# El Sistema de Tipos de Haskell: Tipos y Clases

## Tecnología de Programación

**Adolfo Muñoz – Juan Magallón**

**Grado en Ingeniería Informática**

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

2023-01-20

# Introducción

El objetivo de este tema es presentar:

- la organización de los tipos básicos de Haskell
- la forma de ampliar ese conjunto de tipos

El **Sistema de Tipos** de un Lenguaje es el conjunto de reglas que permite comprender el funcionamiento de los tipos de datos en un lenguaje y la forma de ampliar ese conjunto de tipos, es decir, definir nuevos tipos de datos.

Un renombramiento, sinónimo o *alias* de tipo introduce un nombre nuevo para un tipo ya existente:

```
1  type Entero    = Integer
2  type Punto     = (Entero,Entero)
3  type Racional  = (Entero,Entero)
4
5  a :: Punto       a = (2,3)
6  b :: Racional    b = a
```

Sólo son sinónimos, no definen ningún tipo nuevo y no permiten realizar ningún control de tipos adicional.

Hemos visto que la definición de tipos enumerados es sencilla:

```
1   data Dia   = Lun | Mar | Mie | Jue | Vie | Sab | Dom
2   data State = ON | OFF
```

El ajuste de patrones se basa en las palabras clave del propio tipo enumerado:

```
1  data Dia   = Lun | Mar | Mie | Jue | Vie | Sab | Dom
2
3  esFinde :: Dia -> Bool
4  esFinde Sab = True
5  esFinde Dom = True
6  esFinde _   = False
```

```
data Dia   = Lun | Mar | Mie | Jue | Vie | Sab | Dom
data State = ON | OFF
```

A las palabras clave que definen los valores posibles del tipo las denominamos **contructores de datos**.
Podemos interpretarlas como funciones sin parámetros que generan un valor concreto del tipo:

```
hoy = Lun
estadoLavadora = OFF
```

Los constructores de datos pueden tener parámetros de otros tipos, y el dato construido contiene los valores de esos parámetros:

```
1   data Dummy       = OnlyMe
2   data Temperature = Degrees Float
3   data Fraction    = Frac Integer Integer
4   data Point       = Pnt Integer Integer
5
6   origen  = Pnt 0 0
7   unMedio = Frac 1 2
```

A los tipos definidos de esta forma se les denomina **tipos de datos algebraicos** (por razones que veremos después).

Constructores

La definición de un tipo nuevo de datos en Haskell se realiza mediante un *constructor de tipo* y *constructores de datos*.
Los constructores de datos pueden tener parámetros de otros tipos, y el dato construido contiene los valores de esos parámetros:

```haskell
data Dummy       = OnlyMe
data Temperature = Degrees Float
data Fraction    = Frac Integer Integer
data Point       = Pnt Integer Integer
```

# Constructores

```haskell
data Dummy       = OnlyMe
data Temperature = Degrees Float
data Fraction    = Frac Integer Integer
data Point       = Pnt Integer Integer
```

```haskell
p   :: Point
num :: Fraction -> Integer
t   :: Temperature
```

```haskell
-- Legal
p = Pnt 3 4
t = Degrees 25.0
-- Ilegal
num p
t = 25.0
```

En lenguajes orientados a objetos, casi siempre definimos constructores que dan valor a todos los atributos de una clase.

```
class Fraction {
    int numerator, denominator;
public:
    Fraction(int n, int d) :
        numerator(n), denominator(d) {}
};

Fraction f(3,4);
```

# Constructores

Haskell formaliza esa idea: el propio constructor de datos define los atributos del tipo de datos.

```haskell
data Fraction = Frac Integer Integer


f :: Fraction
f = Frac 3 4
```

El ajuste de patrones permite acceder a los datos asociados a las componentes de un tipo de datos algebraico:

```
1   data Fraction = Frac Integer Integer
2
3   numerator :: Fraction -> Integer
4   numerator (Frac n _) = n
5
6   denominator :: Fraction -> Integer
7   denominator (Frac _ d) = d
8
9   mul :: Fraction -> Fraction -> Fraction
10  mul (Frac a b) (Frac c d) = Frac (a*c) (b*d)
```

Podemos también asociar un nombre a cada uno de los parámetros de un constructor:

```
data Fraction =
    Frac { numerator::Integer, denominator::Integer }

f :: Fraction
f = Frac { denominator=3, numerator=4 }

print (numerator f)
```

Los nombres de los parámetros automáticamente definen funciones para acceder a los campos.

Podemos también asociar un nombre a cada uno de los parámetros de un constructor:

```
1  data Point = Pnt { xval::Float, yval::Float }
2
3  p = Pnt 4 3
4  q = Pnt { yval=3, xval=4 }
```

Los nombres de los parámetros automáticamente definen funciones para acceder a los valores...

```haskell
data Point = Pnt { xval::Float, yval::Float }

distance :: Point -> Point -> Float
distance a b = sqrt (dx*dx+dy*dy)
  where
    dx = (xval a) - (xval b)
    dy = (yval a) - (yval b)
```

...aunque gracias al ajuste de patrones raramente hace falta usarlas.

```
1   data Point = Pnt Float Float
2
3   distance :: Point -> Point -> Float
4   distance (Pnt ax ay) (Pnt bx by) = sqrt (dx*dx+dy*dy)
5     where
6       dx = ax - bx
7       dy = ay - by
```

Los espacios de nombres de tipos y de constructores están separados, así que no es posible que dos nombres entren en conflicto.
Por eso muchas veces se usa el mismo nombre para el tipo y para el constructor, y se diferencian por el contexto.

```haskell
data Point = Point Float Float

-- Hablamos de tipos
distance :: Point -> Point -> Float
-- Hablamos de constructores
distance (Point ax ay) (Point bx by) = ...
```

Aquel que incluye en su definición otros tipos desconocidos.
Su contructor de tipo tiene un parámetro tipo que puede usarse en los constructores de datos:

```
1 data Pair t = MakePair t t
2
3 -- Integer Pair
4 ip :: Pair Integer
5 ip = MakePair 3 4
6
7 first :: Pair t -> t
8 first (MakePair a _) = a
```

Esta es la base de la **programación genérica** en Haskell.

Aquel que incluye en su definición otros tipos desconocidos.
Su contructor de tipo tiene un parámetro tipo que puede usarse en los constructores de datos:

```haskell
data Pair t = Pair t t

-- Integer Pair
ip :: Pair Integer
ip = Pair 3 4

first :: Pair t -> t
first (Pair a _) = a
```

Esta es la base de la **programación genérica** en Haskell.

Tipos Unión

Un tipo de datos unión puede estar compuesto por la **unión** de varios tipos base **iguales** con diferente constructor de datos (*tag*):

```
1   data Temperatura = Celsius Float | Farenheit Float
2
3   congelado :: Temperatura -> Bool
4   congelado (Celsius   t) = t < 0.0
5   congelado (Farenheit t) = t < 32.0
```

El patrón permite especificar para qué datos se aplica esa expresión de la función: si se ha construido con _*este*_ constructor, entonces...

Un tipo de datos unión puede estar compuesto por la **unión** de varios tipos base **distintos** con diferente constructor de datos (*tag*):

```
1   data LetraONumero = Letra Char | Numero Integer
2
3   siguiente :: LetraONumero -> LetraONumero
4   siguiente (Letra  x) = Letra (chr (1 + ord x))
5   siguiente (Numero x) = Numero (1 + x)
```

Un tipo de datos unión puede estar compuesto por la **unión** de varios tipos base **distintos** con diferente constructor de datos (*tag*):

```
1  data Extended = Undefined | Number Double
2
3  mysqrt :: Extended -> Extended
4  mysqrt (Number x)
5    | x >= 0    = Number (sqrt x)
6    | otherwise = Undefined
7  mysqrt Undefined = Undefined
```

En realidad, un tipo enumerado es un caso particular de un tipo unión en el que ninguno de los constructores de datos tiene parámetros.

```
1  data Dia   = Lun | Mar | Mie | Jue | Vie | Sab | Dom
2  data State = ON | OFF
```

Al igual que en los enumerados, el ajuste de patrones en tipos unión se basa en el identificador del constructor de datos.

```
1  data Temperatura = Celsius Float | Farenheit Float
2
3  congelado :: Temperatura -> Bool
4  congelado (Celsius   t) = t < 0.0
5  congelado (Farenheit t) = t < 32.0
```

Al igual que en los enumerados, el ajuste de patrones en tipos unión se basa en el identificador del constructor de datos.

```
1  data LetraONumero = Letra Char | Numero Integer
2
3  siguiente :: LetraONumero -> LetraONumero
4  siguiente (Letra  x) = Letra (chr (1 + ord x))
5  siguiente (Numero x) = Numero (1 + x)
```

Un tipo de datos algebraico puede incluir en su definición otros tipos desconocidos, y puede tener varios constructores:

```haskell
data Maybe a = Nothing | Just a

phonebook :: [(String,Integer)]
find :: [(String,Integer)] -> Maybe Integer

main = case (find phonebook "Juan") of
          Nothing -> print "missing"
          Just n  -> print n
```

Un tipo de datos algebraico puede incluir en su definición otros tipos desconocidos, y puede tener varios constructores:

```
1  data Either a b = Left a | Right b
2
3  safediv :: Float -> Float -> Either String Float
4  safediv x 0 = Left "Divison by zero"
5  safediv x y = Right (x / y)
```

Un tipo puede utilizarse a si mismo en su definición.

```haskell
data List t = Empty | Cons t (List t)
data Tree t = Leaf t | Branch (Tree t) (Tree t)

fl :: List Float
fl = Cons 1.0 (Cons 2.0 Empty)

ft :: Tree Float
ft = Branch (Leaf 1.0) (Leaf 2.0)
```

Ejemplo:

```
1  fringe :: Tree a -> [a]
2
3  fringe (Leaf x)             =  [x]
4  fringe (Branch left right) =  fringe left ++ fringe right
```

Recordar que el ajuste de patrones permite combinar patrones de los tipos adecuados.

Eso puede ser especialmetne útil para tipos recursivos.

```
1 isBalanced :: Tree a -> Bool
2
3 isBalanced (Leaf _) = True
4 isBalanced (Branch (Branch _ _) (Leaf _)) = False
5 isBalanced (Branch (Leaf _) (Branch _ _)) = False
6 isBalanced (Branch l r) = isBalanced l && isBalanced r
```

Tipos de Datos **¿ Algebraicos ?**

**Algebra de Tipos**

| | |
|---|---|
| 0 | `Void` |
| 1 | `()` |
| 1+1 | `data Bool = True \| False` |
| 1+a | `data Maybe = Nothing \| Just a` |
| a+b | `data Either a b = Left a \| Right b` |
| a*b | `(a,b)`<br>`data Pair a b = Pair a b` |

Clases

¿Cómo compruebas si una lista contiene un elemento?

```
1  contiene :: a -> [a] -> Bool
2
3  contiene _ []       = False
4  contiene x (y:ys)  = (x == y) || contiene x ys
```

¿Qué problema tiene esta solución?

¿Cómo compruebas si una lista contiene un elemento?

```
1  contiene :: a -> [a] -> Bool
2
3  contiene _ []       = False
4  contiene x (y:ys)  = (x == y) || contiene x ys
```

¿Qué problema tiene esta solución?

¿Cómo sabes que el operador (==) está definido para el tipo *a*?

```haskell
contiene :: a -> [a] -> Bool
contiene _ []       = False
contiene x (y:ys)   = (x == y) || contiene x ys
```

Ideas:

- Definiendo la función *contiene* para cada tipo de datos que conozcamos que se puede comparar (`Float, Integer, ...`).

```
1   contiene :: a -> [a] -> Bool
2   contiene _ []       = False
3   contiene x (y:ys)   = (x == y) || contiene x ys
```

Ideas:

- Definiendo la función *contiene* para cada tipo de datos que conozcamos que se puede comparar (`Float, Integer, ...`).
  **NO** es posible en Haskell, no hay sobrecarga.

```haskell
contiene :: a -> [a] -> Bool
contiene _ []       = False
contiene x (y:ys)  = (x == y) || contiene x ys
```

Ideas:

- Definiendo la función *contiene* para cada tipo de datos que conozcamos que se puede comparar (`Float, Integer, ...`).
  **NO** es posible en Haskell, no hay sobrecarga.
- Poniendo requisitos/restricciones para el tipo *a*.

Las restricciones en Haskell se pueden indicar utilizando **clases de tipos**.

- No son *exactamente* como las clases de programación orientada a objetos...
- ... pero los tipos deben *instanciar* ciertas clases para indicar su relación.

# Clases

Las **clases** en Haskell son similares en algunos aspectos a las de los lenguajes orientados a objetos:

- Establecen las funciones que deben implementar los tipos que instancien la clase (como clases abstractas o interfaces).
- Pueden incluir funciones que definan comportamiento común.
- *(Java)* Permiten establecer restricciones sobre los tipos de datos involucrados, asegurando que ciertas funciones están disponibles.
- *(C++)* Un tipo de datos puede instanciar varias clases (herencia múltiple).

Sin embargo, las **clases** en Haskell tienen bastantes diferencias:

- No existe asociación dinámica de métodos o funciones mediante reglas de herencia.
- No afecta al comportamiento en tiempo de ejecución.
- Las clases no tienen atributos ni ningún tipo de información.
- Están más relacionadas con los *conceptos* de C++20 o las *restricciones* en Java.

Clase definida en Haskell:

```
1   class  Eq a  where
2     (==), (/=)  :: a -> a -> Bool
3     x == y      =  not (x /= y)
4     x /= y      =  not (x == y)
```

Restricción añadida sobre nuestra función:

```
1   contiene ::  Eq a =>  a -> [a] -> Bool
2               ^^^^^^^^^
3   contiene _ []       = False
4   contiene x (y:ys)   = (x == y) || contiene x ys
```

Instanciación de una clase para que un tipo de datos cumpla con la restricción:

```
1   data Fraction = Frac Integer Integer
2
3   instance Eq Fraction where
4     (==) (Frac n1 d1) (Frac n2 d2) = (n1==n2) && (d1==d2)
5 --  (Frac n1 d1) == (Frac n2 d2)   = (n1==n2) && (d1==d2)
```

Instanciación de una clase para que un tipo de datos cumpla con la restricción:

```
1   data Tree t = Leaf t | Branch (Tree t) (Tree t)
2
3   instance Eq (Tree t) where
4     Leaf a        == Leaf b        = a == b
5     Branch l1 r1 == Branch l2 r2 = (l1==l2) && (r1==r2)
6     _             == _             = False
```

¿ Problema ?

Instanciación de una clase para que un tipo de datos cumpla con la restricción:

```
data Tree t = Leaf t | Branch (Tree t) (Tree t)

instance Eq t => Eq (Tree t) where
  Leaf a        == Leaf b        = a == b
  Branch l1 r1 == Branch l2 r2 = (l1==l2) && (r1==r2)
  _             == _             = False
```

Instanciación **automática** de una clase para que un tipo de datos cumpla con la restricción, para casos triviales:

```
1  data Fraction = Frac Integer Integer
2
3  instance Eq Fraction where
4    (==) (Frac n1 d1) (Frac n2 d2) = (n1==n2) && (d1==d2)
```

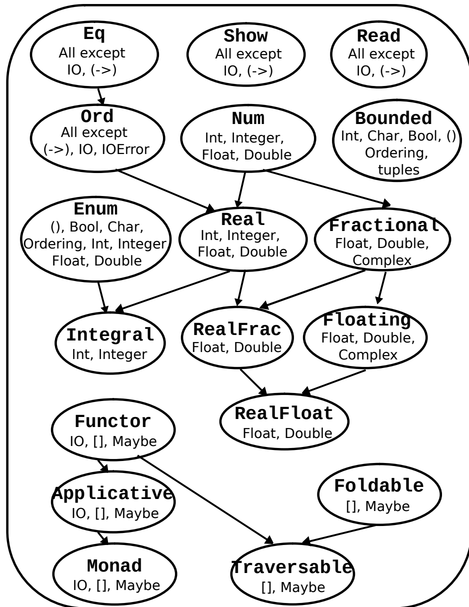
O bien:

```
1  data Fraction = Frac Integer Integer  deriving Eq
```

Definidas por Haskell:

```haskell
1  class  Eq a => Ord a  where
2    (<), (<=), (>=), (>)  :: a -> a -> Bool
3    max, min              :: a -> a -> a
4    ...
5    x < y =  (x <= y) && (x /= y)
6    max x y
7      | x < y = y
8      | otherwise = x
```

```
1  class Show a  where
2    show :: a -> String
3
4  class Read a where
5    read :: String -> (a,String)
6
7  class Functor f where
8      fmap :: (a -> b) -> f a -> f b
9
10 ....
```



Eq
All except
IO, (->)
Show
All except
IO, (->)
Read
All except
IO, (->)
Ord
All except
(->), IO, IOError
Num
Int, Integer,
Float, Double
Bounded
Int, Char, Bool, ()
Ordering,
tuples
Enum
(), Bool, Char,
Ordering, Int, Integer
Float, Double
Real
Int, Integer,
Float, Double
Fractional
Float, Double,
Complex
Integral
Int, Integer
RealFrac
Float, Double
Floating
Float, Double,
Complex
RealFloat
Float, Double
Functor
IO, [], Maybe
Applicative
IO, [], Maybe
Monad
IO, [], Maybe
Foldable
[], Maybe
Traversable
[], Maybe

```haskell
data Fraction = Frac Integer Integer deriving Eq

value :: Fraction -> Double
value (Frac num den) =
    (fromInteger num) / (fromInteger den)
```

Ejemplo de interpretación:

```
1  > value (Frac 1 3)
2  0.3333333333333333
3
4  > map value (zipWith Frac [2..6] [3,5..])
5  [0.6666,0.6,0.5714,0.5556,0.5454]
```

Sin embargo, este tipo de datos no se comporta como el resto de tipos de Haskell: las fracciones no se pueden mostrar, no se pueden operar...

Instanciamos Show:

```haskell
instance Show Fraction where
  show (Frac num den) = (show num)++"/"++(show den)
```

Y podemos hacer lo siguiente:

```haskell
> zipWith Frac [2..6] [3,5..]
[2/3,3/5,4/7,5/9,6/11]
```

Instanciamos Num:

```
1  instance Num Fraction where
2    negate (Frac num den)         = Frac (negate num) den
3    (+) (Frac n1 d1) (Frac n2 d2) = Frac (n1*d2+n2*d1) (d1*d2)
4    (*) (Frac n1 d1) (Frac n2 d2) = Frac (n1*n2) (d1*d2)
5    fromInteger i = Frac i 1
6    abs f
7      | (value f) < 0 = (fromInteger (-1)) * f
8      | otherwise     = f
9    signum f
10     | (value f) < 0 = -1
11     | (value f) > 0 =  1
12     | otherwise     =  0
```

Y podemos hacer lo siguiente:

```
1  > sum ( zipWith Frac [2..6] [3,5..] )
2  30552/10395
3
4  > product ( zipWith Frac [2..6] [3,5..] )
5  720/10395
```

Se pueden definir clases propias:

```haskell
class YesNo a where
    yesno :: a -> Bool
```

sobre las que se construyen funciones

```haskell
yesnoIf :: (YesNo y) => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult =
  if yesno yesnoVal then yesResult else noResult
```

Se instancia esa clase nueva:

```
1   instance YesNo Int where
2       yesno 0 = False
3       yesno _ = True
4
5   instance YesNo [a] where
6       yesno [] = False
7       yesno _  = True
8
9   instance YesNo Bool where
10      yesno = id
```

Usamos la función de la clase sobre los tipos que la instancian:

```
1  ghci> yesno 100
2  True
3  ghci> yesno 0
4  False
5  ghci> yesno True
6  True
7  ghci> yesno []
8  False
9  ghci> yesno [0,0,0]
10 True
```

Usamos la función que incluye restricción sobre la clase sobre los tipos que la instancian:

```
ghci> yesnoIf [] "YEAH!" "NO!"
"NO!"
ghci> yesnoIf [2,3,4] "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf True "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf 0 "YEAH!" "NO!"
"NO!"
ghci> yesnoIf -1 "YEAH!" "NO!"
"YEAH!"
```

- Definir una jerarquía de clases propias.
- Utilizar la herencia múltiple.

Referencias:

- `http://www.haskell.org/onlinereport/prelude-index.html`
- `http://www.cse.unsw.edu.au/~en1000/haskell/inbuilt.html`

# El Sistema de Tipos de Haskell: Tipos y Clases

## Tecnología de Programación

**Adolfo Muñoz – Juan Magallón**

**Grado en Ingeniería Informática**

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20