



# Paradigma Funcional

## Módulo 7: Sistema de tipos

por Fernando Dodino  
Germán Leiva  
Carlos Lombardi  
Nicolás Passerini  
Daniel Solmirano  
Versión 2.1  
Mayo 2018

# Contenido

## [1 Introducción](#)

## [2 Tipos genéricos](#)

### [2.1 La función id](#)

### [2.2 Funciones genéricas con listas](#)

### [2.3 Funciones genéricas con tuplas](#)

### [2.4 Polimorfismo paramétrico](#)

## [3 Clasificando tipos](#)

### [3.1 La función \(+\)](#)

### [3.2 Num como typeclass](#)

### [3.3 La función "mayor"](#)

### [3.4 El typeclass Ord](#)

### [3.5 Un tipo pertenece a muchas clases](#)

### [3.6 La función "igual"](#)

### [3.7 El typeclass Eq](#)

### [3.8 Relaciones entre clases](#)

### [3.9 Polimorfismo ad-hoc](#)

## [4 Ejercicios de inferencia de tipos](#)

### [4.1 Introducción](#)

### [4.2 Ejemplo un poco más complejo](#)

### [4.3 Un ejemplo de final](#)

### [4.4 Otro ejemplo, de parcial](#)

### [4.5 Un último ejemplo](#)

## [5 BONUS: Profundizando typeclasses](#)

### [5.1 Mecanismo de derivación](#)

### [5.2 Definición de Eq](#)

### [5.3 Definición de Ord en base a Eq](#)

### [5.4 Otro mecanismo de adhesión de un tipo a un typeclass](#)

### [5.5 Mostrar una función por consola](#)

### [5.6 Creando nuestros propios typeclasses](#)

## [6 Resumen](#)

# 1 Introducción

Hasta el momento hemos visto los tipos de una función sabiendo que existen

- **tipos simples:** Int, Bool, Float, Char, las funciones cuyos parámetros se delimitan con flechas
- **tipos compuestos:** String, listas, tuplas, los tipos de dato definidos por el usuario (data)

También contamos con los sinónimos de tipo

```
type Nombre = String
```

que permiten establecer equivalencias con nombres más representativos del dominio.

En el presente capítulo explicaremos el poderoso sistema de tipos que tiene Haskell que permite inferir los tipos de cada función que definamos o cada expresión que evaluemos en la consola.

## 2 Tipos genéricos

### 2.1 La función id

Quizás ya hayan conocido la función identidad en Haskell, llamada id:

```
id x = x
```

¿De qué tipo es la función id?

- es válido para booleanos
- pero también para strings
- o incluso para las funciones

```
λ id True
True
:: Bool
λ id "hola"
"hola"
:: [Char]
λ id head
<function>
```

Entonces definimos el tipo de la función id como:

```
id :: a -> a
```

donde **a** representa una variable de tipo sin restricciones particulares: cualquier tipo puede participar del dominio (e imagen) de la función `id`.

Si hubiéramos definido la función `id` de la siguiente manera

```
id :: a -> a
id a = a
```

tenemos dos **a** diferentes:

- las **a** naranjas en la definición `id :: a -> a` se refieren a variables de tipo, son utilizadas por el compilador Haskell para determinar si el tipo es correcto
- las **a** verdes en la definición `id a = a` corresponden a variables para valores, son incógnitas que se resuelven en la aplicación de la función

## 2.2 Funciones genéricas con listas

Ya hemos conocido otras funciones que pueden tomar valores de cualquier tipo:

```
head :: [a] -> a
tail :: [a] -> [a]
last :: [a] -> a
```

Y volviendo a la función `length`:

```
length [] = 0
length (x:xs) = 1 + length xs
```

¿Qué restricción tienen los elementos? Ninguna, de hecho podemos reemplazar `x` por una variable anónima sin perjuicio:

```
length [] = 0
length (_:xs) = 1 + length xs
```

Entonces `length` es una función cuyo dominio son listas de cualquier tipo, y cuya imagen son los enteros, de acuerdo a lo que vemos en los dos patrones:

- por lista vacía, devuelve 0
- por lista con elementos, es 1 + ... longitud que sabemos que es 0 ...

```
length :: [a] -> Int
```

Algunos ejemplos:

```
λ length [ head, last ]
2
```

```
λ length [ (4, True), (8, False), (5, False) ]  
3
```

Otras funciones que trabajan en forma genérica con listas son `map`, `all`, `any`, `filter` y las de la familia `fold`, entre otras.

## 2.3 Funciones genéricas con tuplas

Repasando las tuplas, también tenemos funciones que toman valores de cualquier tipo:

```
fst :: (a, b) -> a
```

```
snd :: (a, b) -> b
```

Solo que aquí utilizamos `a` y `b` como variables diferentes, para aceptar que la tupla tenga dos elementos de diferente tipo. Si `a == b` es un caso particular, donde tengo una tupla con valores del mismo tipo en cada uno de los elementos:

```
(2, 5) :: (Int, Int) ==> a es Int, b es Int
```

```
(2, True) :: (Int, Bool) ==> a es Int, b es Bool
```

## 2.4 Polimorfismo paramétrico

Las funciones que aceptan valores de cualquier tipo, sin ningún tipo de restricciones, tienen **polimorfismo paramétrico**: solo necesitamos definir una vez la función para aplicarla con cualquier tipo.

# 3 Clasificando tipos

Vamos a jugar un poco con algunas funciones que ya vienen en Haskell

## 3.1 La función (+)

Suena razonable sumar

- dos números enteros
- dos números con decimales
- dos fracciones

```
λ 2 + 3  
6
```

```
λ 2.8 + 3.1
```

```
5.9
```

```
λ (2/3) + (1/3)
1.0
```

Para definir el dominio e imagen de la función (+), necesitamos pensar en cada caso puntual:

```
(+) :: Int -> Int -> Int
o bien
(+) :: Float -> Float -> Float
o bien
(+) :: Double -> Double -> Double
```

Pero una función solo puede tener un tipo, o sea, solo puede tener un dominio e imagen determinado. Si intentamos definir el tipo de (+) como

```
(+) :: a -> a -> a    -- mmm...
```

esto no es correcto porque no funciona para todos los tipos. Por ejemplo, para los booleanos la función (+) da error

```
λ True + False
Error
```

lo mismo para dos funciones del mismo tipo:

```
last :: [b] -> b
head :: [b] -> b
```

Cuando uso (+) el tipo a sería ([b] -> b)

*-- no tiene sentido sumar funciones con el operador (+)*

```
λ last + head
Error
```

Entonces algunos tipos se pueden utilizar, otros no. ¿Qué comparten en común los tres tipos: Int, Float y Double? Son números. Necesitamos cambiar nuestra definición de la función (+), sin usar un tipo específico, sino uno genérico pero al que le vamos a aplicar una restricción:

(+) :: (Si asumimos que a es "numérico") entonces a -> a -> a

en Haskell es:

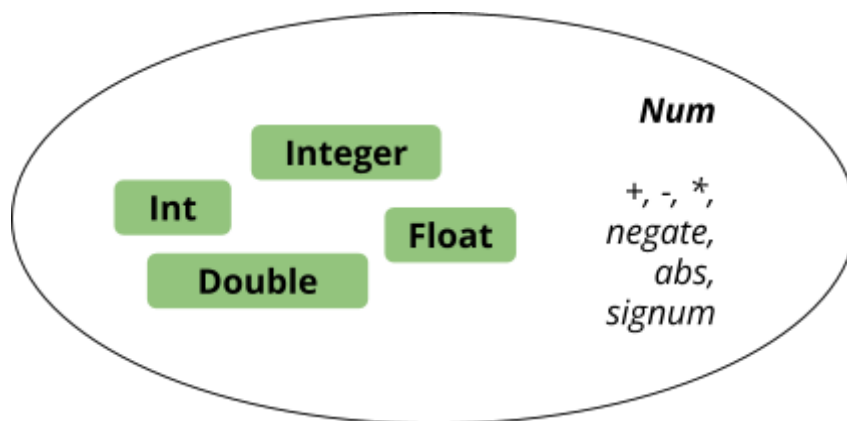
```
(+) :: (Num a) => a -> a -> a
```

### 3.2 Num como typeclass

Un typeclass

- agrupa tipos
- define una interfaz, que son un conjunto de operaciones que todos los tipos pertenecientes deben implementar
- y adicionalmente, permite definir un comportamiento por defecto para ciertas operaciones

Los tipos Int, Float, Double son *instancias* del *typeclass* Num que los agrupa:



Por ser Num estos tipos deben implementar las siguientes operaciones:

- $(+)$ ,  $(-)$ ,  $(*)$  :: (Num a) => a -> a -> a
- negate, abs, signum :: (Num a) => a -> a

Ejemplo: en esta función

**funcionLoca** (x, y) = abs x + y

- x tiene la restricción de aplicarse con la función abs (definida para Num)
- y se aplica con el operador (+) (definido para Num) junto con la imagen de abs x, que es Num

En consecuencia, el tipo de la funcionLoca es

**funcionLoca** :: Num a => (a, a) -> a

### 3.3 La función "mayor"

Suena razonable preguntar

- si un número es mayor a otro

- si una palabra es más grande que otra (porque aparece después en el diccionario)
- si un punto 2D es mayor que otro (comparando la primera componente y si es igual la segunda)

```
λ 1 > 3
False
```

```
λ "hola" > "chau"
True
```

```
λ (3,2) > (1,0)
True
```

Nuevamente, tenemos tres tipos diferentes para la misma función:

```
(>) :: Int -> Int -> Bool
(>) :: String -> String -> Bool
(>) :: (Int, Int) -> (Int, Int) -> Bool
```

Pero dijimos que una función solo puede tener un tipo, o sea, solo puede tener un dominio e imagen determinado.

Tampoco podemos decir que el tipo de (>) es

```
(>) :: a -> a -> Bool -- Esto está mal
```

porque no funciona para comparar funciones:

```
last :: [b] -> b
head :: [b] -> b
```

Cuando uso (>) el tipo **b** sería **([b] -> b)**  
*-- no tiene sentido comparar funciones con la función (>)*  

```
λ last > head
Error
```

Entonces algunos tipos se pueden utilizar, otros no. ¿Qué comparten en común los tres tipos: tuplas, strings y números? Tienen un orden. Necesitamos cambiar nuestra definición de la función (+), sin usar un tipo específico, sino uno genérico pero al que le vamos a aplicar una restricción:

(>) :: (Si asumimos que a es "ordenable") entonces a -> a -> Bool  
 en Haskell es:

```
(>) :: (Ord a) => a -> a -> Bool
```

¿De qué tipo es la función max?

```
max a b | a > b = a
        | otherwise = b
```

a y b deben ser del mismo tipo, pero además deben poder compararse por un orden. En resumen

```
max :: (Ord a) => a -> a -> a
```

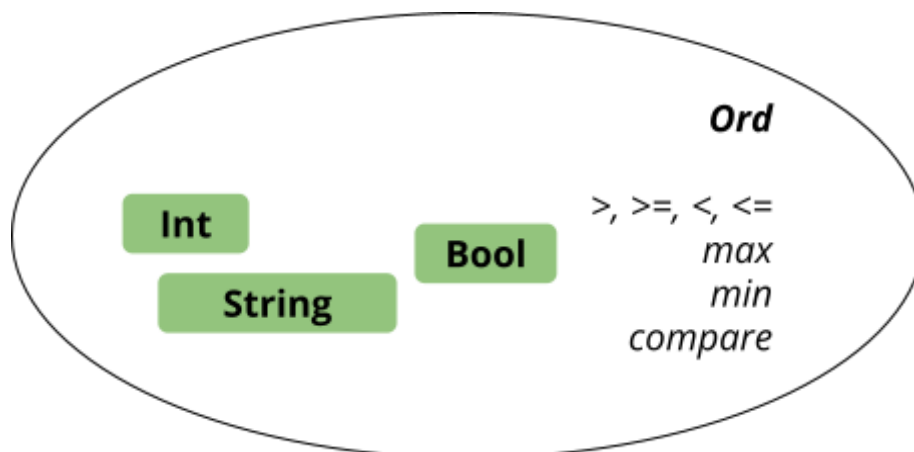
### 3.4 El typeclass Ord

Ord es un typeclass que define las siguientes operaciones

- (<), (>), (<=), (>=) :: (Ord a) => a -> a -> Bool
- max, min :: (Ord a) => a -> a -> a

Y aplica a los tipos

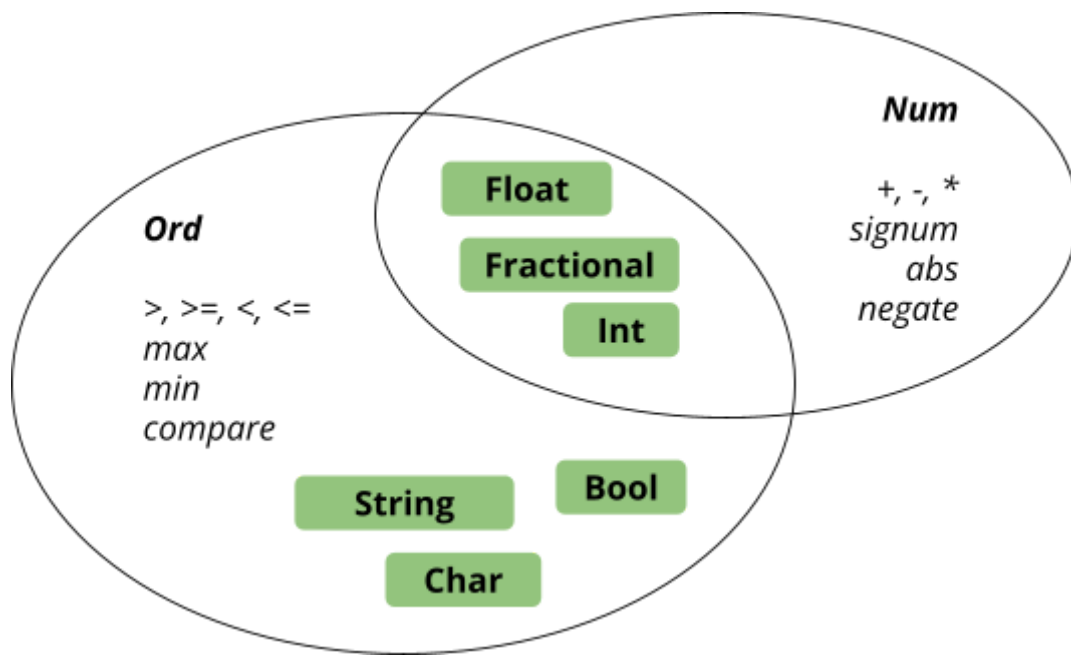
- Bool
- Char
- Double
- Int
- Float
- ... entre otros



### 3.5 Un tipo pertenece a muchas clases

De los ejemplos anteriores hemos visto que

- una clase agrupa varios tipos (definiendo una interfaz común entre ellos)
- pero también un tipo pertenece a varias clases
  - y le obliga al tipo a implementar funciones para cada clase



### 3.6 La función "igual"

Suena razonable preguntar

- si dos números son iguales
- si dos listas son iguales (coinciden en todos sus elementos)
- si dos tuplas son iguales

```
λ 1 == 3
False
```

```
λ [1, 3, 2] == [1, 2, 3]
False
```

```
λ (3,2) == (2+1,2)
True
```

Pero no sirve para todos los tipos,

```
(==) :: a -> a -> Bool -- esto está mal
```

ya que nuevamente no puedo comparar funciones:

```
last :: [b] -> b
head :: [b] -> b
```

Cuando uso `(==)` el tipo `b` sería `([b] -> b)`

```
λ last == head -- no tiene sentido comparar funciones con (==)
Error
```

(==) :: (Si asumimos que a es "equiparable") entonces a -> a -> Bool

En Haskell eso se escribe de la siguiente manera

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

### 3.7 El typeclass Eq

Eq es un typeclass que define las siguientes operaciones

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

Y aplica a los tipos

- Bool
- Char
- Double
- Int
- Float
- ... entre otros

Ejemplo:

```
elem _ [] = False
elem valor (x:xs) = valor == x || elem valor xs
```

¿Qué restricciones de tipo aplican sobre elem?

- las variables *valor* y *x* deben ser del mismo tipo
- pero además deben poderse "equiparar", es decir deben ser de algún tipo que implemente la función (==)
- la lista por lo tanto no puede ser de cualquier tipo, tiene que ser una lista del mismo tipo que del valor
- lo que devuelve es un valor booleano

En resumen, aplicamos una restricción de tipo Eq sobre a:

```
elem :: (Eq a) => a -> [ a ] -> Bool
```

### 3.8 Relaciones entre typeclasses

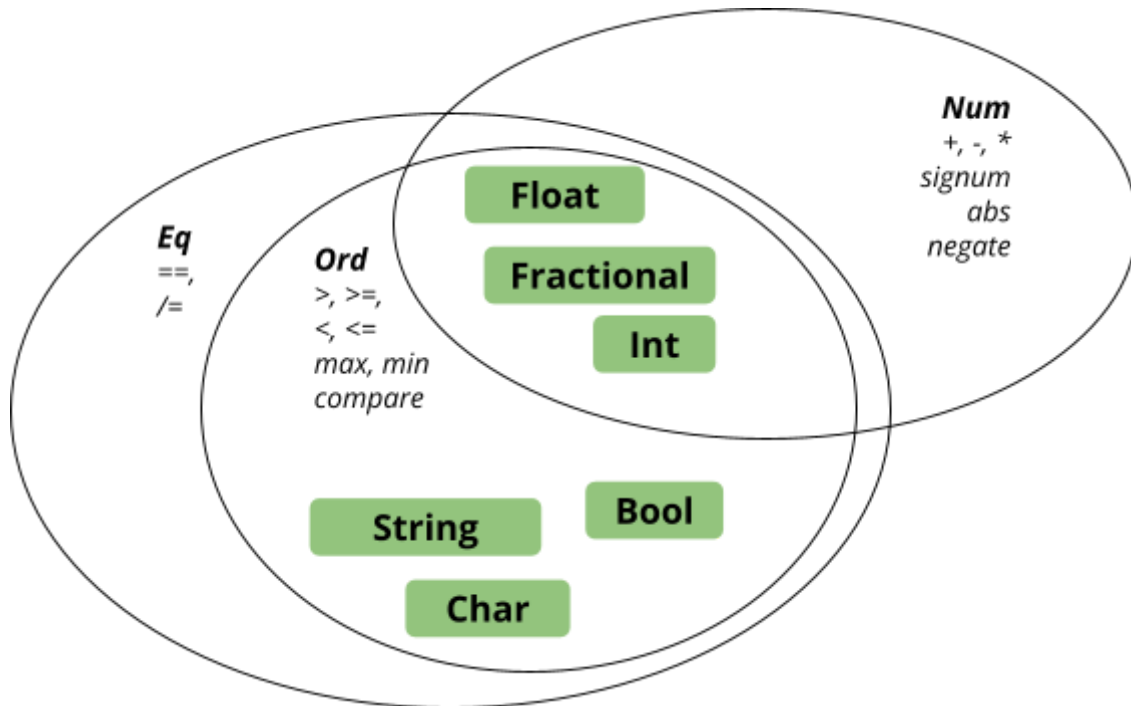
El lector se puede imaginar: "ok, ahora espero el gráfico donde aparecen los tipos Int, Bool y String perteneciendo al conjunto del typeclass Eq". Pero antes de continuar, repasemos la interfaz del typeclass Ord:

- **operación** (<=), (>=)

- y otras operaciones que no nos interesan tanto

Es decir, cualquier tipo que implemente `Ord` está obligado no solo a definir qué valor es mayor que otro, sino también el “mayor o igual”, por lo tanto `Ord` es una especialización de `Eq`. Es decir, todo `Ord` es por lo tanto `Eq`...

Gráficamente



### 3.9 Polimorfismo ad-hoc

Todas estas funciones

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

admiten una gran variedad de tipos, pero no todos: tienen una restricción demarcada por una typeclass. Por este motivo decimos que tienen polimorfismo *ad-hoc*.

## 4 Ejercicios de inferencia de tipos

### 4.1 Introducción

Al intentar calcular el tipo de una función, lo primero que tenemos que hacer es mirar las expresiones que se usan dentro de su definición. Veamos este ejemplo:

```
f x _ [] = x
f x y (z:zs)
  | y z > 0 = z + f x y zs
  | otherwise = f x y zs
```

- Sabemos que  $f$  tiene 3 parámetros (en realidad es uno solo , pero ya hemos trabajado suficiente la aplicación parcial para tomarnos esta licencia)
- Por el primer patrón ( $f\ x\ \_ [] = x$ ) sabemos que
  - el último parámetro es una lista
  - y el tipo del primer parámetro coincide con el que devuelve la función

Entonces por el momento tenemos

```
f :: a -> ?? -> [b] -> a
```

Al ver

```
f x y (z:zs) | y z > 0
```

sabemos que

- el segundo parámetro es una función que recibe un elemento de la lista que forma parte del tercer parámetro: lo que devuelve esa función tiene que saber ordenarse (por la función  $>$ ), pero además se compara  $> 0$ , con lo cual tenemos una restricción adicional, lo que devuelve la función  $y$  es un número

Entonces, tenemos

```
f :: (Ord c, Num c) => a -> (b -> c) -> [b] -> a
```

Por último, veamos cómo termina la segunda guarda

```
f x y (z:zs)
  | y z > 0 = z + f x y zs
```

Lo que devuelve la función **f** debe poder sumarse (operador +, definido en la typeclass Num), lo mismo que cada elemento de la lista que viene como tercer parámetro, aunque no es obligatorio que sepa ordenarse de mayor a menor.

Por otra parte, *z* (que es un elemento de la lista) también se debe poder sumar. Entonces repasemos la definición de tipos de la función:

```
f :: (Ord c, Num c) => a -> (b -> c) -> [b] -> a
```

el tipo *a* es el mismo que el tipo *b*, un número:

```
f :: (Num a, Ord c, Num c) => a -> (a -> c) -> [a] -> a
```

Lo probamos en Haskell

```
λ :t f
```

```
f :: (Ord a, Num a, Num t) => t -> (t -> a) -> [t] -> t
```

Solo tenemos diferencia en los nombres de cada variable de tipo (lo que para nosotros es *a* es para Haskell *t* y lo que nosotros llamamos *c* es para Haskell *a*). En anteriores versiones de Haskell todo Num era también Ord y no hacía falta esta distinción:

```
f :: (Num a, Num t) => t -> (t -> a) -> [t] -> t
```

## 4.2 Otro ejemplo, de parcial

```
f a b c d e
| (a . d e) (1, True) = 0
| otherwise          = length (b:c) + e
```

Determinar los parámetros es fácil: son 5.

- La función *f* devuelve un Int, determinado por la primera guarda (0) y por length en la segunda
- *b* y *c* también se infieren fácil, porque aparece el patrón de lista en la segunda guarda: *b* es un tipo cualquiera y *c* una lista de ese mismo tipo cualquiera
- ¿qué hay con *e*? No puede ser otra cosa que un Int, porque está sumándose con length que devuelve un Int, y la operación (+) está definido para typeclass Num que sean del mismo tipo (int con Int, Float con Float, etc.)
- *d* es una función, que tiene... 2 parámetros, porque aparece en una composición, aplicada parcialmente con un *e* (que dijimos que era un entero). El segundo parámetro es una tupla, y por pattern matching sabemos que es de tipo (Int, Bool)
- Por último la imagen de *d* es el dominio de *a*, que es entonces... otra función, que recibe lo que sea el tipo devuelto por *d* y devuelve un Bool (para usarse dentro de la guarda)

Ordenamos un poco esto y nos queda

```
f :: (b -> Bool) -> a -> [a] -> (Int -> (Int, Bool) -> b) -> Int -> Int
```

## 4.5 Un último ejemplo

```
f x y = (> 10) . head . filter (x y) . map y
```

¿Qué sabemos?

- que  $y$  es una función
- como `map` va de  $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$  sabemos que  $f$  tiene implícito un argumento más, que es una lista (en principio de  $a$ )
- $x$  es otra función, porque está aplicando  $y$  dentro de `filter`
- la lista que devuelve `map`  $y$  se vuelve input para `filter (x y)`, donde los paréntesis están marcando precedencia, y aplicación de una función  $x$  a otra  $y$ .
  - como `map` devuelve una lista de  $b$ , utilizaremos una variable de tipo  $b$  por defecto
  - `filter` va de  $(b \rightarrow \text{Bool}) \rightarrow [b] \rightarrow [b]$ , por lo tanto  $(x y)$  debe ir de  $(b \rightarrow \text{Bool})$ 
    - entonces  $x$  va de  $((a \rightarrow b) \rightarrow b \rightarrow \text{Bool})$ , ya que el primer parámetro de  $x$  es la función  $y$
  - `filter` devuelve una lista de  $b$
- esa lista se compone con `head`, que no tiene restricciones de tipo, por lo tanto lo que devuelve es un elemento de cualquier tipo...
- ...que se compone con  $(> 10)$  que sabemos que necesita ser tanto `Ord` como `Num`, entonces lo que devuelve `filter` (y lo que transforma `map`) tiene que ser `Ord` y `Num`, eso es lo que debe ser  $b$
- y lo que devuelve  $f$  es claramente un `Bool`

Con todo esto sabemos que

```
f :: (Ord b, Num b) => ((a -> b) -> b -> Bool) -> (a -> b) -> [a] -> Bool
```

Pueden encontrar más ejemplos [en la wiki](#).

## 5 BONUS: Profundizando typeclasses

### 5.1 Mecanismo de derivación

Si definimos nuestro tipo de dato Nota con múltiples constructores:

```
data Nota = Insuficiente | Regular | Aprobado | Sobresaliente
```

En realidad estamos definiendo una enumeración, escribiendo en orden cada uno de los valores. No obstante

```
λ Sobresaliente < Regular
```

```
<interactive>:13:15:
```

```
    No instance for (Ord Nota) arising from a use of '<'
```

```
    In the expression: Sobresaliente < Regular
```

```
    In an equation for 'it': it = Sobresaliente < Regular
```

Lo que dice Haskell es que nuestro tipo Nota no es *instancia* de Ord para poderlo comparar por mayor o menor. También tenemos errores con estas operaciones

```
λ show Regular
```

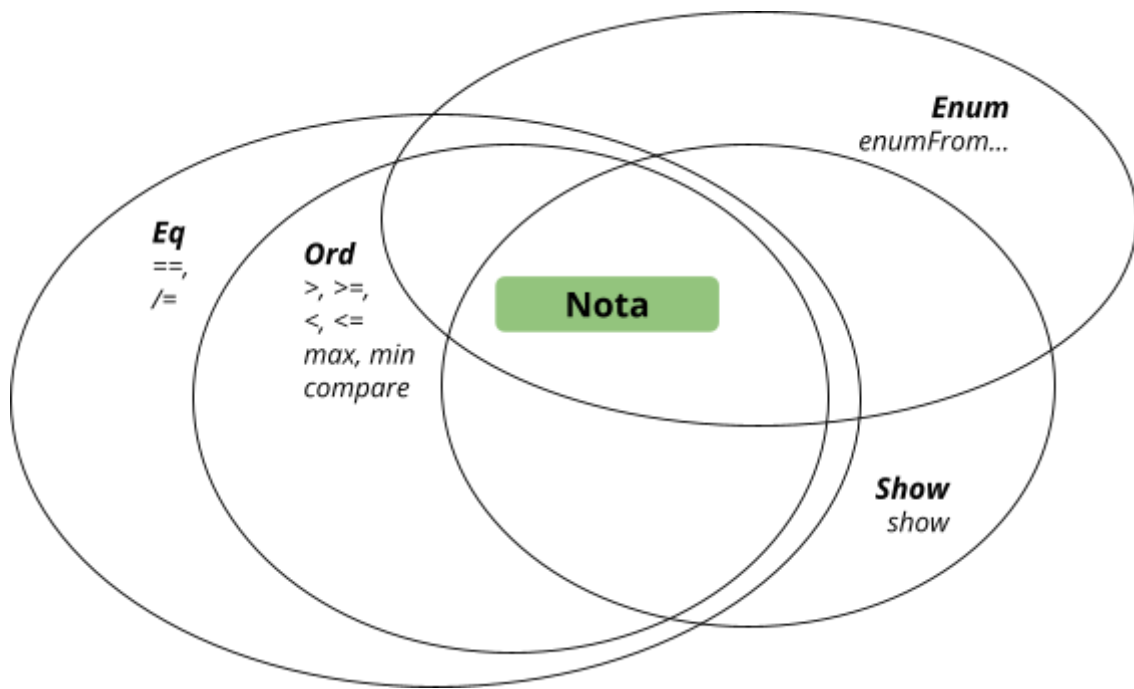
```
Error -- el tipo de dato Nota no es instancia de Show
```

```
λ Sobresaliente == Regular
```

```
Error -- el tipo de dato Nota no es instancia de Eq
```

Ahora vamos a incorporar nuestro data Nota a los typeclasses Eq, Ord, Enum y Show:

```
data Nota = Insuficiente | Regular | Aprobado | Sobresaliente
deriving (Eq, Ord, Show, Enum)
```



Mediante el mecanismo de *deriving* ahora el tipo `Nota` es *instancia* de `Eq`, `Ord`, `Show` y `Num`. Esto permite que comparemos las notas y también se sepan mostrar por consola:

```
λ Sobresaliente < Regular
False
λ Sobresaliente == Regular
False
λ show Regular
"Regular"
λ Sobresaliente > Regular
True
λ enumFromTo Insuficiente Aprobado
[Insuficiente, Regular, Aprobado]
```

Pero nunca definimos la operación (`==`), ni la función `show` para nuestro `data`. ¿Cómo es que funcionó? A partir del mecanismo de derivación “mágico” que especifica el compilador Haskell<sup>1</sup>.

Ésta es la justificación por la cual les hacíamos escribir la definición de un `data` con `deriving Show`: de esa manera se visualiza bien en la consola un tipo de dato definido por nosotros, aunque para ser más estrictos deberíamos haberles pedido que derivaran de `Eq` para poder comparar dos personas, dos clientes, dos autos...

<sup>1</sup> El lector interesado puede investigar el [siguiente link de Stack Overflow](#)

## 5.2 Definición de Eq<sup>2</sup>

Eq es un typeclass cuya definición podemos ver...

```
-- | The 'Eq' class defines equality ('==') and inequality ('/=').
-- All the basic datatypes exported by the Prelude are instances of 'Eq',
-- and 'Eq' may be derived for any datatype whose constituents are also
-- instances of 'Eq'.
--
-- Minimal complete definition: either '==' or '/='.
```

```
class Eq a where
    (==), (/=)      :: a -> a -> Bool
    ...
    x /= y         = not (x == y)
    x == y         = not (x /= y)
```

Lo que dice la definición es:

- creamos un typeclass llamado Eq
- $a$  es una variable de tipo que se utilizará posteriormente en las definiciones de las operaciones que hagamos. Entonces si Int es un Eq,  $a$  aplicará para Int en ese caso. Todos los tipos que sean instancias de Eq pueden encajar en la variable  $a$ , que podríamos llamar con cualquier otro nombre que comience con minúscula.
- luego definimos las operaciones que debe implementar dicho typeclass, en este caso la igualdad (==) y desigualdad (/=). Aquí usamos el tipo a, en particular la igualdad se define como `(==) :: (Eq a) => a -> a -> Bool`
- Y por último están las implementaciones o *métodos default*, que son las definiciones de funciones que todos los tipos que sean instancia de ese typeclass van a aprovechar. De manera que si definimos la operación (==) para comparar dos alumnos, no necesitamos que el tipo Alumno defina la operación (/=), porque esa definición la toma de la implementación del typeclass.
- Por consiguiente, la definición mínima completa (*minimal complete definition*) es el conjunto de operaciones que un tipo está obligado a implementar para poder ser instancia de ese typeclass. Las otras operaciones surgen de los métodos default, que cada tipo puede sobrescribir.

---

<sup>2</sup> Lo que vamos a explicar a continuación es el mecanismo por defecto, que es diferente al mecanismo mediante *deriving*

### 5.3 Definición de Ord en base a Eq

Además de las relaciones de tipo y typeclass, un typeclass puede estar contenido dentro de otro typeclass. Es el caso de Ord, que especializa la definición de Eq<sup>3</sup>:

```
class (Eq a) => Ord a where
  compare          :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min         :: a -> a -> a
  ...
  -- These two default methods use '<=' rather than 'compare'
  -- because the latter is often more expensive
  max x y = if x <= y then y else x
  min x y = if x <= y then x  else y
```

Aquí vemos que

- hay una función compare que no tiene implementación, hay que definirla para cada tipo
- pero todo tipo que sea *instancia* de Ord no necesita definir max ni min, porque ya hay una implementación por defecto (default method), que delega en la operación (<=)

### 5.4 Otro mecanismo de adhesión de un tipo a un typeclass

Tenemos el tipo de dato Persona definido de la siguiente manera:

```
data Persona = Persona {
  nombre :: String,
  edad   :: Int,
  hobbies :: [String]
}
```

```
alf = Persona "Alfredo" 29 ["Dar clases", "Cocinar"]
dodain = Persona "Fernando" 44 ["Dar clases", "Jugar al futbol",
"Cocinar", "Jardineria"]
nico = Persona "NicoS" 30 ["Dar clases", "Comer carne"]
```

---

<sup>3</sup> Pueden ver la definición completa en <http://hackage.haskell.org/package/base-4.4.0.0/docs/src/GHC-Classes.html>

Tenemos el siguiente requerimiento: “Una persona es mayor que otra si tiene más años”. No podemos evaluar una expresión que compare si una persona es mayor a otra:

```
λ dodain > alf
<interactive>:28:8:
  No instance for (Ord Persona) arising from a use of '>'
  In the expression: dodain > alf
  In an equation for 'it': it = dodain > alf
```

Y sabemos por qué: el tipo Persona no es instancia de Ord. Ahora si agregamos el mecanismo de deriving:

```
data Persona = Persona {
  nombre :: String,
  edad :: Int,
  hobbies :: [String]
} deriving (Ord)
```

Al tratar de resolver Persona, vemos que a Haskell le molesta algo:

```
No instance for (Eq Persona)
  arising from the 'deriving' clause of a data type declaration
Possible fix:
  use a standalone 'deriving instance' declaration,
  so you can specify the instance context yourself
When deriving the instance for (Ord Persona)
```

La molestia pasa porque queremos que Persona sea instancia de Ord, pero sabemos que todo Ord es también un Eq. Y Persona no es un Eq, porque no se lo dijimos. Entonces, modificamos nuestra definición:

```
data Persona = Persona {
  nombre :: String,
  edad :: Int,
  hobbies :: [String]
} deriving (Eq, Ord)
```

Ahora sí, la construcción de Persona es correcta. Y puedo comparar dos personas:

```
λ dodain > alf
True
```

¡Parece funcionar! Ya que Dodain tiene 44 y Alf 29. Pero cuando pregunto

```
λ dodain > nico
False
```

Vemos que el comportamiento derivado de Ord es comparar en base al primer atributo, por eso el orden es "Alf" - "Fernando" - "NicoS". Si queremos modificar el comportamiento tenemos que definir una función que compare explícitamente dos personas:

```
instance Ord Persona where
    (>) unaPersona otraPersona = edad unaPersona > edad otraPersona
```

```
λ dodain > alf
True
λ dodain > nico
True
λ alf < nico
True
```

Ahora sí vemos que la comparación se hace efectivamente contra la edad de cada persona.

Haskell permite mostrar de qué typeclasses es instancia un tipo, evaluando en la consola

```
λ :i Persona
data Persona
  = Persona {
      nombre :: String,
      edad :: Int,
      hobbies :: [String]
  } -- Defined at recordSyntaxPersona.hs:1:1
instance Eq Persona -- Defined at recordSyntaxPersona.hs:9:19
instance Ord Persona -- Defined at recordSyntaxPersona.hs:21:10
instance Show Persona -- Defined at recordSyntaxPersona.hs:9:13
```

## 5.5 Mostrar una función por consola

Cuando empezamos a utilizar las funciones como valores, nos encontramos con que Haskell no sabía cómo mostrar una función por consola:

```
λ map (+ 1)
<interactive>:23:1:
  No instance for (Show ([b0] -> [b0]))
    (maybe you haven't applied enough arguments to a function?)
```

arising from a use of 'print'  
In the first argument of 'print', namely 'it'  
In a stmt of an interactive GHCi command: print it

Lo que nos está diciendo es que la función es un tipo que no es *instancia* de Show, por lo tanto no sabe cómo mostrarla por consola.

La solución que les dimos en su momento fue agregar un import  
`import Text.Show.Functions`

¿Qué hace ese archivo? Lo pueden ver completo en [este link](#). Una implementación posible sería:

```
instance Show (a -> b) where
  show _ = "<function>"
```

Esta definición agrega que cualquier función (todas tienen un parámetro para Haskell) es instancia de Show. Eso permite que cualquier función se muestre por consola como

```
λ map (+ 1)
<function>
λ map
<function>
```

## 5.6 Creando nuestros propios typeclasses

Los typeclasses de Haskell no son cerrados sino **extensibles**, podemos crear nuestros propios typeclasses.

En el sistema educativo universitario queremos tener

- notas numéricas
  - para aprobar, necesitamos sacar 6 ó más
  - para promocionar, necesitamos sacar 8 ó más
- y notas de concepto: Insuficiente, Regular, Aprobado y Sobresaliente
  - para aprobar, de Regular para arriba alcanza
  - para promocionar, hay que sacar Sobresaliente

Definimos entonces la Nota como un typeclass, sin métodos default ya que no podemos inferir la promoción en base a la aprobación o viceversa:

```
class Nota a where
  aprobo :: a -> Bool
  promociono :: a -> Bool
```

Ahora tenemos que definir la forma en la que trabajan las notas numéricas: Integer será una instancia posible para Nota:

```
instance Nota Integer where
  aprobo nota = nota >= 6
  promociono nota = nota >= 8
```

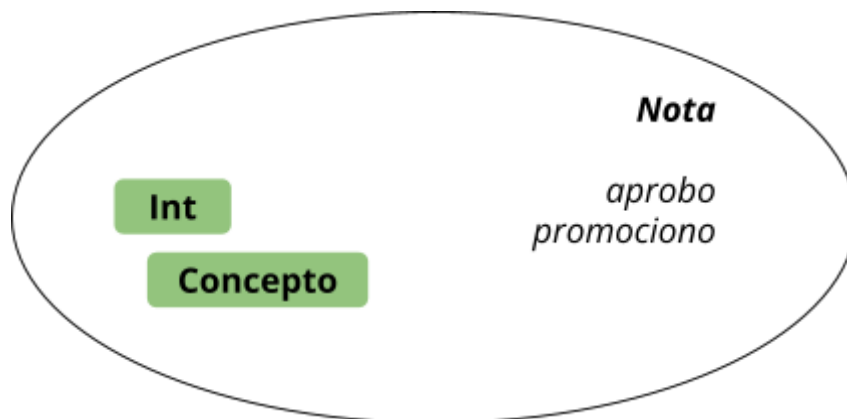
A su vez, definamos la nota de concepto como un Enum nuevo:

```
data Concepto = Insuficiente | Regular | Aprobado | Sobresaliente
  deriving (Eq, Ord, Show, Enum)
```

Y también agreguemos a Concepto como una instancia del typeclass Nota:

```
instance Nota Concepto where
  aprobo Insuficiente = False
  aprobo _             = True

  promociono Sobresaliente = True
  promociono _             = False
```



Esto permitirá que las funciones aprobo y promociono sean polimórficas:

```
λ aprobo 10
True
λ aprobo 4
False
λ aprobo Regular
True
λ promociono 7
False
```

```
λ promociono 10
True
λ promociono Sobresaliente
True
```

El polimorfismo es *ad-hoc*, como podemos comprobar al preguntar por el tipo de alguna de las funciones:

```
λ :t aprobo
aprobo :: Nota a => a -> Bool
```

Es importante recalcar que **el typeclass no define un tipo**, sino que provee un mecanismo para que uno o más tipos se asocien a él (los *a* que son justamente variables de tipo) y suscriban su contrato. Para más información pueden leer [la página de typeclasses de la wiki](#).

Entonces esta función

```
esBuen@ notas = all aprobo notas
```

¿qué tipo tiene?

## 6 Resumen

A lo largo de este capítulo hemos visto el poderoso sistema de tipos de Haskell, en donde

- todo valor tiene un tipo
- varios tipos se agrupan en typeclasses
  - un tipo puede pertenecer a varios typeclasses a la vez, eso marca las funciones que está obligado a definir y las funciones que toma de la definición default de la typeclass. Ejemplo: el valor 4.0 es de tipo Float, que pertenece a Ord y Num.
  - a su vez, cada typeclass tiene varios tipos, esto permite que haya funciones que acepten diferentes tipos como parámetro, lo que se conoce como **función polimórfica**. En el caso del typeclass Num, tiene como instancias Int, Integer, Natural, etc.
- un typeclass puede extender otro typeclass, como es el caso de Ord, que extiende la definición de Eq, para comparar por <, <=, > y >=. Por lo tanto todas las operaciones de Eq son válidas para Ord, además de las nuevas operaciones que el propio Ord defina.

Mediante este sistema de tipos y el pattern matching que cada tipo define, Haskell puede inferir el tipo de cualquier expresión

- esto libera al programador de hacer anotaciones de tipo para las funciones que él define
- por otro lado el chequeo estricto de tipos que posee hace que cualquier error se detecte lo más temprano posible

En resumen, el sistema de tipos es una herramienta que garantiza mayor robustez y consistencia a la definición de las abstracciones que hace el desarrollador dentro del paradigma funcional.