Programación Funcional

Introducción a los tipos de datos en Haskell

Mariano Rean

Universidad Nacional de Hurlingham

Tipos de datos

- Un tipo es una colección de valores relacionados.

Tipos de datos

- Un tipo es una colección de valores relacionados.
- Por ejemplo, el tipo `Bool` representa los valores de verdad `True` y `False`.

Tipos de datos

- Un tipo es una colección de valores relacionados.
- Por ejemplo, el tipo `Bool` representa los valores de verdad `True` y `False`.
- Escribimos `v :: T` para representar que `v` es un valor de tipo `T`.

Tipos de datos

- Un tipo es una colección de valores relacionados.
- Por ejemplo, el tipo `Bool` representa los valores de verdad `True` y `False`.
- Escribimos `v :: T` para representar que `v` es un valor de tipo `T`.
- Las funciones también tienen tipo, y escribimos `f :: A -> B` para el tipo de las funciones de `A` en `B`, donde `A` y `B` son tipos.

Tipos de datos

- Un tipo es una colección de valores relacionados.
- Por ejemplo, el tipo `Bool` representa los valores de verdad `True` y `False`.
- Escribimos `v :: T` para representar que `v` es un valor de tipo `T`.
- Las funciones también tienen tipo, y escribimos `f :: A -> B` para el tipo de las funciones de `A` en `B`, donde `A` y `B` son tipos.
- Por ejemplo, `not :: Bool -> Bool` es el tipo de una función que calcula la negación de un valor booleano.

Tipos de datos

- Un tipo es una colección de valores relacionados.
- Por ejemplo, el tipo `Bool` representa los valores de verdad `True` y `False`.
- Escribimos `v :: T` para representar que `v` es un valor de tipo `T`.
- Las funciones también tienen tipo, y escribimos `f :: A -> B` para el tipo de las funciones de `A` en `B`, donde `A` y `B` son tipos.
- Por ejemplo, `not :: Bool -> Bool` es el tipo de una función que calcula la negación de un valor booleano.
- **Regla de inferencia**: Si `f :: A -> B` y `x :: A` entonces `f x :: B`.
- Esta regla se generaliza a más argumentos, por ejemplo la función que suma dos enteros tiene el tipo `suma :: Int -> Int -> Int`, por lo tanto `suma 2 3 :: Int`.

Tipos de datos

En GHCI podemos ver el tipo de un valor o expresión usando `:t`.

Tipos de datos

En GHCI podemos ver el tipo de un valor o expresión usando `:t`.

```
*Main> :t True
True :: Bool
```

Tipos de datos

En GHCI podemos ver el tipo de un valor o expresión usando `:t`.

```
*Main> :t True
True :: Bool
```

```
*Main> :t 3^48 < 2^23
3^48 < 2^23 :: Bool
```

Tipos de datos

En GHCI podemos ver el tipo de un valor o expresión usando `:t`.

```
*Main> :t True
True :: Bool
```

```
*Main> :t 3^48 < 2^23
3^48 < 2^23 :: Bool
```

```
maximo :: Int -> Int -> Int
maximo x y | x >= y     = x
           | otherwise = y
```

Tipos de datos

En GHCI podemos ver el tipo de un valor o expresión usando `:t`.

```
*Main> :t True
True :: Bool
```

```
*Main> :t 3^48 < 2^23
3^48 < 2^23 :: Bool
```

```
maximo :: Int -> Int -> Int
maximo x y | x >= y     = x
           | otherwise = y
```

```
*Main> :t maximo
maximo :: Int -> Int -> Int
```

Tipos de datos

En GHCI podemos ver el tipo de un valor o expresión usando `:t`.

```
*Main> :t True
True :: Bool
```

```
*Main> :t 3^48 < 2^23
3^48 < 2^23 :: Bool
```

```
maximo :: Int -> Int -> Int
maximo x y | x >= y     = x
           | otherwise = y
```

```
*Main> :t maximo
maximo :: Int -> Int -> Int
```

```
*Main> :t maximo 5 6
maximo 5 6 :: Int
```

Clases de tipos

Clase de tipos

Un **conjunto de tipos de datos** a los que se les puede aplicar un **conjunto de funciones**.

Clases de tipos

Clase de tipos

Un **conjunto de tipos de datos** a los que se les puede aplicar un **conjunto de funciones**.

Algunas clases:

1. `Integral` $:=$ ($\{$ `Int, Integer, ...` $\}$, $\{$ `mod, div, ...` $\}$)

Clases de tipos

Clase de tipos

Un **conjunto de tipos de datos** a los que se les puede aplicar un **conjunto de funciones**.

Algunas clases:

1. `Integral` $:=$ ($\{$ `Int, Integer, ...` $\}$, $\{$ `mod, div, ...` $\}$)
2. `Fractional` $:=$ ($\{$ `Float, Double, ...` $\}$, $\{$ `(/), ...` $\}$)

Clases de tipos

Clase de tipos

Un **conjunto de tipos de datos** a los que se les puede aplicar un **conjunto de funciones**.

Algunas clases:

1. `Integral` $:=$ ($\{$ `Int, Integer, ...` $\}$, $\{$ `mod, div, ...` $\}$)
2. `Fractional` $:=$ ($\{$ `Float, Double, ...` $\}$, $\{$ `(/), ...` $\}$)
3. `Floating` $:=$ ($\{$ `Float, Double, ...` $\}$, $\{$ `sqrt, sin, cos, tan, ...` $\}$)

Clases de tipos

Clase de tipos

Un **conjunto de tipos de datos** a los que se les puede aplicar un **conjunto de funciones**.

Algunas clases:

1. `Integral` $:=$ $(\{$ `Int, Integer, ...` $\}, \{$ `mod, div, ...` $\})$
2. `Fractional` $:=$ $(\{$ `Float, Double, ...` $\}, \{$ `(/), ...` $\})$
3. `Floating` $:=$ $(\{$ `Float, Double, ...` $\}, \{$ `sqrt, sin, cos, tan, ...` $\})$
4. `Num` $:=$ $(\{$ `Int, Integer, Float, Double, ...` $\}, \{$ `(+), (*), abs, ...` $\})$

Clases de tipos

Clase de tipos

Un **conjunto de tipos de datos** a los que se les puede aplicar un **conjunto de funciones**.

Algunas clases:

1. `Integral` $:=$ ($\{$ `Int, Integer, ...` $\}$, $\{$ `mod, div, ...` $\}$)
2. `Fractional` $:=$ ($\{$ `Float, Double, ...` $\}$, $\{$ `(/), ...` $\}$)
3. `Floating` $:=$ ($\{$ `Float, Double, ...` $\}$, $\{$ `sqrt, sin, cos, tan, ...` $\}$)
4. `Num` $:=$ ($\{$ `Int, Integer, Float, Double, ...` $\}$, $\{$ `(+), (*), abs, ...` $\}$)
5. `Ord` $:=$ ($\{$`Bool, Int, Integer, Float, Double, ...` $\}$, $\{$ `(<=), compare` $\}$)

Clases de tipos

Clase de tipos

Un **conjunto de tipos de datos** a los que se les puede aplicar un **conjunto de funciones**.

Algunas clases:

1. `Integral` $:=$ ($\{$ `Int, Integer, ...` $\}$, $\{$ `mod, div, ...` $\}$)
2. `Fractional` $:=$ ($\{$ `Float, Double, ...` $\}$, $\{$ `(/), ...` $\}$)
3. `Floating` $:=$ ($\{$ `Float, Double, ...` $\}$, $\{$ `sqrt, sin, cos, tan, ...` $\}$)
4. `Num` $:=$ ($\{$ `Int, Integer, Float, Double, ...` $\}$, $\{$ `(+), (*), abs, ...` $\}$)
5. `Ord` $:=$ ($\{$`Bool, Int, Integer, Float, Double, ...` $\}$, $\{$ `(<=), compare` $\}$)
6. `Eq` $:=$ ($\{$ `Bool, Int, Integer, Float, Double, ...` $\}$, $\{$ `(==), (/=)` $\}$)

Clases de tipos

Supongamos que definimos la función `triple x = 3 * x` sin definir su tipo. ¿Qué tipos de datos admite esta función?

Clases de tipos

Supongamos que definimos la función `triple x = 3 * x` sin definir su tipo. ¿Qué tipos de datos admite esta función?

¿Qué pasa con...? ¿funcionan?

- `triple 2`

Clases de tipos

Supongamos que definimos la función `triple x = 3 * x` sin definir su tipo. ¿Qué tipos de datos admite esta función?

¿Qué pasa con...? ¿funcionan?

- `triple 2` $\rightsquigarrow$ `6`

Clases de tipos

Supongamos que definimos la función `triple x = 3 * x` sin definir su tipo. ¿Qué tipos de datos admite esta función?

¿Qué pasa con...? ¿funcionan?

- `triple 2` $\rightsquigarrow$ `6`
- `triple 2.5`

Clases de tipos

Supongamos que definimos la función `triple x = 3 * x` sin definir su tipo. ¿Qué tipos de datos admite esta función?

¿Qué pasa con...? ¿funcionan?

- `triple 2` $\rightsquigarrow$ `6`
- `triple 2.5` $\rightsquigarrow$ `7.5`

Clases de tipos

Supongamos que definimos la función `triple x = 3 * x` sin definir su tipo. ¿Qué tipos de datos admite esta función?

¿Qué pasa con...? ¿funcionan?

- `triple 2` $\rightsquigarrow$ `6`
- `triple 2.5` $\rightsquigarrow$ `7.5`
- `triple True`

Clases de tipos

Supongamos que definimos la función `triple x = 3 * x` sin definir su tipo. ¿Qué tipos de datos admite esta función?

¿Qué pasa con...? ¿funcionan?

- `triple 2` $\rightsquigarrow$ `6`
- `triple 2.5` $\rightsquigarrow$ `7.5`
- `triple True` ¡error de tipos!

```
No instance for (Num Bool) arising from a use of ‘*’
```

Clases de tipos

Supongamos que definimos la función `triple x = 3 * x` sin definir su tipo. ¿Qué tipos de datos admite esta función?

¿Qué pasa con...? ¿funcionan?

- `triple 2` $\rightsquigarrow$ `6`
- `triple 2.5` $\rightsquigarrow$ `7.5`
- `triple True` ¡error de tipos!

  ```
  No instance for (Num Bool) arising from a use of ‘*’
  ```

```
1 *Main> :t triple
2 triple :: Num a => a -> a
```

Clases de tipos

Supongamos que definimos la función `triple x = 3 * x` sin definir su tipo. ¿Qué tipos de datos admite esta función?

¿Qué pasa con...? ¿funcionan?

- `triple 2` $\rightsquigarrow$ `6`
- `triple 2.5` $\rightsquigarrow$ `7.5`
- `triple True` ¡error de tipos!

  ```
  No instance for (Num Bool) arising from a use of ‘*’
  ```

```
1 *Main> :t triple
2 triple :: Num a => a -> a
```

¿Qué significa `Num a => ...` ?

Clases de tipos

Supongamos que definimos la función `triple x = 3 * x` sin definir su tipo. ¿Qué tipos de datos admite esta función?

¿Qué pasa con...? ¿funcionan?

- `triple 2` $\rightsquigarrow$ `6`
- `triple 2.5` $\rightsquigarrow$ `7.5`
- `triple True` ¡error de tipos!

  ```
  No instance for (Num Bool) arising from a use of ‘*’
  ```

```
1 *Main> :t triple
2 triple :: Num a => a -> a
```

¿Qué significa `Num a => ...` ?

Acá `a` es una *variable de tipo*, y `Num a => ...` quiere decir que `a` puede ser cualquier tipo que pertenezca a la clase `Num`.

Clases de tipos

Supongamos que definimos la función `triple x = 3 * x` sin definir su tipo. ¿Qué tipos de datos admite esta función?

¿Qué pasa con...? ¿funcionan?

- `triple 2` $\rightsquigarrow$ `6`
- `triple 2.5` $\rightsquigarrow$ `7.5`
- `triple True` ¡error de tipos!

  ```
  No instance for (Num Bool) arising from a use of ‘*’
  ```

```
1 *Main> :t triple
2 triple :: Num a => a -> a
```

¿Qué significa `Num a => ...` ?

Acá `a` es una *variable de tipo*, y `Num a => ...` quiere decir que `a` puede ser cualquier tipo que pertenezca a la clase `Num`.

En resumen, la función `triple` solo admite tipos de datos numéricos.

Funciones polimórficas

Es posible definir funciones usando variables de tipo para que puedan funcionar sobre muchos tipos de datos.

Funciones polimórficas

Es posible definir funciones usando variables de tipo para que puedan funcionar sobre muchos tipos de datos.

Por ejemplo:

```
1 identidad :: a -> a
2 identidad x = x
```

Funciones polimórficas

Es posible definir funciones usando variables de tipo para que puedan funcionar sobre muchos tipos de datos.

Por ejemplo:

```
1 identidad :: a -> a
2 identidad x = x
```

Notar que a va necesariamente en minúscula.

Funciones polimórficas

Es posible definir funciones usando variables de tipo para que puedan funcionar sobre muchos tipos de datos.

Por ejemplo:

```
1 identidad :: a -> a
2 identidad x = x
```

Notar que a va necesariamente en minúscula.

En Haskell esa función ya existe, se llama `id` y vale para cualquier tipo de datos.

```
1 *Main> :t id
2 id :: a -> a
```

Funciones polimórficas

Es posible definir funciones usando variables de tipo para que puedan funcionar sobre muchos tipos de datos.

Por ejemplo:

```
1 identidad :: a -> a
2 identidad x = x
```

Notar que a va necesariamente en minúscula.

En Haskell esa función ya existe, se llama `id` y vale para cualquier tipo de datos.

```
1 *Main> :t id
2 id :: a -> a
```

Podemos definir nosotros, por ejemplo:

```
1 cuadruple :: Num a => a -> a
2 cuadruple x = 4 * x
```

Nueva familia de tipos: Tuplas

- Dados dos tipos de datos `A` y `B`, podemos crear el tipo de datos `(A, B)` que representa pares ordenados de elementos, donde el primero es de tipo `A` y el segundo es de tipo `B`. Por ejemplo `(Int, Int)`, `(Float, Int)`, `(Bool, (Float, Int))`.

Nueva familia de tipos: Tuplas

- Dados dos tipos de datos `A` y `B`, podemos crear el tipo de datos `(A, B)` que representa pares ordenados de elementos, donde el primero es de tipo `A` y el segundo es de tipo `B`. Por ejemplo `(Int, Int)`, `(Float, Int)`, `(Bool, (Float, Int))`.
- Algunas funciones para tuplas de dos elementos:

Nueva familia de tipos: Tuplas

- Dados dos tipos de datos `A` y `B`, podemos crear el tipo de datos `(A, B)` que representa pares ordenados de elementos, donde el primero es de tipo `A` y el segundo es de tipo `B`. Por ejemplo `(Int, Int)`, `(Float, Int)`, `(Bool, (Float, Int))`.
- Algunas funciones para tuplas de dos elementos:
  - `fst :: (a, b) -> a` Ejemplo de uso: `fst (1 + 4, 2)` $\rightsquigarrow$ `5`

Nueva familia de tipos: Tuplas

- Dados dos tipos de datos `A` y `B`, podemos crear el tipo de datos `(A, B)` que representa pares ordenados de elementos, donde el primero es de tipo `A` y el segundo es de tipo `B`. Por ejemplo `(Int, Int)`, `(Float, Int)`, `(Bool, (Float, Int))`.
- Algunas funciones para tuplas de dos elementos:
  - `fst :: (a, b) -> a` Ejemplo de uso: `fst (1 + 4, 2)` $\rightsquigarrow$ `5`
  - `snd :: (a, b) -> b` Ejemplo de uso: `snd (1, (2, 3))` $\rightsquigarrow$ `(2, 3)`

Nueva familia de tipos: Tuplas

- Dados dos tipos de datos `A` y `B`, podemos crear el tipo de datos `(A, B)` que representa pares ordenados de elementos, donde el primero es de tipo `A` y el segundo es de tipo `B`. Por ejemplo `(Int, Int)`, `(Float, Int)`, `(Bool, (Float, Int))`.
- Algunas funciones para tuplas de dos elementos:
  - `fst :: (a, b) -> a` Ejemplo de uso: `fst (1 + 4, 2)` $\rightsquigarrow$ `5`
  - `snd :: (a, b) -> b` Ejemplo de uso: `snd (1, (2, 3))` $\rightsquigarrow$ `(2, 3)`
- Podemos crear también ternas, cuaternas, etc...

Ejercicios

Implementar las siguientes funciones, especificando su tipo.

1. `sumaTerna`: dada una terna de enteros, calcula la suma de sus tres elementos.
2. `todoMenor`: dadas dos ternas de números reales, decide si es cierto que cada coordenada de la primera es menor a la coordenada correspondiente de la segunda.
3. `posicPrimerPar`: dada una terna de enteros, devuelve la posición del primer número par si es que hay alguno, y devuelve 4 si son todos impares.
4. `crearPar :: a -> b -> (a, b)`: crea un par a partir de sus dos componentes dadas por separado (debe funcionar para elementos de cualquier tipo).
5. `invertir :: (a, b) -> (b, a)`: invierte los elementos del par pasado como parámetro (debe funcionar para elementos de cualquier tipo).