Inferencia y restricciones de tipo.

Conceptos clave:

* Polimorfismo paramétrico
* Familia de tipos (typeclass)
* Polimorfismo ad-hoc

## Introducción a la clase

Buenas..

Llegamos a la última clase del paradigma funcional (por lo menos la última donde van a ver algo nuevo creo), hoy les vamos estar a contando ese puchito que nunca les terminábamos de contar sobre inferencia de tipos. Vamos a ver un par de ejercicios que podrían aparecer eventualmente en algún parcial/final. Y en la segunda parte vamos a hablar un poco de como trabaja Haskell al momento de resolver sus funciones.

Ya a estas alturas habrán jugado un poco bastante con Haskell y con el **:t** y habrán notado que algunas funciones que nosotros las definíamos en clase de una forma…

1. **(+) :: Int -> Int -> Int**

Haskell nos las define de otra…

1. **(+) :: Num a => a -> a -> a**

Mismo cuando creábamos nuestros **data,** usábamos sin mucho conocimiento o les decíamos que usen *esto* porque sino algunas cosas nos las iban a poder probar…

1. **data Alumno = Alumno {**
2. **nombre :: String,**
3. **legajo :: Int**
4. **} deriving (Show, Eq, Ord)**

¿**Num**?¿**Show**?¿**Eq**? ¿Qué/Quiénes son estos muchachos? Bueno para poder presentarlos formalmente vamos a empezar viendo un par de funciones sencillitas..

## Tipos genéricos

**Función id ¿De qué tipo es?**

Ejemplos de consulta:

1. **> id True**
2. **True**
3. **> id “hola“**
4. **Hola**
5. **> id 1**
6. **1**

Funciona para Booleanos, Números, Strings. Entonces podemos definir su tipo como:

1. **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**.

Definamos la función **id**:

1. **id :: a -> a**
2. **id a = a**

Tenemos dos **a** diferentes:

Se refieren a variables de tipo, son utilizadas por el compilador de Haskell para determinar si el tipo es correcto.

Corresponden a variables para valores, son incógnitas que se resuelven en la aplicación de la función

**Hagamos lo mismo para las siguientes funciones:**

1. **head :: [a] -> a**
2. **tail :: [a] -> [a]**
3. **length :: [a] -> Int  --No es exactamente así pero meh..**

¿Qué restricciones tienen los elementos? **NINGUNA**

Definición de **length**

1. **length [] = 0**
2. **length (x:xs) = 1 + length xs**

Entonces **length** es una función cuyo dominio son listas de cualquier tipo y como imagen son los enteros.

Las funciones que aceptan valores de cualqueir tipo, sin ningún tipo de restricciones, tienen **polimorfismo paramétrico**.

**Lo interesante de esto es que solo necesitamos definir una vez la función para aplicarla a cualquier tipo.**

## ¿Qué pasa con la suma?

Suena razonablepensar que la función suma **(+)** pueda ser usada con enteros, decimales y fraccionales…

1. **> 2 + 3**
2. **5**
3. **> 2.8 + 3.1**
4. **5.9**
5. **> (2/3) + (1/3)**
6. **1.0**

Peeeero para poder hacer esto necesitamos pensar en cada caso puntual…

1. **(+) :: Int -> Int -> Int**
2. **(+) :: Float -> Float -> Int**
3. **(+) :: Fractional -> Fractional -> Fractional**

Ey!! ¿No era que las funciones solamente podían tener un único tipo? O es que el tipo de la suma es en realidad…

1. **(+) :: a -> a -> a**

Esto no es correcto porque no funciona para cualquier tipo, Booleanos y Funciones no pueden ser sumadas.

**Entonces ¿qué?**

Necesitamos cambiar nuestra definición de la funcion **(+)**, sin usar un tipo específico, sino uno genérico pero al que le vamos a aplicar una restricción:

1. **(+) :: (**Si **a** es un número) **entonces a -> a -> a**

**¿Cómo hacemos eso?...**

## Bienvenido al club

**Typeclass**

Un **typeclass** o también llamado **familia de tipos** define una *interfaz* la cual nos provee de un conjunto de operaciones que todos los tipos pertenecientes deben implementar. Adicionalmente, nos permite definir un comportamiento por defecto para ciertas operaciones.

Bien, vamos paso a paso…

**Definición de los Igualables**

Sisi, estaba con la suma y ahora pasé a los Eq, mi idea es ver como se define un typeclass en Haskell, cuales son ese conjunto de operaciones que debemos respetar y como es eso del comportamiento por defecto. Seguido de eso crear una instancia sencillita y volver a los Num.

Definamos nuestra primer familia de tipos los **Igualables:**

1. class **Igualable** a **where**
2. esIgual :: a -> a -> Bool
3. esDistinto :: a -> a -> Bool
4. -- Definición mínima completa: esIgual o esDistinto
5. esIgual x y = not (esDistinto x y)
6. esDistinto x y = not (esIgual x y)

Que tenemos hasta acá. Creamos la familia de tipos **Igualable**, donde nos dice que todo **a** para poder ser parte del club debe tener definidas las funciones **esIgual** y **esDistinto**.

Ya establecimos la interfaz, ahora hagamos que los booleanos sean parte del club.

Para esto los booleanos deben implementar aquellos metodos que el club exige.

1. **instance Igualable Bool where**
2. **esIgual True True = True**
3. **esIgual False False = True**
4. **esIgual \_ \_  = False**

Notese que solo definimos el comportamiento de la función **esIgual**, esto se debe a que en nuestros *default methods*, establecimos una en función de la otra. Por lo tanto cuando un Bool use el método **esDistinto** va a tomar la implementación definida en el typeclass.

A esto se lo conoce como **definición mínima completa** o **minimal complete definition** y establece el conjunto de operaciones que un tipo está obligado a implementar para poder ser una instancia del typeclass.

## Bueno ahora si, Num, Ord y Polimorfismo ad-hoc

Num, Ord, Eq, Show son familias de tipos. No me digas…

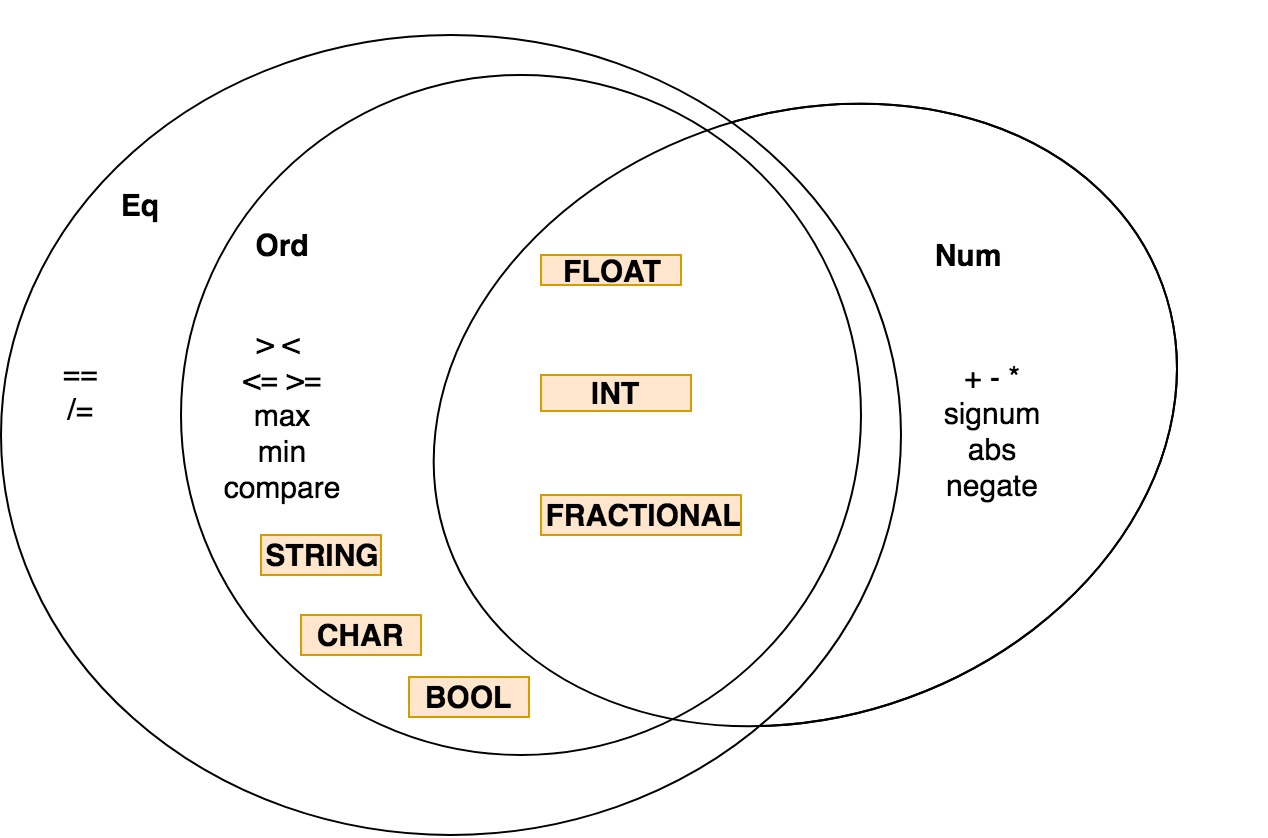
Sisi cada uno de estos tiene definidos su propia interfaz que hay que respetar para poder ser parte de la familia y tiene sus propias implementaciones por defecto.

Implementaciones necesarias Num:

1. **(+), (-), (\*) :: Num a => a -> a -> a**
2. **negate, abs, signum :: Num a => a -> a**

Implementaciones necesarias Ord:

1. **(<), (>), (<=), (>=) :: Ord a => a -> a -> Bool**
2. **max, min :: Ord a => a -> a -> a**



Pero entonces un Int y String pueden ser igualados.

Si claro pertenecen al mismo typeclass, por lo tanto implementan la función (==) y …

Mmm NO.