Tema 4. Funciones de orden superior

4.1 Funciones de orden superior

✓ Una función tal que alguno de sus argumentos es una función o que devuelve una función como resultado.

✓ Son útiles porque permiten capturar esquemas de cómputo generales (*abstracción*).

Ejemplo:

```
dosVeces :: (Integer → Integer) → Integer → Integer
dosVeces f x = f (f x)

inc :: Integer → Integer
inc x = x + 1

dec :: Integer → Integer
dec x = x + 1
```

✓ El primer argumento de $dosVeces$ debe ser una función con tipo $Integer \rightarrow Integer$.

✓ Los paréntesis en el tipo de $dosVeces$ son **obligatorios**.

Uso

```
? dosVeces inc 10
12 :: Integer

? dosVeces dec 10
8 :: Integer
```

4.2 Expresiones lambda

✓ Permiten definir funciones *anónimas* (sin nombre).

Ejemplo:

$\lambda\ x \rightarrow x\ +\ 1$ denota en Haskell la función que toma un argumento (x) y lo devuelve incrementado.

```
? λ x → x + 1
≪function≫  :: Integer → Integer

? (λ x → x + 1) 10
11  :: Integer
```

Paso a paso:

$(\lambda\ x \rightarrow x\ +\ 1)\ 10$

$\Longrightarrow$ {Sustitutyendo el argumento x por 10}

$10\ +\ 1$

$\Longrightarrow$ {por $(+)$}

11

✓ Funciones de más de un argumento con la notación lambda:

```
? (λ x y → x + y)
≪function≫  :: Integer → Integer → Integer

? (λ x y → x + y) 5 7
12  :: Integer
```

✓ Son útiles como argumentos de funciones de orden superior:

```
? dosVeces (λ x → x + 1) 10
12  :: Integer

? dosVeces (λ x → x - 1) 10
8  :: Integer

? dosVeces (λ x → x * 2) 10
40  :: Integer
```

4.3 Aplicación parcial

✓ Permite aplicar a una función menos argumentos de los que tiene para obtener una nueva función

Aplicación parcial o **parcialización**: Si f es una función de n argumentos y se le aplican $k \leq n$ argumentos con los tipos adecuados, se obtiene como resultado una nueva función que espera los $n - k$ argumentos restantes.

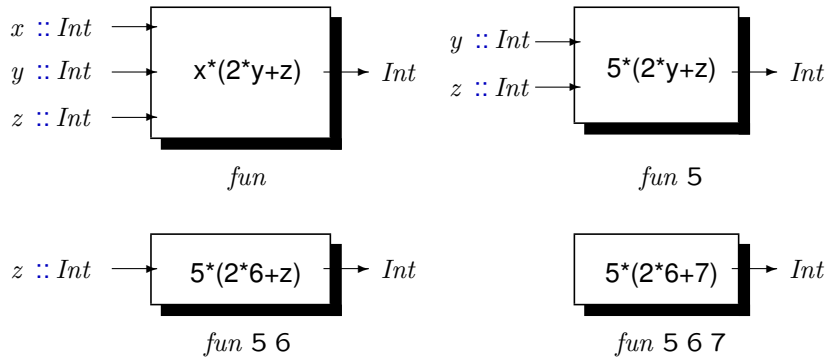
Regla de la cancelación

si	$f :: t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t_r$
y	$e_1 :: t_1,\ e_2 :: t_2 \ldots, e_k :: t_k$ con $(k \leq n)$
entonces	$f\ e_1\ e_2\ \ldots\ e_k :: t_{k+1} \rightarrow t_{k+2} \ldots \rightarrow t_n \rightarrow t_r$

Ejemplo: A la siguiente función de tres argumentos:

$$\begin{array}{lcl} fun & :: & Int \rightarrow Int \rightarrow Int \rightarrow Int \\ fun\ x\ y\ z & = & x * (2 * y + z) \end{array}$$

es posible aplicarle uno, dos o tres argumentos. En cada caso obtenemos una función con un argumento menos.



Aplicación parcial (2)

Todo esto funciona gracias a los siguientes convenios

Definiciones de funciones

$$f\,x_1 x_2 \ldots x_n = e \quad \not\Longrightarrow \quad f = \lambda x_1 x_2 \ldots x_n \to e$$

Regla de λ-abstracciones

$$\lambda x_1 x_2 \ldots x_n \to e \quad \not\Longrightarrow \quad \lambda x_1 \to (\lambda x_2 \to (\ldots \to (\lambda x_n \to e)))$$

Asociatividad a la derecha de $(\to)$ (En Tipos)

$$t_1 \to t_2 \to \ldots t_n \quad \not\Longrightarrow \quad (t_1 \to (t_2 \to (\ldots \to t_n)))$$

Asociatividad izquierda de la aplicación de funciones

$$f\,a_1 a_2 \ldots a_n \quad \not\Longrightarrow \quad (((f\ a_1)\ a_2) \ldots\ a_n)$$

Consideremos la función anterior:

$$\begin{array}{lll} fun & :: & Int \to Int \to Int \to Int \\ fun\ x\ y\ z & = & x * (2 * y + z) \end{array}$$

Por las reglas anteriores la definición es equivalente a:

$$\begin{array}{lll} fun & :: & Int \to (Int \to (Int \to Int)) \\ fun & = & \lambda\ x \to (\lambda\ y \to (\lambda\ z \to x * (2 * y + z))) \end{array}$$

Aplicación parcial (3)

$$\begin{array}{lcl} fun & :: & Int \to (Int \to (Int \to Int)) \\ fun & = & \lambda\, x \to (\lambda\, y \to (\lambda\, z \to x * (2 * y + z))) \end{array}$$

Así,

$$\begin{array}{l} \underline{fun\ 5} \\ \Longrightarrow \{\text{por definición de } fun\} \\ \underline{\lambda\, x \to (\lambda\, y \to (\lambda\, z \to x * (2 * y + z)))\ 5} \\ \Longrightarrow \{\text{sustituyendo } x \text{ por } 5\} \\ \lambda\, y \to (\lambda\, z \to 5 * (2 * y + z)) \\ \Longrightarrow \{\text{regla de las } \lambda\text{-abstracciones}\} \\ \lambda\, y\ z \to 5 * (2 * y + z) \end{array}$$

que tiene tipo $Int \to Int \to Int$

También,

$$\begin{array}{l} fun\ 5\ 6 \\ \Longrightarrow \{\text{asociatividad izq de aplicación}\} \\ \underline{(fun\ 5)}\ 6 \\ \Longrightarrow \{\text{por reducción anterior}\} \\ \underline{(\lambda\, y \to (\lambda\, z \to 5 * (2 * y + z)))\ 6} \\ \Longrightarrow \{\text{sustituyendo } y \text{ por } 6\} \\ \lambda\, z \to 5 * (2 * 6 + z) \end{array}$$

que tiene tipo $Int \to Int$

Por último,

$$\begin{array}{l} fun\ 5\ 6\ 7 \\ \Longrightarrow \{\text{asociatividad izq de aplicación}\} \\ \underline{(fun\ 5)\ 6}\ 7 \\ \Longrightarrow \{\text{por reducción anterior}\} \\ (\lambda\, z \to 5 * (2 * 6 + z))\ 7 \\ \Longrightarrow \{\text{sustituyendo } z \text{ por } 7\} \\ 5 * (2 * 6 + 7) \\ \Longrightarrow \{\text{por aritmética}\} \\ 95 \end{array}$$

Secciones

- ✓ Los operadores pueden ser aplicados parcialmente
- ✓ Se obtienen funciones de un argumento

Si $(\bigstar)$ es un operador tenemos las siguientes equivalencias (los paréntesis son obligatorios):

Secciones de operadores

$$(x\ \bigstar) \Longrightarrow \lambda\ y \rightarrow x \bigstar y$$
$$(\bigstar\ y) \Longrightarrow \lambda\ x \rightarrow x \bigstar y$$
$$(\bigstar) \Longrightarrow \lambda\ x\ y \rightarrow x \bigstar y$$

Ejemplos:

- $(2.0/)$ Toma un valor real x y devuelve $2.0/x$.
- $(/2.0)$ Toma un valor real x y devuelve $x/2.0$.
- $(/)$ Toma dos valores reales y devuelve su cociente.
- (> 2) Toma un argumento y devuelve *True* si es mayor que 2.
- $(2 >)$ Toma un argumento y devuelve *True* si es menor que 2.

```
? (/2.0) 8.0
4.0 :: Double

? dosVeces (+1) 10
12 :: Integer
```

- ✓ Excepción: $(-e)$ donde e es una expresión **NO es una sección**

4.4 Ejemplo: una función de orden superior para enteros

Muchas funciones sobre enteros siguen el siguiente esquema:

- Caso base: cuando el argumento es 0
- Paso recursivo: se calcula el resultado para $n + 1$ a partir del resultado para n

Ejemplos:

```
factorial               :: Integer → Integer
factorial 0             =  1
factorial m@(n + 1)     =  (*) m (factorial n)

sumatorio               :: Integer → Integer
sumatorio 0             =  0
sumatorio m@(n + 1)     =  (+) m (sumatorio n)
```

Ambas siguen el esquema:

```
fun                 :: Integer → Integer
fun 0               =  e
fun m@(n + 1)       =  f m (fun n)
```

Función de orden superior que lo captura:

```
iter        :: (Integer → Integer → Integer) → Integer →
               (Integer → Integer)
iter f e    =  fun
  where
    fun             :: Integer → Integer
    fun 0           =  e
    fun m@(n + 1)   =  f m (fun n)
```

Ahora es posible una definición más compacta:

```
factorial'  :: Integer → Integer
factorial'  =  iter (*) 1

sumatorio'  :: Integer → Integer
sumatorio'  =  iter (+) 0
```

Objetivos del tema

El alumno debe:

- ✓ Conocer el concepto de función de orden superior
- ✓ Saber definir y utilizar funciones de orden superior
- ✓ Saber utilizar lambda expresiones
- ✓ Conocer el concepto de aplicación parcial y los convenios que hacen que tenga sentido
- ✓ Saber construir nuevas funciones aplicando parcialmente otras
- ✓ Conocer el tipo y el significado de una expresión construída mediante una aplicación parcial
- ✓ Entender que es posible capturar un patrón de cómputo habitual mediante una función de orden superior (*abstracción*) y cómo definir casos concretos de dicho patrón (*concreción*)