Tipos, currificación y alto orden

Programación funcional

Repaso

La clase pasada vimos:

- Expresiones
- Reducción

Declaración de tipos:

```
isWeekend :: Day -> Bool

not :: Bool -> Bool

inc :: Nat -> Nat

type String = [Char]
```

Visión denotacional:

- **Valores:** elementos abstractos.
- **Expresiones:** construcciones sintácticas (correctas) que denotan valores.
- **Tipos:** conjuntos de valores con propiedades comunes.

Tipado

Tipos básicos:

- Datos: `A`
- Funciones: `A -> B`

Ejemplo: `isWeekend :: Day -> Bool`

Ventajas:

- Detección de errores.
- Especificación rudimentaria.
- Optimización durante la compilación.

Polimorfismo paramétrico

¿Qué tipo tiene la función `id x = x`?

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

Donde `a` es un parámetro de tipo.

Bottom

¿Qué tipo tiene `bottom`?

```
bottom = bottom
```

Reglas de tipado

Aplicando las siguientes reglas podemos hacer chequeo e inferencia:

$$\frac{\texttt{f :: A -> B} \qquad \texttt{x :: A}}{\texttt{f x :: B}}$$

$$\frac{\texttt{C1, ..., Cn :: A} \qquad \texttt{T1, ..., Tn :: B}}{\texttt{case Ci of \{C1 -> T1; ...; Cn -> Tn\} :: B}}\ 1$$

! Una expresión que no tipa es inválida.

[1] Abuso del lenguaje entre expresiones y patrones.

Typeclasses

¿Qué tipo tiene el operador (/=)?

```
x /= y = not (x == y)
```

Tenemos que pedir que x e y sean comparables por (==).

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

Entonces:

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

Funciones como valores

! Las funciones son valores, al igual que los números o los booleanos.

Funciones como valores

! Las funciones son valores, al igual que los números o los booleanos.

¿Qué implica?

- Pueden ser argumento de otras funciones.
- Pueden ser resultado de otras funciones.
- Pueden almacenarse en estructuras de datos.
- Pueden representar (estructuras de) datos.

Funciones como valores

! Las funciones son valores, al igual que los números o los booleanos.

¿Qué implica?

- Pueden ser argumento de otras funciones.
- Pueden ser resultado de otras funciones.
- Pueden almacenarse en estructuras de datos.
- Pueden representar (estructuras de) datos.

Llamamos funciones de alto orden a las funciones que:

- toman funciones como parámetro; o
- retornan funciones como resultado.

Funciones como valores

! Las funciones son valores, al igual que los números o los booleanos.

¿Qué implica?

- Pueden ser argumento de otras funciones.
- Pueden ser resultado de otras funciones.
- Pueden almacenarse en estructuras de datos.
- Pueden representar (estructuras de) datos.

Llamamos funciones de alto orden a las funciones que:

- toman funciones como parámetro; o
- retornan funciones como resultado.

Ejemplos:

```
const :: a -> (b -> a)
twice :: (a -> a) -> (a -> a)
flip  :: (a -> b -> c) -> (b -> a -> c)
```

Currificación

Haskell Brooks Curry

¿Cómo definimos funciones que toman múltiples parámetros?

Con funciones que retornan funciones:

```
const :: a -> (b -> a)
const x = \y -> x

apply :: (a -> b) -> (a -> b)
apply f = \x -> f x

(.) :: (a -> b) -> (b -> c) -> (a -> c)
(.) f = \g -> \x -> f (g x)
```

Asociatividad

¿Podemos ahorrarnos paréntesis?

La aplicación de funciones asocia a izquierda:

```
(const False) True  ≡  const False True
```

El tipo de las funciones asocia a derecha:

```
const :: a -> (b -> a)  ≡  const :: a -> b -> a
```

Ejemplos:

```
twice :: (a -> a) -> a -> a
flip  :: (a -> b -> c) -> b -> a -> c
apply :: (a -> b) -> a -> b
(.)   :: (a -> b) -> (b -> c) -> a -> c
```

Point-free

Consideramos la siguiente equivalencia:

```
(\x -> f x)  ≡  f
```

Ejemplo:

```
inc = \n -> 1 + n  ≡  inc = (1+)
```

Interpretación

¿Está currificada la siguiente función?

```
distance (x,y) = sqrt (x*x + y*y)
```

¿Podemos definir una versión que tome `x` e `y` separadas?

```
distance' x y = sqrt (x*x + y*y)
```

¿Qué representa la aplicación parcial?

```
distance' 0
(1+) o (+) 1
(/2) o flip (/) 2
```

¿Qué pasa con el argumento?

Currificar o no currificar esa es la cuestión

Desafío

Defina las siguientes funciones:

```
curry :: ((a,b) -> c) -> a -> b -> c

uncurry :: (a -> b -> c) -> (a,b) -> c
```