Programación Funcional

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

Type Classes

¿Por qué Sobrecarga?

- Supongamos que queremos definir una función que determine si un *Bool* pertenece a una lista:

$$
\begin{aligned}
&elemBool :: Bool \to [Bool] \to Bool \\
&elemBool \ \_ \ [\,] \quad\quad = False \\
&elemBool \ x \ (y : ys) = eqBool \ x \ y \ || \ elemBool \ x \ ys
\end{aligned}
$$

- lo mismo para *Int*

$$
\begin{aligned}
&elemInt :: Int \to [Int] \to Bool \\
&elemInt \ \_ \ [\,] \quad\quad = False \\
&elemInt \ x \ (y : ys) = eqInt \ x \ y \ || \ elemInt \ x \ ys
\end{aligned}
$$

- y para otros tipos.

¿Por qué Sobrecarga? ¿Polimorfismo?

- Se puede resolver usando polimorfismo y funciones de alto orden

$$
\begin{aligned}
&\mathit{elemGen} :: (a \to a \to \mathit{Bool}) \to a \to [a] \to \mathit{Bool} \\
&\mathit{elemGen}\ \_\ \ \_\ [\,] \qquad\quad = \mathit{False} \\
&\mathit{elemGen}\ \mathit{eq}\ x\ (y : \mathit{ys}) = \mathit{eq}\ x\ y \mid\mid \mathit{elemGen}\ \mathit{eq}\ x\ \mathit{ys}
\end{aligned}
$$

- Cada vez que uso *elemGen* tengo que pasar explícitamente la función que implementa la igualdad.

Sobrecarga - Type Classes

- Usando **type classes** se implementa

$$
\begin{aligned}
&elem :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool \\
&elem\ \_\ [\,] \quad\quad\ = False \\
&elem\ x\ (y : ys) = x == y\ ||\ elem\ x\ ys
\end{aligned}
$$

  donde el tipo a se restringe a los tipos que tienen igualdad.

- Es decir que existe una **instancia** de Eq

$$
\begin{aligned}
&\text{class } Eq\ a \text{ where} \\
&\quad (==) :: a \rightarrow a \rightarrow Bool \\
&\quad (/=) :: a \rightarrow a \rightarrow Bool
\end{aligned}
$$

Definiciones por defecto e Instancias

- En realidad *Eq* se define:

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

- Las definiciones **por defecto** se pueden sobreescribir al definir una instancia
- Por lo que basta definir una operación para declarar que un tipo es miembro de la clase

$$
\begin{aligned}
&\text{data } ConIgual = Mismo \mid Mismisimo\\
&\text{instance } Eq\ ConIgual \text{ where}\\
&\quad Mismo\ \ == Mismo\ \ = True\\
&\quad Mismisimo == Mismisimo = True\\
&\quad \_\ \ == \_\ \ = False
\end{aligned}
$$

Herencia

- Se pueden declarar clases que dependen de que sus tipos pertenezcan a otras clases

$$\begin{aligned}&\text{class } Eq\ a \Rightarrow Ord\ a \text{ where}\\ &\quad (<), (\leqslant), (>), (\geqslant) :: a \to a \to Bool\\ &\quad max, min \quad\quad :: a \to a \to a\\ &\quad compare \quad\quad :: a \to a \to Ordering\end{aligned}$$

$$\text{data } Ordering = LT \mid EQ \mid GT$$

- Para que un tipo pertenezca a *Ord* debe proveer definiciones de las operaciones de *Ord* y de *Eq*.

- La clase *Ord* cuenta con definiciones por defecto para todas las operaciones. Al menos hay que dar una definición de *compare* o $(\leqslant)$.

Restricciones en las Instancias

- Las instancias también pueden depender de que los tipos pertenezcan a otras clases

$$\begin{aligned}&\text{instance } (Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b) \text{ where}\\ &\quad (x, y) == (z, w) = x == z \;\&\&\; y == w\end{aligned}$$

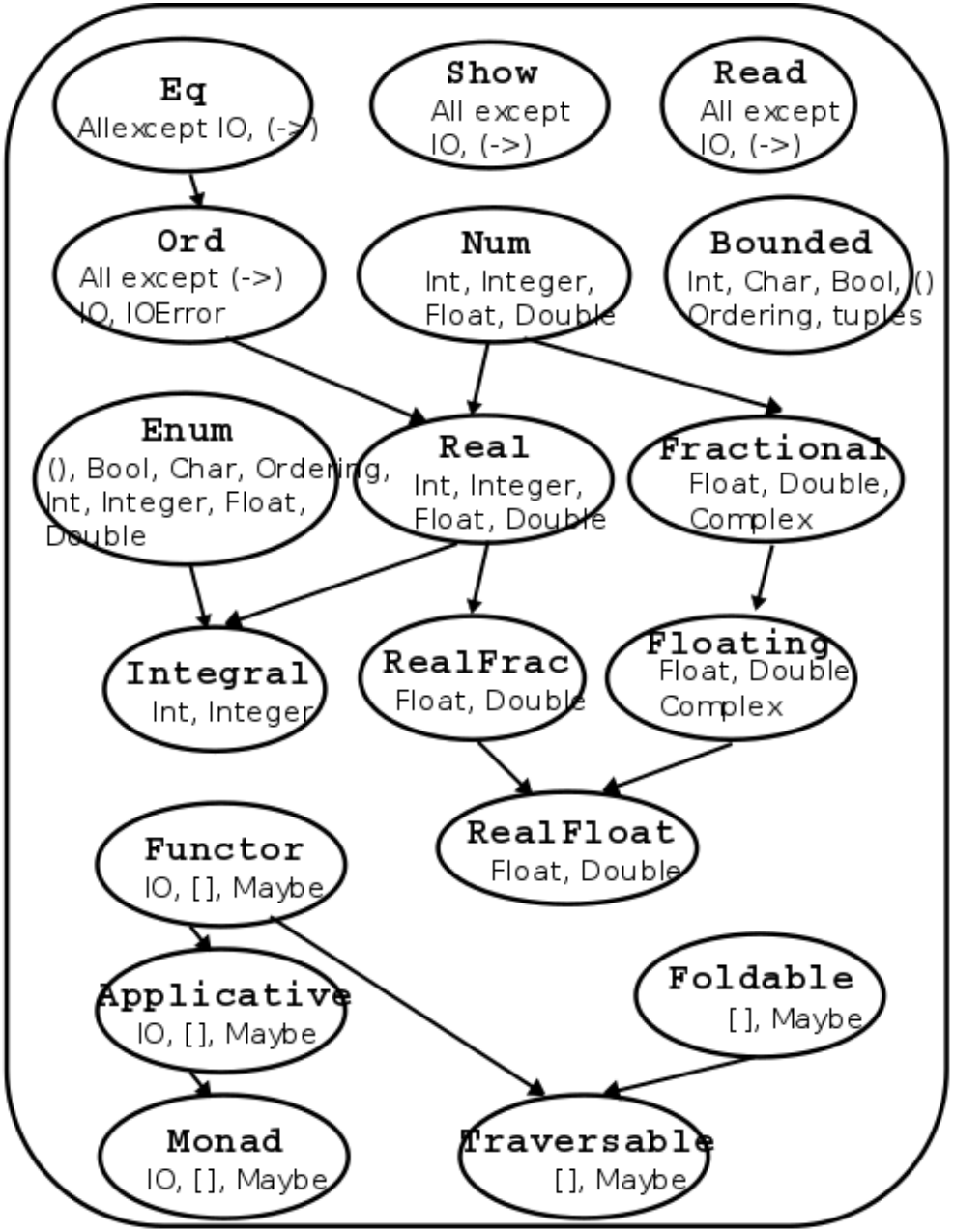
- Se permiten restricciones múltiples tanto en instancias, clases y funciones.

- Por ejemplo

$$\begin{aligned}&myPEq :: (Eq\ a, Eq\ b) \Rightarrow (a, b) \rightarrow (a, b) \rightarrow Bool\\ &myPEq\ (x, y)\ (z, w) = (x, y) == (z, w)\end{aligned}$$

Preguntas

Clases en Haskell 2010



Enumerados

- Clase de los tipos **Enumerados**

```
class Ord a ⇒ Enum a where
  succ, pred        :: a → a
  toEnum            :: Int → a
  fromEnum          :: a → Int
  enumFrom          :: a → [a]              -- [n .. ]
  enumFromThen      :: a → a → [a]          -- [n,m .. ]
  enumFromTo        :: a → a → [a]          -- [n .. m]
  enumFromThenTo :: a → a → a → [a]         -- [n,n' .. m]
```

- Nos permite definir listas como

$$
\begin{array}{ll}
[1\,..\,10] & \rightsquigarrow\ [1,2,3,4,5,6,7,8,9,10] \\
[2,4\,..\,10] & \rightsquigarrow\ [2,4,6,8,10] \\
\mathit{take}\ 10\ [1\,..] & \rightsquigarrow\ [1,2,3,4,5,6,7,8,9,10] \\
\texttt{['a'..'d']} & \rightsquigarrow\ \texttt{['a','b','c','d']}
\end{array}
$$

Numéricos

- Clase de los tipos **Numéricos**

  class *Num a* where

  $$\begin{aligned}(+), (-), (*) &:: a \to a \to a \\ negate &:: a \to a \\ abs &:: a \to a \\ signum &:: a \to a \\ fromInteger &:: Integer \to a\end{aligned}$$

- Como mínimo hay que definir
  $(+), (*)$, *abs*, *signum*, *fromInteger* y *negate* o $(-)$.

Convertir a *String* "legible"

- Clase *Show*

$$\text{type } ShowS = String \to String$$

$$\begin{aligned}&\text{class } Show\ a \text{ where}\\ &\quad showsPrec :: Int \to a \to ShowS\\ &\quad show \quad\quad :: a \to String\\ &\quad showList \quad :: [a] \to ShowS\end{aligned}$$

- Como mínimo hay que definir *show* o *showsPrec*.

Leer de un *String*

- Clase *Read*

$$\text{type } ReadS\ a = String \rightarrow [(a, String)]$$

$$\begin{aligned}&\text{class } Read\ a \text{ where}\\ &\quad readsPrec :: Int \rightarrow ReadS\ a\\ &\quad readList \quad :: ReadS\ [a]\end{aligned}$$

- Como mínimo hay que definir *readsPrec*.
- La función $read :: Read\ a \Rightarrow String \rightarrow a$ está definida en top-level y utiliza *readsPrec*.