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. (Minúscula)
- 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.

¿Cuántas funciones "distintas" de tipo `a -> a` *hay?*

Polimorfismo paramétrico

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

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

- ¿Conocen alguna otra?
- ¿Hay otros tipos de polimorfismos? Si, ***ad hoc***.

Currificación

Las funciones tienen un único parámetro!

Acá requiero que el tipo **a** tenga definido un orden, es decir, un **<**.
¡Spoiler! Lo veremos en Proy. 2

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

Al pensar una función con **currificación**, el lado derecho es el la imagen de la función ...

Currificación

Las funciones tienen un único parámetro!

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

La **descurrificación** permite pensar en funciones de muchos parámetros. Si pienso **`max`** de esta manera, toma dos elementos de tipo a y devuelve un a

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]
```

Definición en el *preludio* de la función

T

```
map f [] = []
map f (x:xs) = f x : map f xs
```

`map y filter`

La función `map`:

- Toma 2 argumentos, uno de los cuales es una función
- Aplica f a cada elemento de xs
- El resultado es una lista con la aplicación en el mismo orden

map y filter:

ejemplo de uso de `map`:

```
Prelude> map succ [1,2,3,4]
[2,3,4,5]
Prelude> map not [False, False, True]
[True,True,False]
```

map y filter

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

Definición en el *preludio* de la función

```
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

`map y filter`

La función `filter`:

- Toma 2 argumentos, uno de los cuales es un predicado
- El resultado es una lista con los elementos que cumplen el predicado

map y filter:

ejemplo de uso de `filter`:

```
Prelude> filter (<2) [3,1,0,6,9]
[1,0]
Prelude> filter even [8,2,3,6,11]
[8,2,6]
```

Qué leer para aprender más:

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