Bool}$

$\text{max}, \text{min} :: a \rightarrow a \rightarrow a$

...

-- Mínimo a implementar: ($<=$)

$x < y = x <= y \ \&\& \ x \neq y$

$x >= y = y <= x$

$x > y = y < x$

$\text{max } x \ y$

| $x >= y = x$

| $\text{otherwise} = y$

$\text{min } x \ y$

| $x <= y = x$

| $\text{otherwise} = y$

...

- ✓ Cualquier tipo instancia de *Ord* debe ser instancia de *Eq* (no necesariamente lo contrario)

Las clases Show y Read

✓ Tipos mostrables

```
class Show a where
  show :: a → String
  ...
```

✓ Tipos leíbles

```
class Read a where
  ...

read :: Read a ⇒ String → a
read s = ...
```

Las clases Num y Fractional

```
class Num a where
  (+), (-), (*) :: a → a → a
  negate      :: a → a
  abs, signum :: a → a
  fromInteger :: Integer → a
```

-- Mínimo a implementar: todos, excepto *negate* o (-)

```
x - y = x + negate y
negate x = 0 - x
```

```
class Fractional a where
  (/) :: a → a → a
  recip :: a → a
  ...
```

```
recip x = 1 / x
x / y = x * recip y
...
```

7.3 Derivación de instancias

- ✓ La cláusula `deriving` permite generar instancias de ciertas clases pre-definidas de forma automática.
- ✓ Aparece al final de una declaración de tipo

```
data Color = Rojo | Amarillo | Azul | Verde deriving (Eq, Ord, Show, Read)
? Rojo == Verde
False :: Bool
? Rojo < Verde
True :: Bool
? show Rojo
"Rojo" :: String
? read "Rojo" :: Color
Rojo :: Color
```

- ✓ Al derivar la clase `Eq` se usa *igualdad estructural*:
 - ◇ Dos valores son iguales si tienen la misma forma

EJEMPLO:

```
infix 9 :/
data Racional = Integer :/ Integer deriving Eq
```

genera

```
instance Eq Racional where
  x :/y == x' :/y' = (x == x') && (y == y')
? 1 :/2 == 2 :/4
False :: Bool
```

- ✓ La igualdad estructural no es adecuada en este caso

Derivación de instancias (2)

EJEMPLO:

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

genera

```
instance Eq Nat where  
  Cero == Cero = True  
  Suc x == Suc y = (x == y)  
  _ == _ = False
```

✓ La igualdad estructural es adecuada en este caso

Al derivar la clase *Ord* se usa *orden estructural*.

- ◇ Es adecuado para tipos enumerados
- ◇ No es adecuado en todos los casos.

7.4 Tipos sobrecargados: Contextos

- ✓ Los métodos de una clase solo pueden ser usados con tipos instancia de dicha clase
- ✓ Ésto queda reflejado en el tipo de los métodos:

? :t (==)

(==) :: Eq a => a -> a -> Bool

? :t (+)

(+) :: Num a => a -> a -> a

? True + False

ERROR - Illegal Haskell 98 class constraint in inferred type

*** Expression : True + False

*** Type : Num Bool => Bool

- ✓ El *contexto* establece una restricción sobre el polimorfismo de la variable.
- ✓ *Propagación de contextos:*

Si función polimórfica usa una sobrecargada se convierte en sobrecargada.

doble :: Num a => a -> a

doble x = x + x

elem :: Eq a => a -> [a] -> Bool

x 'elem' [] = False

x 'elem' (y : ys) = x == y || x 'elem' ys

f :: (Ord a, Num a) => a -> a -> a

f x y

| x < y = x

| otherwise = x + y

Objetivos del tema

El alumno debe:

- ✓ Comprender el concepto de sobrecarga y diferenciarlo del concepto de polimorfismo
- ✓ Saber definir clases e instancias para sobrecargar funciones
- ✓ Conocer las clases predefinidas expuestas
- ✓ Conocer la posibilidad de derivar instancias.
- ✓ Conocer la igualdad estructural generado para instancias derivadas
- ✓ Saber si una instancia derivada es adecuada
- ✓ Entender los tipos sobrecargados (*contextos*)
- ✓ Saber cuál es el tipo de una función sobrecargada a partir de su definición