El sistema de clases de Haskell

Introducción

- Función monomórfica: sólo se puede usar para valores de un tipo concreto

$$\begin{aligned} &not &&:: Bool \to Bool \\ &not\ True &&= False \\ &not\ False &&= True \end{aligned}$$

- Función polimórfica: se puede usar sobre una familia completa de tipos (aquella que representa el tipo polimórfico)

$$\begin{aligned} &id &&:: a \to a \\ &id\ x &&= x \\ &length &&:: [a] \to Int \\ &length[\,] &&= 0 \\ &length(x : xs) &&= 1 + length\ xs \end{aligned}$$

$$\begin{aligned} &id\ 1 \\ \Longrightarrow\ &\text{! definición de } id \\ &1 \end{aligned}$$

$$\begin{aligned} &id\ True \\ \Longrightarrow\ &\text{! definición de } id \\ &True \end{aligned}$$

✓ Funciones válidas para cualquier tipo a.

✓ Siempre se computan igual (definición única).

- Función sobrecargada:
  - ✓ pueden usarse para más de un tipo, pero con restricciones (**contextos**)
  - ✓ las definiciones (código) pueden ser distintas para cada tipo

EJEMPLO: El operador $(+)$ de Haskell

- puede usarse para sumar enteros
  $1 + 2 \Longrightarrow 3$
- puede usarse para sumar reales
  $1.5 + 2.5 \Longrightarrow 4.0$

- NO puede usarse para sumar booleanos
  $True + False$ "no tiene sentido, no está definido"

▶ El tipo NO puede ser polimórfico $(+) :: a \rightarrow a \rightarrow a$, ya que

NO se suman del mismo modo valores enteros, reales o racionales

▶ Deben existir definiciones distintas

  ✓ En HASKELL el tipo de la suma es $(+) :: \underbrace{Num\ a}_{contexto} \Rightarrow a \rightarrow a \rightarrow a$

  ✓ Solo se pueden sumar tipos numéricos

Clases e Instancias

HASKELL organiza los distintos tipos 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 $(\neq)$
- Ord tipos que definen un orden: $(\leq)$, $(<)$, $(\geq)$, ...
- 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 $Double$

Declaraciones de clases e instancias

- Definición de la clase predefinida Eq

$$\begin{array}{l}\textbf{class } Eq\ a \textbf{ where}\\ \quad (==) :: a \to a \to Bool\\ \quad (\neq) :: a \to a \to Bool\end{array}$$

  - ✓ Las operaciones de una clase se llaman **métodos**
  - ✓ La definición de clase especifica la signatura (tipos) de los métodos
  - ✓ El nombre de la clase comienza por mayúscula

- Para especificar que un tipo implementa los métodos de una clase se usa una definición de instancia:

$$\begin{array}{l}\textbf{instance } Eq\ Bool \textbf{ where}\\ \quad True == True = True\\ \quad False == False = True\\ \quad \_ == \_ = False\\ \\ \quad x \neq y = not(x == y)\end{array}$$

$$\begin{array}{l}\textbf{data } Racional = Integer :/ Integer \textbf{ deriving} Show\\ \\ \textbf{instance } Eq\ Racional \textbf{ where}\\ \quad (x :/ y) == (x' :/ y') = (x * y' == y * x')\\ \\ \quad r \neq r' = not(r == r')\end{array}$$

Métodos por defecto

- Se definen en la clase:

$$\begin{aligned}&\textbf{class } Eq\ a \textbf{ where}\\ &\quad (==) :: a \to a \to Bool\\ &\quad (\neq) :: a \to a \to Bool\\ &\\ &\quad x == y = not\,(x \neq y)\\ &\quad x \neq y = not\,(x == y)\end{aligned}$$

  ✓ Para realizar una instancia de Eq basta con definir $(==)$ o $(\neq)$.

  ✓ Al menos hay que definir una (definiciones mutuamente recursivas).

$$\begin{aligned}&\textbf{instance } Eq\ Bool \textbf{ where}\\ &\quad True == True = True\\ &\quad False == False = True\\ &\quad \_ == \_ = False\end{aligned}$$

- Un método por defecto se usa si no aparece en la instancia:

$$\begin{aligned}&\underline{True \neq True}\\ \Longrightarrow\ &\text{! Método por defecto}\\ &not\,(\underline{True == True})\\ \Longrightarrow\ &\text{! Método en instancia}\\ &\underline{not\,(True)}\\ \Longrightarrow\ &\text{! Definción de } not\\ &False\end{aligned}$$

Subclases

- La clase predefinida Ord es **subclase** de la clase Eq:

$$\begin{array}{l}
\textbf{class } Eq\ a \Rightarrow Ord\ a \textbf{ where} \\
\quad \begin{array}{lll}
compare & :: & a \to a \to Ordering \\
(<),\ (\leq),\ (\geq),\ (>) & :: & a \to a \to Bool \\
max,\ min & :: & a \to a \to a
\end{array} \\
\quad \begin{array}{lll}
compare\ x\ y \mid x == y & = & EQ \\
\qquad\qquad \mid x \leq y & = & LT \\
\qquad\qquad \mid otherwise & = & GT
\end{array} \\
\quad x \leq y = compare\ x\ y \neq GT \\
\quad \ldots \\
\textbf{data } Ordering = LT \mid EQ \mid GT \textbf{ deriving } (Eq,\ Ord,\ Ix,\ Enum,\ Read,\ Show,\ Bounded)
\end{array}$$

✓ Toda instancia de la subclase debe ser instancia de la clase padre: es un error de compilación incluir un tipo en Ord sin incluirlo en Eq.

✓ Lo contrario es posible: hacer un tipo instancia tan solo de Eq.

✓ Una subclase tiene visibilidad sobre los métodos de la clase padre: se pueden usar los métodos de Eq en los métodos por defecto de Ord

Instancias paramétricas

- Permiten definir un conjunto de instancias estableciendo condiciones

✓ si un tipo a dispone de la igualdad, es deseable que el tipo $[a]$ disponga también de una igualdad:

$$\textbf{instance } Eq\ a \Rightarrow Eq\ [a] \textbf{ where}$$

$$\begin{array}{lll} [\,] & == [\,] & = True \\ (x : xs) & == (y : ys) & = x == y \ \&\& \ xs == ys \\ \_ & == \_ & = False \end{array}$$

✓ una única declaración genera varias instancias: $[Int]$, $[Integer]$, $[Float]$, $[Double]$, $[Char]$, ...

✓ no todas las listas son instancias de Eq, solo aquellas cuyos elementos son instancias de Eq

Algunas clases predefinidas

- Tipos con igualdad

class $Eq\ a$ **where**

$$(==) :: a \to a \to Bool$$
$$(\neq) :: a \to a \to Bool$$

$--$ Mínimo a implementar: $(==)$ o bien $(\neq)$

$$x == y = not(x \neq y)$$
$$x \neq y = not(x == y)$$

- Tipos con orden

class $Eq\ a \Rightarrow Ord\ a$ **where**

$$compare :: a \to a \to Ordering$$
$$(<), (\leq), (\geq), (>) :: a \to a \to Bool$$
$$max, min :: a \to a \to a$$

$--$ Mínimo a implementar: $(\leq)$ o $compare$

$$\begin{aligned} compare\ x\ y \mid x == y &= EQ \\ \mid x \leq y &= LT \\ \mid otherwise &= GT \end{aligned}$$

$$\begin{aligned} x \leq y &= compare\ x\ y \neq GT \\ x < y &= compare\ x\ y == LT \\ x \geq y &= compare\ x\ y \neq LT \\ x > y &= compare\ x\ y == GT \end{aligned}$$

$$\begin{aligned} max\ x\ y \mid x \geq y &= x \\ \mid otherwise &= y \\ min\ x\ y \mid x \leq y &= x \\ \mid otherwise &= y \end{aligned}$$

data $Ordering = LT \mid EQ \mid GT$
deriving $(Eq, Ord, Ix, Enum, Read, Show, Bounded)$

- Tipos mostrables

```
type ShowS = String → String

class Show a where

  show      :: a → String
  showsPrec :: Int → a → ShowS
  showList  :: [a] → ShowS

  -- Mínimo: show o showsPrec

  show x          = showsPrec 0 x ""
  showsPrec _ x s = show x ++ s
  ...
```

- Tipos "leíbles"

```
type ReadS a = String → [(a, String)]

class Read a where

  readsPrec :: Int → ReadS a
  readList  :: ReadS [a]

  -- Mínimo: readsPrec

  ...

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

- Tipos numéricos

```
class (Eq a, Show a) ⇒ Num a where

  (+), (−), (∗) :: a → a → a
  negate        :: a → a
  abs, signum   :: a → a
  fromInteger   :: Integer → a

  -- Mínima definición: todos, excepto negate o (−)

  x − y      = x + negate y
  negate x = 0 − x


class (Num a) ⇒ Fractional a where

  (/)            :: a → a → a
  recip          :: a → a
  fromRational   :: Rational → a
  fromDouble     :: Double → a

  -- Mínima definición: fromRational y ((/) o recip)

  recip x      =1/x
  fromDouble =fromRational . toRational
  x/y          =x ∗ recip y
```

Derivación de instancias

- La claúsula **deriving** permite generar instancias de ciertas clases predefinidas de forma automática.

✓ Aparece al final de la declaración de tipo

```
data Color = Rojo | Amarillo | Azul | Verde deriving (Eq, Ord, Show)
```

```
MAIN> Rojo==Verde
False :: Bool
MAIN> Rojo < Verde
True :: Bool
```

```
MAIN> show Rojo
"Rojo" :: String
```

▶ Las instancias generadas usan igualdad estructural: Dos valores son iguales si tienen la misma forma

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

genera

```
instance Eq Racional where
  x :/ y == x' :/ y' = (x == x') && (y == y')
```

La igualdad estructural no es adecuada en este caso

```
MAIN> 1 :/ 2 == 2 :/ 4
False :: Bool
```

```
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 para la clase Ord se usa orden estructural:

- un dato es menor que otro si su constructor de datos aparece más a la izquierda en la definición
- para dos datos con el mismo constructor, se comparan sus argumentos de izquierda a derecha

$$\textbf{data}\ Nat\ =\ Cero \mid Suc\ Nat\ \textbf{deriving}\ (Eq,\ Ord)$$

genera las siguiente instancia de orden

$$\textbf{instance}\ Ord\ Nat\ \textbf{where}$$

$$\begin{aligned} Cero\ &\leq \_ &&= True \\ Suc\ x &\leq Suc\ y &&= x \leq y \\ \_\ &\leq \_ &&= False \end{aligned}$$

Tipos sobrecargados: Contextos

- Los métodos definidos en una clase solo pueden ser usados con tipos que sean instancia de la clase
  - ✓ Queda reflejado en el tipo de los métodos:

```
MAIN> :t (==)
(==) :: Eq a => a -> a -> Bool
MAIN> :t (+)
(+) :: Num a => a -> a -> a
```

  - ✓ El **contexto** establece una restricción sobre el polimorfismo de la variable.
  - ✓ Cuando una función polimórfica usa una función 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
                    ---------------
                      contexto
f x y
  | x < y     = x
  | otherwise = x + y
```