Tipos: polimorfismo y alto orden

Algoritmos y Estructuras de Datos I

Tipos

Haskell es un lenguaje de programación **fuertemente tipado** y **con tipado estático:**

- toda expresión tiene un tipo,
- los errores de tipo son detectados **antes de la ejecución de un programa**.

Permite evitar muchos errores de programación.

Tipos

```
(&&) :: Bool -> Bool -> Bool
```

La expresión `'a' && True` no está bien tipada y provocará un error en tiempo de **compilación**.

En lenguajes no tipados, errores como este no pueden identificarse antes de la ejecución.

Tipos

Haskell puede **inferir** el tipo de una expresión:

- el programador puede definir una función sin declarar su tipo,
- al compilar se corre un algoritmo que calcula el tipo, en caso que lo tenga, y dará un error en caso contrario.

Tipos

```
f = ‘a’ && True
```

provocará el siguiente error al compilar:

```
Couldn't match expected type `Bool' with actual type
`Char'
In the first argument of `(&&)', namely 'a'
In the expression: 'a' && True
In an equation for `f': f = 'a' && True
```

Tipos

- Básicos: **Int, Integer, Float, Double, Bool, Char, String, ...**
- Estructurados**: listas, tuplas, ...**

Spoiler alert*: en el segundo proyecto veremos cómo definir tipos nosotros mismos.*

Polimorfismo

Hay expresiones que pueden tener más de un tipo:

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

¿Qué tipo tiene la función id?

Polimorfismo

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

- **a** es una variable de tipo.
- Puede ser reemplazada por cualquier tipo concreto (**Char**, **Int, Bool…)**

Polimorfismo

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

puede tener tipo

- `Char -> Char`,
- `Bool -> Bool`,
- **`(Bool -> Bool) -> (Bool -> Bool)`**, etc.

Esto se llama **polimorfismo paramétrico**.

Polimorfismo paramétrico

- Una función polimórfica no conoce nada del tipo.
- Su comportamiento es independiente del tipo concreto con el que se use.

¿Cuantas funciones de tipo `a -> a` *hay?*

Polimorfismo paramétrico

- Una función polimórfica no conoce nada del tipo.
- Su comportamiento es independiente del tipo concreto con el que se use.

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

Polimorfismo *ad hoc*

- Una función que puede tener **distintos comportamientos** dependiendo del tipo concreto con que se use.
- Este polimorfismo se llama **ad hoc.**
- En Haskell se logra mediante **Type Classes.**

Polimorfismo *ad hoc*

```
elem :: a -> [a] -> Bool
```

Polimorfismo *ad hoc*

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

podrá utilizarse con un tipo que instancie la clase **`Eq`**.

Polimorfismo *ad hoc*

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

podrá utilizarse con un tipo que instancie la clase **`Eq`**.

- Una clase define requisitos que debe satisfacer un tipo.
- Por ejemplo, una instancia de **`Eq`** tendrá definidas las funciones de igualdad y desigualdad.

Polimorfismo *ad hoc*

- Utilizando **typeclasses** podemos definir funciones sobre tipos para los cuales pedimos algunas restricciones.
- El comportamiento dependerá de cómo el tipo defina las funciones especificadas en la clase.
- Algunas clases: **Eq**, **Ord**, **Show**, **Num**, ...

Currificación

Las funciones tienen un único parámetro!

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

Currificación

Las funciones tienen un único parámetro!

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

La **descurrificación** permite pensar en funciones de muchos parámetros

Aplicación parcial

```
max :: Ord a => a -> (a -> a)
max x y | x <= y = y
        | x > y = x

(max 4) :: Ord a => a -> a
```

Aplicación parcial

```
max :: Ord a => a -> (a -> a)
max x y | x <= y = y
        | x > y = x

(max 4) :: Ord a => a -> a
(max 4) y | 4 <= y = y
          | 4 > y = 4
```

Aplicación parcial

- Es la aplicación de una función con menos parámetros.
- Es una forma simple de crear nuevas funciones.
- Los operadores también pueden aplicarse parcialmente, usando paréntesis.

Alto orden, vieja

Las funciones devuelven funciones.

¿Pueden tomar funciones como parámetros?

```
applyTwice :: (a -> a) -> a -> a
```

Alto orden, vieja

Las funciones devuelven funciones.

¿Pueden tomar funciones como parámetros?

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

Map y *Filter*

```
map :: (a -> b) -> [a] -> [b]

filter :: (a -> Bool) -> [a] -> [a]
```

Qué leer para aprender más:

- https://wiki.haskell.org/Polymorphism
- http://learnyouahaskell.com/types-and-typeclasses
- http://aprendehaskell.es (cap. 3 y 6)