Temario

1 Introducción y semántica operacional

2 Tipos predefinidos

3 Patrones y Definiciones de Funciones

4 Funciones de orden superior

5 Polimorfismo

6 Definiciones de tipos

7 El sistema de clases

8 Listas

9 Árboles

10 Razonamiento ecuacional

Bibliografía

✓ *Razonando con Haskell. Un curso sobre programación funcional.* Blas Ruiz, Francisco Gutiérrez, Pablo Guerrero y José Gallardo. Thomson, 2004. (`http://www.lcc.uma.es/RazonandoConHaskell`)

✓ *Introduction to Functional Programming using Haskell.* Richard Bird. Prentice Hall, 1998.

✓ *The Haskell School of Expression. Learning Functional Programming through multimedia.* Paul Hudak. Cambridge University Press, 2000.

✓ *Haskell. The Craft of Functional Programming.* Simon Thompson. Addison-Wesley, 1999.

Profesor

Pepe Gallardo.

Despacho 3.2.50.

`pepeg@lcc.uma.es`

Web asignatura

`http://www.lcc.uma.es/~pepeg/mates`

Tema 1. Introducción y semántica operacional

1.1 Programación Funcional

1.2 El lenguaje Haskell

1.3 La notación Currificada

Aplicación de funciones

Definición de funciones

1.4 Sesiones y declaraciones

1.5 Reducción de expresiones

Orden de reducción aplicativo

Orden de reducción normal

Evaluación Perezosa

1.1 Programación Funcional

- ✓ *Programar*: especificar cómo resolver un problema.
- ✓ Un modo natural de describir programas es mediante funciones.
- ✓ El estilo funcional está basado en expresiones:
  - ◇ Un programa es un conjunto de definiciones de *funciones matemáticas*
  - ◇ El programador define funciones
  - ◇ El ordenador evalúa la expresión
- ✓ Ventajas:
  - ◇ Permite escribir programas claros, concisos y con alto nivel de abstracción
  - ◇ Soporta *Software* reusable
  - ◇ Facilita el uso de la verificación formal

Usaremos *Haskell 98*

`http://haskell.org`

1.2 El lenguaje Haskell

Haskell es un lenguaje funcional

- ✓ Puro:
  - ◇ Una misma expresión denota siempre el mismo valor (*Transparencia referencial*).
  - ◇ Verificación formal relativamente fácil.
- ✓ No estricto:
  - ◇ El orden utilizado para reducir expresiones es normal.
  - ◇ Las implementaciones de Haskell suelen usar *evaluación perezosa*.
  - ◇ Permite trabajar con estructuras infinitas.
- ✓ Fuertemente tipado:
  - ◇ Cada elemento tiene un *tipo*.
  - ◇ Se usa para comprobar el uso consistente de los elementos.
  - ◇ Usos inconsistentes dan lugar a errores de tipo.
  - ◇ Muchos errores se detectan pronto.

1.3 La notación Currificada

Aplicación de funciones

✓ En *Matemáticas* la aplicación de funciones es denotada usando paréntesis:

$f\ (a, b)\ +\ c\ \times\ d$ aplicar la función f a los argumentos a y b

✓ En *Haskell* la aplicación de funciones es denotada usando espacios (notación *currificada*):

$f\ a\ b\ +\ c * d$ aplicar la función f a los argumentos a y b

✓ En *Haskell* la aplicación de funciones tiene prioridad máxima:

$g\ a\ +\ b$ significa $(g\ \ a) + b$ y NO $g\ \ (a + b)$

✓ En *Haskell* los argumentos compuestos van entre paréntesis:

$f\ (a\ +\ b)\ c$ aplicar la función f a dos args: $(a + b)$ y c

Ejemplos:

Matemáticas	Haskell
$g\,(x)$	$g\ \ x$
$f\,(x, y)$	$f\ \ x\ \ y$
$g\,(f\,(x, y))$	$g\,(f\ \ x\ \ y)$
$f\,(x, g\,(y))$	$f\ \ x\ \ (g\ \ y)$
$g\,(x + y)$	$g\,(x + y)$
$g\,(x) + y$	$g\ \ x + y$

Definición de funciones

✓ Se usa también la notación *currificada*:

$$
\begin{array}{lll}
\text{-- Un comentario} & & \\
g & :: & Integer \rightarrow Integer \\
g\ x & = & x + 1 \\
 & & \\
f & :: & Integer \rightarrow Integer \rightarrow Integer \\
f\ x\ y & = & x + y + 2 \\
 & & \\
doble & :: & Integer \rightarrow Integer \\
doble\ x & = & x + x \\
 & & \\
cuadruple & :: & Integer \rightarrow Integer \\
cuadruple\ x & = & doble\ (doble\ x)
\end{array}
$$

Significado:

$$
\overbrace{f}^{nombre\ fun}\ \overbrace{::}^{tiene\ tipo}\ \overbrace{Integer}^{Tipo\ Arg\ 1^0} \rightarrow \overbrace{Integer}^{Tipo\ Arg\ 2^0} \rightarrow \overbrace{Integer}^{Tipo\ Res}
$$

$$
\underbrace{f}_{nombre\ fun}\ \underbrace{x}_{par.\ 1^0}\ \underbrace{y}_{par.\ 2^0}\ \underbrace{=}_{se\ define}\ \underbrace{x + y + 2}_{resultado}
$$

✓ Nombres de función: comienzan por minúscula

$f \qquad f' \qquad fun3 \qquad fun_3$

✓ Nombres de parámetros: comienzan por minúscula

$x \qquad y \qquad x' \qquad x1 \qquad xs$

✓ Nombres de tipos: comienzan por mayúscula

$Integer \qquad Bool \qquad Char$

1.4 Sesiones y declaraciones

El ordenador funciona como una calculadora o *evaluador*:

?

Valores numéricos enteros:

? $1 + 2$
$3 \;::\; \mathit{Integer}$

Valores reales:

? $\mathit{cos}\ \mathit{pi}$
$-1.0 \;::\; \mathit{Double}$

Solo los argumentos compuestos van entre paréntesis:

? $\mathit{cos}\ (2 * \mathit{pi})$
$1.0 \;::\; \mathit{Double}$

Ejemplo más elaborado:

? $[1..5]$
$[1, 2, 3, 4, 5] \;::\; [\mathit{Integer}]$

? $\mathit{sum}\ [1..10]$
$55 \;::\; \mathit{Integer}$

Sesiones y declaraciones (2)

Funciones de más de un argumento:

$$? \ mod \ 10 \ 3$$
$$1 \ :: \ Integer$$

$$? \ mod \ 10 \ (3+1)$$
$$2 \ :: \ Integer$$

✓ Haskell proporciona un rico conjunto de elementos predefinidos

✓ Este conjunto es extensible: el programador puede definir nuevas funciones, operadores y tipos de datos.

Ejemplo. función que calcula el sucesor de un número entero:

$$\begin{aligned} sucesor & :: && Integer \rightarrow Integer \\ sucesor \ x & = && x + 1 \end{aligned}$$

Tras proporcionar la declaración de función anterior al evaluador:

$$? \ sucesor \ 3$$
$$4 \ :: \ Integer$$

$$? \ 10 * sucesor \ 3$$
$$40 \ :: \ Integer$$

Una función de dos argumentos:

$$\begin{aligned} sumaCuadrados & :: && Integer \rightarrow Integer \rightarrow Integer \\ sumaCuadrados \ x \ y & = && x * x + y * y \end{aligned}$$

$$? \ sumaCuadrados \ 2 \ (sucesor \ 3)$$
$$20 \ :: \ Integer$$

$$? \ sumaCuadrados \ (2+2) \ 3$$
$$25 \ :: \ Integer$$

1.5 Reducción de expresiones

El evaluador calcula el resultado de una expresión utilizando las definiciones de las funciones involucradas.

Ejemplo

$$\begin{aligned} cuadrado &:: Integer \rightarrow Integer \\ cuadrado\ x &= x * x \end{aligned}$$

$$\begin{aligned} & 2 + \underline{cuadrado\ 3} \\ \Longrightarrow & \{\text{por la definición de } cuadrado\} \\ & 2 + (\underline{3 * 3}) \\ \Longrightarrow & \{\text{por el operador } (*)\} \\ & \underline{2 + 9} \\ \Longrightarrow & \{\text{por el operador } (+)\} \\ & 11 \end{aligned}$$

- ✓ Cada uno de los pasos efectuados es una *reducción*.
- ✓ En cada reducción, el evaluador busca una parte de la expresión que sea simplificable (*redex* o *reducto*) y la simplifica.
- ✓ Cuando una expresión no puede ser reducida más se dice que está en *forma normal*.
- ✓ Labor del ordenador: buscar un *redex* en la expresión, *reducirlo* y repetir este proceso hasta que la expresión esté en *forma normal*.

Reducción desde dentro hacia fuera

La definición del comportamiento del evaluador dada es *ambigua*.

¿Qué pasa cuando hay más de un *redex*?

Podemos reducir la expresión desde dentro hacia fuera (reducir primero aquellos reductos más anidados).

$$\begin{array}{lcl} cuadrado & :: & Integer \rightarrow Integer \\ cuadrado\ x & = & x\ *\ x \end{array}$$

$$\begin{array}{l} \quad cuadrado(\underline{cuadrado\ 3}) \\ \Longrightarrow \{\text{por la definición de } cuadrado\} \\ \quad cuadrado(\underline{3*3}) \\ \Longrightarrow \{\text{por el operador } (*)\} \\ \quad \underline{cuadrado\ 9} \\ \Longrightarrow \{\text{por la definición de } cuadrado\} \\ \quad \underline{9*9} \\ \Longrightarrow \{\text{por el operador } (*)\} \\ \quad 81 \end{array}$$

Esta estrategia presenta problemas.

Reducción desde fuera hacia dentro

Reducir la expresión desde fuera hacia dentro (reducir primero los reductos menos anidados).

La definición de la función $cuadrado$

$$
\begin{array}{lcl}
cuadrado & :: & Integer \to Integer \\
cuadrado\ x & = & x\ *\ x
\end{array}
$$

puede ser vista como una *regla de reescritura*:

$$cuadrado\ \boxed{x} \Longrightarrow \boxed{x} * \boxed{x}$$

Se pasan los argumentos a las funciones como expresiones sin reducir, no como valores.

$$
\begin{array}{l}
\quad \underline{cuadrado (cuadrado\ 3)} \\
\Longrightarrow \{\text{por la definición de } cuadrado\} \\
\quad \underline{(cuadrado\ 3)}\ *\ (cuadrado\ 3) \\
\Longrightarrow \{\text{por la definición de } cuadrado\} \\
\quad (\underline{3\ *\ 3})\ *\ (cuadrado\ 3) \\
\Longrightarrow \{\text{por la definición de } (*)\} \\
\quad 9\ *\ (\underline{cuadrado\ 3}) \\
\Longrightarrow \{\text{por la definición de } cuadrado\} \\
\quad 9\ *\ (\underline{3\ *\ 3}) \\
\Longrightarrow \{\text{por el operador } (*)\} \\
\quad \underline{9\ *\ 9} \\
\Longrightarrow \{\text{por el operador } (*)\} \\
\quad 81
\end{array}
$$

Observación: los operadores aritméticos son *estrictos*.

Importancia de la estrategia de reducción

Transparencia referencial: una misma expresión denota siempre el mismo valor.

Consecuencia:

- ✓ Sea cual sea la estrategia seguida en las reducciones, el resultado final (el valor 81) coincide (si se alcanza).
- ✓ La elección de un *redex* equivocado puede hacer que no se obtenga la forma normal de una expresión.

Ejemplo:

$$\begin{aligned} \mathit{infinito} &:: \mathit{Integer} \\ \mathit{infinito} &= 1 + \mathit{infinito} \end{aligned}$$

$$\begin{aligned} \mathit{cero} &:: \mathit{Integer} \to \mathit{Integer} \\ \mathit{cero}\ x &= 0 \end{aligned}$$

Comportamiento esperado $\forall\, n :: \mathit{Integer}\ .\ \mathit{cero}\ n \Longrightarrow 0$.

Si reducimos siempre el *redex* más interno:

$$\begin{aligned} & \mathit{cero}\ \underline{\mathit{infinito}} \\ \Longrightarrow\ & \{\text{por definición de } \mathit{infinito}\} \\ & \mathit{cero}\ (1 + \underline{\mathit{infinito}}) \\ \Longrightarrow\ & \{\text{por definición de } \mathit{infinito}\} \\ & \mathit{cero}\ (1 + (1 + \underline{\mathit{infinito}})) \\ \Longrightarrow\ & \{\text{por definición de } \mathit{infinito}\} \\ & \ldots \end{aligned}$$

Si reducimos el *redex* más externo:

$$\begin{aligned} & \underline{\mathit{cero}\ \mathit{infinito}} \\ \Longrightarrow\ & \{\text{por definición de } \mathit{cero}\} \\ & 0 \end{aligned}$$

La estrategia utilizada para seleccionar el *redex* es crucial, ya que puede hacer que se obtenga o no la forma normal de la expresión.

Orden de reducción aplicativo

- Seleccionar en cada reducción el *redex* más interno (el más anidado).

- En caso de que existan varios reductos que cumplan la condición anterior, se selecciona el que aparece más a la izquierda en la expresión.

Esto significa que

- Ante una aplicación de función, se reducen primero los argumentos de la función para obtener sus correspondientes valores (*paso de parámetros por valor*).

A los evaluadores que utilizan este orden se los llama *estrictos* o *impacientes*.

Problemas:

- A veces, se efectúan reducciones que no son necesarias:

$$
\begin{array}{l}
\quad cero\ (\underline{10 * 4}) \\
\Longrightarrow \{\text{por el operador } (*)\} \\
\quad \underline{cero\ 40} \\
\Longrightarrow \{\text{por definición de } cero\} \\
\quad 0
\end{array}
$$

- No encuentra la forma normal de ciertas expresiones:

$$
\begin{array}{l}
\quad cero\ \underline{infinito} \\
\Longrightarrow \{\text{por definición de } infinito\} \\
\quad cero\ (1 + \underline{infinito}) \\
\Longrightarrow \{\text{por definición de } infinito\} \\
\quad cero\ (1 + (1 + \underline{infinito})) \\
\Longrightarrow \{\text{por definición de } infinito\} \\
\quad \ldots
\end{array}
$$

Orden de reducción normal

- ✓ Seleccionar el *redex* más externo (menos anidado)
- ✓ En caso de conflicto, de entre los más externos el que aparece más a la izquierda de la expresión.

Esto significa que

- ✓ Se pasan como argumentos expresiones sin evaluar necesariamente (*paso de parámetros por nombre*)

A los evaluadores que utilizan este orden se los llama *no estrictos*.

Ventajas:

- ✓ Es *normalizante*: si la expresión tiene forma normal, una reducción mediante este orden la alcanza. (*Teorema de estandarización*).
- ✓ Un evaluador no estricto solo reducirá aquellos reductos que son necesarios para calcular el resultado final.

Problema:

- ✓ La reducción de los argumentos puede repetirse (menor eficiencia).

$\underline{cuadrado(cuadrado\ 3)}$

$\Longrightarrow$ {por la definición de $cuadrado$}

$\underline{(cuadrado\ 3)} * (cuadrado\ 3)$

$\Longrightarrow$ {por la definición de $cuadrado$}

$(\underline{3 * 3}) * (cuadrado\ 3)$

$\Longrightarrow$ {por la definición de $(*)$}

$9 * (\underline{cuadrado\ 3})$

$\Longrightarrow$ {por la definición de $cuadrado$}

$9 * (\underline{3 * 3})$

$\Longrightarrow$ {por el operador $(*)$}

$\underline{9 * 9}$

$\Longrightarrow$ {por el operador $(*)$}

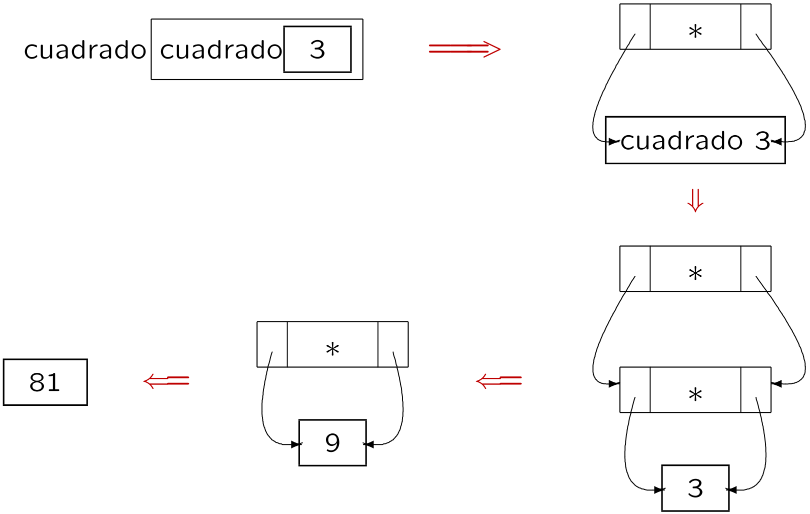
81

Evaluación Perezosa

La *Evaluación perezosa* soluciona este problema.

Evaluación perezosa = *paso por nombre* + recordar los valores de los argumentos ya calculados (evita que el cálculo se repita)

Cada expresión se representa mediante un *grafo*.



La reducción de la figura la escribiremos como:

$\underline{cuadrado\ (cuadrado\ 3)}$
$\Longrightarrow$ {por la definición de $cuadrado$}
$a * a$ **donde** $a = \underline{cuadrado\ 3}$
$\Longrightarrow$ {por la definición de $cuadrado$}
$a * a$ **donde** $a = \underline{b * b}$ **donde** $b = 3$
$\Longrightarrow$ {por el operador $(*)$}
$\underline{a * a}$ **donde** $a = 9$
$\Longrightarrow$ {por el operador $(*)$}
81

No se realizarán más reducciones que utilizando *paso por valor*.

Posee las ventajas del *paso por nombre* y no es menos eficiente que el *paso por valor*.

Objetivos del tema

El alumno debe:

- ✓ Conocer las bases del estilo de programación funcional
- ✓ Conocer las principales características de Haskell
- ✓ Conocer la notación currificada de Haskell
- ✓ Conocer los principales órdenes de reducción: aplicativo y normal
- ✓ Conocer las principales ventajas e inconvenientes de cada orden de reducción
- ✓ Saber reducir expresiones utilizando los distintos órdenes

Tema 2. Tipos predefinidos

2.1 Tipos en Haskell

2.2 Tipos simples predefinidos

El tipo Bool

El tipo Int

El tipo Integer

El tipo Float

El tipo Double

El tipo Char

Operadores de igualdad y orden

2.3 Constructores de tipo predefinidos

Tuplas

Listas

El constructor de tipo (→)

2.1 Tipos en Haskell

- Un *tipo* es una colección de valores relacionados.
  - $Integer$ es el conjunto de los enteros $\{ \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \}$

- La notación $e :: T$ indica que la expresión e tiene tipo T.
  - Por ejemplo $10 :: Integer$

- Cualquier expresión tiene un tipo.

- Antes de evaluar una expresión se comprueba que los tipos son consistentes (*chequeo de tipos*).

2.2 Tipos simples predefinidos

El tipo Bool

✓ Los valores de este tipo representan expresiones lógicas cuyo resultado puede ser verdadero o falso.

✓ Solo hay dos valores para el tipo: *True* y *False*.

Funciones y operadores

- $(\&\&) :: Bool \to Bool \to Bool$ conjunción lógica.
- $(||) :: Bool \to Bool \to Bool$ disyunción lógica.
- $not :: Bool \to Bool$ negación lógica.
- $otherwise :: Bool$ función constante que devuelve el valor *True*.

Comportamiento de las funciones anteriores:

$v1$	$v2$	$v1$ && $v2$	$v1 \,||\, v2$
True	*True*	*True*	*True*
True	*False*	*False*	*True*
False	*True*	*False*	*True*
False	*False*	*False*	*False*

v	$not\ v$
True	*False*
False	*True*

```
? True && False
False :: Bool

? not (True && False)
True :: Bool
```

El tipo Int

✓ Números enteros de rango limitado que cubren al menos el intervalo $[-2^{29}, 2^{29}-1]$.

Funciones y operadores

- $(+), (-), (*)$ :: $Int \rightarrow Int \rightarrow Int$. Suma, resta y producto de enteros.
- $(\wedge)$ :: $Int \rightarrow Int \rightarrow Int$. Operador potencia. El exponente debe ser mayor o igual a cero.
- div, mod :: $Int \rightarrow Int \rightarrow Int$. Cociente y resto de dividir dos enteros.
- abs :: $Int \rightarrow Int$. Valor absoluto.
- $signum$ :: $Int \rightarrow Int$. devuelve +1, -1 o 0, según el signo del entero argumento.
- $negate$ :: $Int \rightarrow Int$. Invierte el signo de su argumento. También puede usarse un signo menos prefijo.
- $even$, odd :: $Int \rightarrow Bool$. Comprueban la naturaleza par o impar de un número.

El tipo Integer

✓ Los valores de este tipo son números enteros de rango ilimitado.

✓ Para los valores del tipo *Integer* están disponibles las mismas operaciones que para el tipo *Int*.

✓ Los cálculos con datos de tipo *Integer* son menos eficientes que con datos de tipo *Int*.

```
? 2^100
1267650600228229401496703205376 :: Integer

? 111111111 * 111111111
12345678987654321 :: Integer
```

El tipo Float

- ✓ Subconjunto de un intervalo de los números reales.
- ✓ Hay dos modos de escribir valores reales:
  - ◇ La notación habitual: Por ejemplo, 1.35, -15.345, 1.0, 1
  - ◇ La notación científica: Por ejemplo, $1.5e7$, $1.5e-17$

Funciones y operadores

- $(+), (*), (-), (/)$ :: *Float* → *Float* → *Float*. Suma, producto, resta y división de reales
- $(\wedge)$ :: *Float* → *Int* → *Float*. Potencia, de base real, pero exponente entero y positivo.
- $(**)$ :: *Float* → *Float* → *Float*. Potencia, de base y exponente real.
- *abs* :: *Float* → *Float*. Valor absoluto.
- *signum* :: *Float* → *Float*. Devuelve -1.0, 0.0 ó +1.0 dependiendo del signo del real argumento.
- *negate* :: *Float* → *Float*. Devuelve el valor del real argumento negado. Puede usarse también el signo menos prefijo.

Funciones y operadores (2)

- sin, $asin$, cos, $acos$, tan, $atan$ $::$ $Float \rightarrow Float$. Funciones trigonométricas (trabajan con radianes)

- $atan2$ $::$ $Float \rightarrow Float \rightarrow Float$. $atan2\ x\ y$ devuelve la arcotangente de $\frac{x}{y}$.

- log, exp $::$ $Float \rightarrow Float$. Funciones logarítmicas y exponenciales.

- $sqrt$ $::$ $Float \rightarrow Float$. Raíz cuadrada.

- pi $::$ $Float$. El valor del número π.

- $truncate$, $round$, $floor$ y $ceiling$ $::$ $Float \rightarrow Integer$ o $Float \rightarrow Int$. Funciones de redondeo.

- $fromInt$ $::$ $Int \rightarrow Float$ y $fromInteger$ $::$ $Integer \rightarrow Float$. Funciones de conversión de tipo.

El tipo Double

✓ Se trata de un subconjunto de un intervalo de los números reales.

✓ El subconjunto es mayor que el correspondiente al tipo *Float* y las aproximaciones más precisas.

✓ Todas las operaciones disponibles para el tipo *Float* están también disponibles para el tipo *Double*.

El tipo Char

✓ Un valor de tipo *Char* representa un carácter (una letra, un dígito, un signo de puntuación, etc.).

✓ Un valor constante de tipo carácter se escribe entre comillas simples. 'a', '1', '?'

✓ Algunos caracteres especiales se escriben precediéndolos del carácter \:

- '\n' es el carácter de salto de línea.
- '\t' es el carácter tabulador.
- '\'' es el carácter comilla.
- '\"' es el carácter comilla doble.
- '\\' es el carácter \.

Funciones

- *ord* :: *Char* → *Int*. código ASCII del carácter argumento.
- *chr* :: *Int* → *Char*. Función inversa a la anterior.
- *isUpper*, *isLower*, *isDigit*, *isAlpha* :: *Char* → *Bool*. Comprueban si un carácter es una letra mayúscula, minúscula, un dígito o una letra.
- *toUpper*, *toLower* :: *Char* → *Char*. Convierten un carácter a mayúscula o minúscula.

Operadores de igualdad y orden

✓ Para todos los tipos básicos comentados están definidos los siguientes *operadores binarios*, que devuelven un valor booleano:

(>)	mayor que
(>=)	mayor o igual que
(<)	menor que
(<=)	menor o igual que
(==)	igual que
(/=)	distinto que

✓ El tipo de los dos argumentos debe ser el mismo (no se pueden comparar valores de tipos distintos).

Ejemplos

```
? 10 <= 15
True :: Bool

? 'x' == 'y'
False :: Bool

? 'x' /= 'y'
True :: Bool

? True < 'a'
ERROR                  : Type error in application
*** Expression         : True < 'a'
*** Term               : True
*** Type               : Bool
*** Does not match     : Char
```

✓ Para el tipo *Char* el orden viene dado por el código ASCII del carácter.

✓ Para el tipo *Bool*, el valor *False* se considera menor que *True*.

2.3 Constructores de tipo predefinidos

✓ Haskell define *tipos estructurados* que permiten representar colecciones de objetos.

Tuplas

✓ Una *tupla* es un dato compuesto donde el tipo de cada componente puede ser distinto.

> Tuplas
>
> Si $v_1, v_2, \ldots, v_n$ son valores con tipo $t_1, t_2, \ldots, t_n$
> entonces $(v_1, v_2, \ldots, v_n)$ es una tupla con tipo $(t_1, t_2, \ldots, t_n)$

Ejemplos:

```
? ()
() :: ()

? ('a', True)
('a', True) :: (Char, Bool)

? ('a', True, 1.5)
('a', True, 1.5) :: (Char, Bool, Double)
```

Las tuplas son útiles cuando una función tiene que devolver más de un valor.

```
predSuc   :: Integer → (Integer, Integer)
predSuc x =  (x − 1, x + 1)
```

Listas

✓ Una *lista* es una colección de cero o más elementos **todos del mismo tipo**.

Hay dos constructores para listas:

- [] Representa la lista vacía (lista con cero elementos).

- (:) Permite añadir un elemento a principio de una lista. Si xs es una lista con n elementos, y x es un elemento, entonces $x : xs$ es una lista con $n + 1$ elementos.

Listas
Si $v_1, v_2, \ldots, v_n$ son valores con tipo t entonces $v_1 : (v_2 : (\ldots (v_{n-1} : (v_n : []))))$ es una lista con tipo $[t]$

✓ El tipo de una lista no dice nada sobre su longitud

Ejemplos:

- $1 : []$ Una lista que almacena un único entero. Tiene tipo $[Integer]$.

- $3 : (1 : [])$ Una lista que almacena dos enteros. El valor 3 ocupa la primera posición dentro de la lista. El valor 1 la segunda.

- 'a' : (1 : []) Es una expresión errónea (produce un error de tipos).

Listas (2)

✓ El constructor (:) es asociativo a la derecha:

Asociatividad derecha de (:)

$$x_1 : x_2 : \ldots x_{n-1} : x_n : [\,] \quad \approx\!\approx\!\approx\!> \quad x_1 : (x_2 : (\ldots (x_{n-1} : (x_n : [\,]))))$$

Aún así, la notación sigue siendo engorrosa.

✓ Haskell permite una sintaxis para listas más cómoda:

Sintaxis para listas

$$[x_1, x_2, \ldots x_{n-1}, x_n] \quad \approx\!\approx\!\approx\!> \quad x_1 : (x_2 : (\ldots (x_{n-1} : (x_n : [\,]))))$$

Tres modos de escribir la misma lista:

```
? 1 : (2 : (3 : []))
[1,2,3] :: [Integer]

? 1 : 2 : 3 : []
[1,2,3] :: [Integer]

? [1,2,3]
[1,2,3] :: [Integer]
```

Cadenas de caracteres (Strings)

✓ Una *cadena de caracteres* es una secuencia de cero o más caracteres.

✓ En Haskell, las cadenas de caracteres son listas de caracteres.

✓ El tipo asociado a las cadenas de caracteres es *String* (un modo equivalente de escribir el tipo [*Char*]).

✓ Haskell permite una sintaxis más cómoda para escribir cadenas de caracteres: escribir el texto entre comillas dobles:

Cadenas de caracteres

$"x_1\, x_2\, \ldots\, x_{n-1}\, x_n" \quad \approx\approx\approx> \quad [\text{'x}_1\text{'}, \text{'x}_2\text{'}, \ldots \text{'x}_{n-1}\text{'}, \text{'x}_n\text{'}]$

Ejemplos:

```
? 'U' : 'n' : ' ' : 'C' : 'o' : 'c' : 'h' : 'e' : []
  "Un Coche" :: [Char]

? ['U', 'n', ' ', 'C', 'o', 'c', 'h', 'e']
  "Un Coche" :: [Char]

? "Un Coche"
  "Un Coche" :: String
```

El constructor de tipo $(\rightarrow)$

✓ Es posible declarar el tipo correspondiente a las distintas funciones. Para ello disponemos de un único constructor: $(\rightarrow)$.

Tipos Funcionales

Si $t_1, t_2, \ldots, t_n, t_r$ son tipos válidos
entonces $t_1 \rightarrow t_2 \rightarrow \ldots t_n \rightarrow t_r$ es el tipo
de una función con n argumentos

El tipo del resultado es t_r

Ejemplos:

$$\begin{array}{lcl} inc & :: & Integer \rightarrow Integer \\ inc\ x & = & x + 1 \end{array}$$

$$\begin{array}{lcl} esCero & :: & Integer \rightarrow Bool \\ esCero\ x & = & (x == 0) \end{array}$$

$$\begin{array}{lcl} sumaCuadrados & :: & Integer \rightarrow Integer \rightarrow Integer \\ sumaCuadrados\ x\ y & = & x \wedge 2 + y \wedge 2 \end{array}$$

Objetivos del tema

El alumno debe:

- ✓ Conocer los distintos tipos simples predefinidos
- ✓ Conocer las distintas funciones y operadores predefinidos para cada tipo
- ✓ Conocer los tipos estructurados predefinidos

Tema 3. Patrones y Definiciones de Funciones

3.1 Comparación de Patrones

Patrones constantes

Patrones para listas

Patrones para tuplas

Patrones aritméticos

Patrones nombrados o seudónimos

El patrón subrayado

Anidando patrones

Errores comunes

3.2 Expresiones condicionales

3.3 Funciones por casos

3.4 Expresiones case

3.5 La función error

3.6 Definiciones locales

3.7 Operadores

3.8 Sangrado

3.1 Comparación de Patrones

- ✓ Permiten modelar cómo se define una función para distintas formas del argumento.
- ✓ Es posible definir una función mediante varias ecuaciones.
  - ◇ Cada ecuación define la función para distintas formas del argumento (patrón).
  - ◇ Al utilizar la función, la comparación de patrones determina la ecuación adecuada.
- ✓ El orden de las ecuaciones es importante:
  - ◇ Se prueban las distintas ecuaciones en el orden dado por el programa.
  - ◇ Dentro de una misma ecuación, se intentan unificar los patrones correspondientes a los argumentos de izquierda a derecha.
  - ◇ En cuanto un patrón falla para un argumento, se pasa a la siguiente ecuación.
  - ◇ Se selecciona solamente la primera ecuación que unifique.
  - ◇ Si ninguna ecuación unifica se produce un error durante la reducción.

Patrones constantes

- ✓ Puede ser un número, un carácter o un constructor de dato
- ✓ Con un patrón constante solo *unifica* un argumento que coincida con dicha constante.

$$
\begin{array}{lcl}
f & :: & \mathit{Integer} \rightarrow \mathit{Bool} \\
f\ 1 & = & \mathit{True} \\
f\ 2 & = & \mathit{False}
\end{array}
$$

? $f\ 1$
$\mathit{True}\ ::\ \mathit{Bool}$

? $f\ 2$
$\mathit{False}\ ::\ \mathit{Bool}$

? $f\ 3$
$\mathit{Program\ error}:\ \{f\ 3\}$

Recursividad

✓ La función definida puede invocarse a sí misma para un argumento *más simple*

$$\begin{aligned} fact &:: Integer \rightarrow Integer \\ fact\ 0 &= 1 \\ fact\ n &= n * fact\ (n-1) \end{aligned}$$

Ejemplo: reducción de $fact\ 2$:

$\underline{fact\ 2}$

$\Longrightarrow$ {segunda ecuación de $fact$ $\langle\ n \leftarrow 2\ \rangle$}

$2 * fact\ \underline{(2-1)}$

$\Longrightarrow$ {definición de $(-)$}

$2 * \underline{fact\ 1}$

$\Longrightarrow$ {segunda ecuación de $fact$ $\langle\ n \leftarrow 1\ \rangle$}

$2 * (1 * fact\ \underline{(1-1)})$

$\Longrightarrow$ {definición de $(-)$}

$2 * (1 * \underline{fact\ 0})$

$\Longrightarrow$ {primera ecuación de $fact$}

$2 * (\underline{1 * 1})$

$\Longrightarrow$ {definición de $(*)$}

$\underline{2 * 1}$

$\Longrightarrow$ {definición de $(*)$}

2

✓ Recursividad termina si

- ◇ Existe caso base (no recursivo)
- ◇ Llamada recursiva se acerca al caso base

Patrones para listas

✓ Toman las siguientes formas:

- ◇ [] unifica con una lista vacía.
- ◇ $[x]$, $[x, y]$, etc. solo unifican con listas de uno, dos, etc. argumentos.
- ◇ $(x : xs)$ unifica con listas no vacías. x queda ligada a la *cabeza* y xs queda ligada a la *cola*. También se puede usar $(x : y : zs)$, $(x : y : v : zs)$, etc. para listas de al menos dos, tres, etc. elementos.

Ejemplo: suma lista de enteros:

$$
\begin{array}{lcll}
suma & :: & [Integer] \to Integer & \\
suma\ [\,] & = & 0 & \text{-- caso base} \\
suma\ (x : xs) & = & x + suma\ xs & \text{-- caso recursivo}
\end{array}
$$

$suma\ [1, 2, 3]$

$\leadsto$ {sintaxis de listas}

$\underline{suma\ (1 : (2 : (3 : [\,])))}$

$\Longrightarrow$ {segunda ecuación de $suma$ $\langle\ x \leftarrow 1,\ xs \leftarrow 2 : (3 : [\,])\ \rangle$}

$1 + \underline{suma\ (2 : (3 : [\,]))}$

$\Longrightarrow$ {segunda ecuación de $suma$ $\langle\ x \leftarrow 2,\ xs \leftarrow 3 : [\,]\ \rangle$}

$1 + (2 + \underline{suma\ (3 : [\,])})$

$\Longrightarrow$ {segunda ecuación de $suma$ $\langle\ x \leftarrow 3,\ xs \leftarrow [\,]\ \rangle$}

$1 + (2 + (3 + \underline{suma\ [\,]}))$

$\Longrightarrow$ {primera ecuación de $suma$}

$1 + (2 + (3 + 0))$

$\Longrightarrow$ {definición de (+) tres veces}

6

Patrones para tuplas

✓ Siguen la misma forma que las tuplas

Ejemplos: funciones que permiten seleccionar el primer elemento de tuplas de dos y tres componentes enteras:

$$
\begin{array}{lcl}
primero2 & :: & (Integer,\ Integer) \rightarrow Integer \\
primero2\ (x,\ y) & = & x
\end{array}
$$

$$
\begin{array}{lcl}
primero3 & :: & (Integer,\ Integer,\ Integer) \rightarrow Integer \\
primero3\ (x,\ y,\ z) & = & x
\end{array}
$$

? $primero2\ (5, 8)$
$5\ ::\ Integer$

? $primero3\ (5, 8, 7)$
$5\ ::\ Integer$

? $primero2\ (5, 8, 7)$

$$
\begin{array}{lcl}
ERROR & : & Type\ error\ in\ application \\
***\ Expression & : & primero2\ (5, 8, 7) \\
***\ \ Term & : & (5, 8, 7) \\
***\ \ Type & : & (a, b, c) \\
***\ \ Does\ not\ match & : & (Int, Int)
\end{array}
$$

Patrones aritméticos

✓ Para argumentos enteros

✓ Tienen la forma $(n + k)$, donde k es una constante natural

✓ Solo unifica con un número entero mayor o igual a k

✓ La variable n toma el valor del argumento unificado menos k.

Ejemplo: la función factorial:

$$
\begin{array}{lll}
\mathit{factorial} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{factorial}\ 0 & = & 1 \\
\mathit{factorial}\ (n+1) & = & (n+1)\ *\ \mathit{factorial}\ n
\end{array}
$$

✓ No unifica con argumentos negativos

La reducción de $\mathit{factorial}\ 2$ es:

$$
\begin{array}{l}
\quad \underline{\mathit{factorial}\ 2} \\
\Longrightarrow \{\text{segunda ecuación de } \mathit{factorial} \ \ \langle\ n \leftarrow 1\ \rangle\} \\
\quad \underline{(1+1)}\ *\ \mathit{factorial}\ 1 \\
\Longrightarrow \{\text{definición de } (+)\} \\
\quad 2\ *\ \underline{\mathit{factorial}\ 1} \\
\Longrightarrow \{\text{segunda ecuación de } \mathit{factorial} \ \ \langle\ n \leftarrow 0\rangle\} \\
\quad 2\ *\ (\underline{(0+1)}\ *\ \mathit{factorial}\ 0) \\
\Longrightarrow \{\text{definición de } (+)\} \\
\quad 2\ *\ (1\ *\ \underline{\mathit{factorial}\ 0}) \\
\Longrightarrow \{\text{primera ecuación de } \mathit{factorial}\} \\
\quad 2\ *\ (1\ *\ 1) \\
\Longrightarrow \{\text{definición de } (*) \text{ dos veces}\} \\
\quad 2
\end{array}
$$

Patrones nombrados o seudónimos

✓ Permiten nombrar un patrón compuesto y utilizar el nombre en vez del patrón

✓ Toman la forma $nombre$ @ $patrón$

$$
\begin{array}{lcl}
factorial' & :: & Integer \rightarrow Integer \\
factorial'\ 0 & = & 1 \\
factorial'\ m@(n+1) & = & m * factorial'\ n
\end{array}
$$

- El cualificador @ en la segunda ecuación asigna el nombre m al patrón $(n+1)$.
- m queda asociada con el valor del argumento
- n queda asociada con el valor del argumento menos uno.

Por ejemplo, la reducción de $factorial''\ 2$ es:

$\underline{factorial'\ 2}$
$\Longrightarrow$ {segunda ecuación de $factorial'$ $\langle\, n \leftarrow 1,\ m \leftarrow 2 \,\rangle$}
$2 * \underline{factorial'\ 1}$
$\Longrightarrow$ {segunda ecuación de $factorial'$ $\langle\, n \leftarrow 0,\ m \leftarrow 1 \,\rangle$}
$2 * (1 * \underline{factorial'\ 0})$
$\Longrightarrow$ {primera ecuación de $factorial'$}
$2 * (1 * 1)$
$\Longrightarrow$ {definición de $(*)$ dos veces}
2

✓ Los patrones nombrados pueden mejorar ligeramente la eficiencia de una función.

El patrón subrayado

- ✓ Toman la forma $_$
- ✓ Unifican con cualquier argumento
- ✓ No producen ninguna ligadura

Pueden utilizarse cuando el argumento no es usado en el cuerpo de la función.

Ejemplo: número de elementos de una lista de enteros

$$
\begin{array}{lcl}
longitud & :: & [Integer] \rightarrow Integer \\
longitud\ [\,] & = & 0 \\
longitud\ (x : xs) & = & 1 + longitud\ xs
\end{array}
$$

Puede ser escrita usando un patrón subrayado como

$$
\begin{array}{lcl}
longitud & :: & [Integer] \rightarrow Integer \\
longitud\ [\,] & = & 0 \\
longitud\ (_ : xs) & = & 1 + longitud\ xs
\end{array}
$$

Anidando patrones

✓ Se pueden anidar patrones

Ejemplo: función que suma todos los elementos de una lista de pares:

$$\begin{array}{lcl} sumaPares & :: & [(Integer,\ Integer)] \rightarrow Integer \\ sumaPares\ [\,] & = & 0 \\ sumaPares\ ((x,y):xs) & = & x\ +\ y\ +\ sumaPares\ xs \end{array}$$

Por ejemplo

$sumaPares\ [(1,2),(3,4)]$

$\approx\!\approx\!\approx\!>$ {sintaxis de listas}

$\underline{sumaPares\ ((1,2):((3,4):[\,]))}$

$\Longrightarrow$ {segunda ecuación de $sumaPares$ $\langle\ x \leftarrow 1,\ y \leftarrow 2,\ xs \leftarrow (3,4):[\,]\ \rangle$}

$1\ +\ 2\ +\ \underline{sumaPares\ ((3,4):[\,])}$

$\Longrightarrow$ {segunda ecuación de $sumaPares$ $\langle\ x \leftarrow 3,\ y \leftarrow 4,\ xs \leftarrow [\,]\ \rangle$}

$1\ +\ 2\ +\ (3\ +\ 4\ +\ \underline{sumaPares\ [\,]})$

$\Longrightarrow$ {primera ecuación de $sumaPares$}

$1\ +\ 2\ +\ (3\ +\ 4\ +\ 0)$

$\Longrightarrow$ {definición de $(+)$ cuatro veces}

10

Errores comunes

✓ Un nombre de variable no puede aparecer repetido en la parte izquierda (a la izquierda del signo igual) de una misma ecuación:

$$\begin{array}{lcl} sonIguales & :: & Integer \rightarrow Integer \rightarrow Bool \\ sonIguales\ x\ x & = & True \qquad \text{-- INCORRECTO: x REPETIDA !!!} \\ sonIguales\ x\ y & = & False \end{array}$$

En su lugar, hay que escribir:

$$\begin{array}{lcl} sonIguales & :: & Integer \rightarrow Integer \rightarrow Bool \\ sonIguales\ x\ y & = & (x == y) \end{array}$$

✓ No es un error repetir el patrón subrayado:

$$\begin{array}{lcl} siempreVerdad & :: & Integer \rightarrow Integer \rightarrow Bool \\ siempreVerdad\ _\ \ _ & = & True \end{array}$$

✓ El tipo de todas las ecuaciones correspondientes a la definición de una función debe ser el mismo:

$$\begin{array}{lcl} f\ 0 & = & 0 \qquad \text{-- INCORRECTO: } Bool \text{ e } Integer \text{ son tipos distintos !!!} \\ f\ True & = & 2 \end{array}$$

3.2 Expresiones condicionales

✓ Expresiones cuyo resultado depende de una condición.

if exprBool **then** exprSi **else** exprNo

✓ El tipo de *exprBool* debe ser *Bool*.

✓ Los tipos de *exprSi* y *exprNo* deben ser iguales.

✓ La parte **else** es obligatoria

Comportamiento:

1. Se evalúa el valor de *exprBool*.
2. Si el valor es *True*, el valor de la expresión es el de *exprSi*.
3. En otro caso, el valor de la expresión es el de *exprNo*.

Ejemplo. Máximo de dos enteros:

```
máximo     :: Integer → Integer → Integer
máximo x y  = if x >= y then x else y
```

✓ La evaluación es perezosa:

```
? if 5 > 2 then 10.0 else (10.0/0.0)
10.0 :: Double

? if 5 < 2 then 10.0 else (10.0/0.0)
Program error : {primDivFloat 10.0 0.0}

? 2 * if 'a' < 'z' then 10 else 4
20 :: Integer
```

3.3 Funciones por casos

- ✓ Generalización de las expresiones condicionales

$$
\begin{array}{lll}
\mathit{absoluto} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{absoluto}\ x & & \\
\quad \mid\ x >= 0 & = & x \\
\quad \mid\ x < 0 & = & -x
\end{array}
$$

- ✓ Las expresiones entre los símbolos | y = se denominan *guardas* (tipo Bool)

- ✓ Se devuelve el resultado correspondiente a la **primera** guarda cierta

Ejemplo: devuelve -1, 0 ó 1 dependiendo del signo del argumento:

$$
\begin{array}{lll}
\mathit{signo} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{signo}\ x & & \\
\quad \mid\ x > 0 & = & 1 \\
\quad \mid\ x == 0 & = & 0 \\
\quad \mid\ x < 0 & = & -1
\end{array}
$$

- ✓ $\mathit{otherwise}$ es equivalente al valor True. Suele aparecer como última guarda:

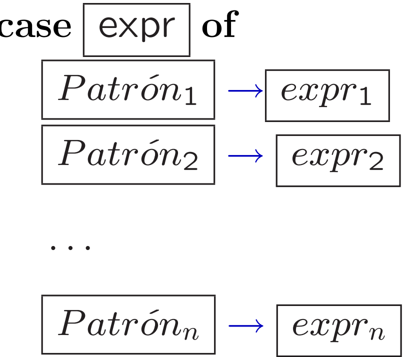
$$
\begin{array}{lll}
\mathit{signo} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{signo}\ x & & \\
\quad \mid\ x > 0 & = & 1 \\
\quad \mid\ x == 0 & = & 0 \\
\quad \mid\ \mathit{otherwise} & = & -1
\end{array}
$$

- ✓ Cuidado con el sangrado

3.4 Expresiones case

✓ Permiten calcular un resultado que depende de la forma de una expresión

Sintaxis:



Comportamiento:

1. Se evalúa $expr$.
2. Se devuelve la primera $expr_i$ tal que $Patrón_i$ unifica con el resultado de evaluar $expr$
3. Si ningún $Patrón_i$ unifica se produce un error

✓ $expr$ y todos los $Patrón_i$ han de tener el mismo tipo.

✓ Todas las $expr_i$ han de tener el mismo tipo.

Ejemplo:

$$
\begin{array}{lcl}
suma & :: & [Integer] \rightarrow Integer \\
suma\ ls & = & \textbf{case}\ ls\ \textbf{of} \\
 & & \quad [\,] \qquad\quad \rightarrow 0 \\
 & & \quad (x : xs) \rightarrow x + suma\ xs
\end{array}
$$

3.5 La función error

✓ Abortan la evaluación de una expresión y muestran un mensaje por pantalla.

Ejemplo:

$$\begin{array}{lcl} cabeza & :: & [Integer] \rightarrow Integer \\ cabeza\ [\,] & = & error\ \text{"cabeza de lista vacía no definida"} \\ cabeza\ (x : _\ ) & = & x \end{array}$$

✓ El intérprete mostrará el mensaje cuando se aplique $cabeza$ a una lista vacía:

? $cabeza\ [1, 2, 3]$
$1\ ::\ Integer$

? $cabeza\ [\,]$
$Program\ error:\ cabeza\ de\ lista\ vacía\ no\ definida$

✓ $error$ permite controlar casos para los que la definición de la función no tiene sentido y emitir un mensaje por pantalla.

3.6 Definiciones locales

✓ Definiciones con visibilidad limitada

✓ Suelen usarse para nombrar una expresión que aparece varias veces en una función

Ejemplo:

```
raíces :: Float → Float → Float → (Float, Float)
raíces a b c
  | disc >= 0  = ((−b + raízDisc)/denom, (−b − raízDisc)/denom)
  | otherwise  = error "raíces no reales"
  where
    disc     = b^2 − 4 ∗ a ∗ c
    raízDisc = sqrt disc
    denom    = 2 ∗ a
```

disc, *raízDisc* y *denom* son *definiciones locales* de la función *raíces*.

✓ Las definiciones locales **where** solo pueden aparecer al final de una declaración de función.

✓ Es posible introducir definiciones locales en cualquier parte de una expresión usando las palabras **let** e **in**:

```
? let f n = n^2 + 2 in f 100
10002 :: Integer
```

3.7 Operadores

✓ Funciones de dos argumentos con nombre simbólico

✓ Suelen usarse de modo infijo

```
? 1 + 2
3 :: Integer
```

✓ Pueden usarse de modo prefijo (entre paréntesis)

```
? (+) 1 2
3 :: Integer
```

✓ Una función de dos argumentos se puede usar infija (entre acentos franceses):

```
? 10 `div` 3
3 :: Integer
```

✓ Para definir operadores se pueden utilizar uno o más de:

```
: ! # $ % & * + . / < = > ? @ \ ^ | - ~
```

✓ Si comienzan por dos puntos (:) son *constructores de datos infijos*.

Algunos ejemplos:

```
+    ++    &&    ||    <=    ==    /=    .    //    $
::   ->    =>    :     ..    =     @     \    |     <-    ~
%    @@    -*-         /     <+>   ?
```

✓ Los de la primera línea están predefinidos

✓ Los de la segunda están reservados

Operadores (2)

✓ Prioridad:

◇ Entre 0 y 9

◇ Valor mayor significa mayor prioridad

✓ Asociatividad

◇ Izquierda(**infixl**), derecha (**infixr**) o ninguna (**infix**)

✓ Tabla Prelude

infixr	9	.
infixl	9	!!
infixr	8	^, ^^, **
infixl	7	*, /, ‘*quot*‘, ‘*rem*‘, ‘*div*‘, ‘*mod*‘
infixl	6	+, −
infixr	5	:
infixr	5	++
infix	4	==, /=, <, <=, >=, >, ‘*elem*‘, ‘*notElem*‘
infixr	3	&&
infixr	2	||
infixl	1	>>, >>=
infixr	1	=<<
infixr	0	$, $!, ‘*seq*‘

✓ Ejemplo

```
-- O exclusivo
infixr 2 |||

(|||)          :: Bool -> Bool -> Bool
True  ||| True  = False
False ||| False = False
_     ||| _     = True
```

3.8 Sangrado

✓ La regla del *sangrado* se aplica tras las palabras **let**, **where** , **doy** **of**, además de a las definiciones globales y guardas.

✓ Consecuencia de esta regla:

- todas las definiciones globales deben tener el mismo sangrado (se recomienda que comiencen en la primera columna).
- todas las definiciones locales introducidas por **where** o **let** deben tener el mismo sangrado.

Ejemplo de sangrado **INCORRECTO**:

```
f1   :: Integer -> Integer
f1 x  =  z + y
   where
      z = 3
         y = 4
```

✓ Una sintaxis alternativa es utilizar llaves y el separador punto y coma:

```
g     :: Integer -> Integer -> Integer
g x y  = doble x + triple y where {doble n = 2 * n; triple n = 3 * n}

h     :: Integer -> Integer -> Integer
h x y  = let{doble n = 2 * n; triple n = 3 * n} in doble x + triple y
```

Objetivos del tema

El alumno debe:

- ✓ Conocer el concepto de recursividad
- ✓ Conocer los distintos tipos de patrones
- ✓ Conocer como unifican dichos patrones
- ✓ Saber definir funciones utilizando patrones
- ✓ Saber reducir expresiones en las que sea necesario unificar patrones
- ✓ Conocer las expresiones condicionales, las funciones por casos, las expresiones case y la función error
- ✓ Saber definir funciones utilizando las construcciones anteriores
- ✓ Saber utilizar definiciones locales siempre que sea adecuado:
  - ◇ La definición solo sea usada desde una función
  - ◇ Una expresión constante aparezca varias veces en una función
- ✓ Saber sangrar adecuadamente el código que escriba

Tema 4. Funciones de orden superior

4.1 Funciones de orden superior

4.2 Expresiones lambda

4.3 Aplicación parcial

Secciones

4.4 Ejemplo: una función de orden superior para enteros

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*).
- ✓ Son más útiles que las funciones normales (parte del comportamiento se especifica al usarlas).

Ejemplo:

$$\begin{array}{lll} dosVeces & :: & (Integer \to Integer) \to Integer \to Integer \\ dosVeces\ f\ x & = & f\ (f\ x) \end{array}$$

$$\begin{array}{lll} inc & :: & Integer \to Integer \\ inc\ x & = & x + 1 \end{array}$$

$$\begin{array}{lll} dec & :: & Integer \to Integer \\ dec\ x & = & x - 1 \end{array}$$

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

? $\lambda\ x \rightarrow x + 1$
$\ll function \gg$ :: $Integer \rightarrow Integer$

? $(\lambda\ x \rightarrow x + 1)$ 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:

? $(\lambda\ x\ y \rightarrow x + y)$
$\ll function \gg$ :: $Integer \rightarrow Integer \rightarrow Integer$

? $(\lambda\ x\ y \rightarrow x + y)$ 5 7
12 :: $Integer$

✓ Son útiles como argumentos de funciones de orden superior:

? $dosVeces\ (\lambda\ x \rightarrow x + 1)$ 10
12 :: $Integer$

? $dosVeces\ (\lambda\ x \rightarrow x - 1)$ 10
8 :: $Integer$

? $dosVeces\ (\lambda\ x \rightarrow x * 2)$ 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 $k \le n$ argumentos con los tipos adecuados, se obtiene como resultado una nueva función que espera los $n - k$ argumentos restantes.

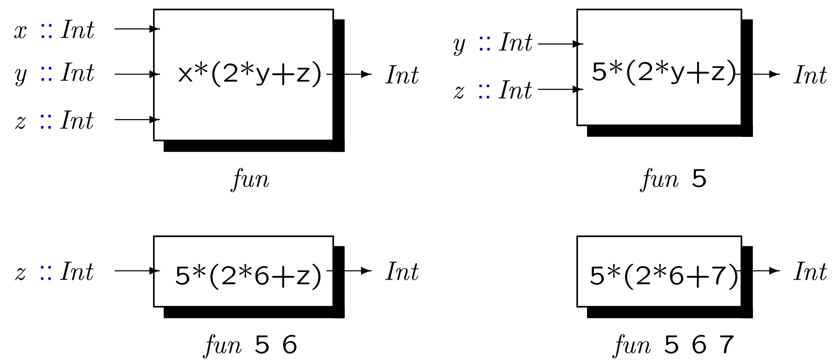
Regla de la cancelación

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 $(k \le n)$
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

Asociatividad a la derecha de (→) (En Tipos)

$$t_1 \rightarrow t_2 \rightarrow \ldots t_n \quad \approx\approx> \quad (t_1 \rightarrow (t_2 \rightarrow (\ldots \rightarrow t_n)))$$

Asociatividad izquierda de la aplicación de funciones

$$f a_1 a_2 \ldots a_n \quad \approx\approx> \quad (((f\ a_1)\ a_2) \ldots\ a_n)$$

Consideremos la función anterior:

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

Por las reglas anteriores la definición es equivalente a:

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

✓ La expresión fun 5 tiene tipo $Int \rightarrow (Int \rightarrow Int)$

✓ La expresión fun 5 6 es equivalente a (fun 5) 6 y tiene tipo $Int \rightarrow Int$

✓ La expresión fun 5 6 7 es equivalente a ((fun 5) 6) 7 y tiene tipo Int

Secciones

✓ Los operadores pueden ser aplicados parcialmente

✓ Se obtienen funciones de un argumento

Si (★) es un operador tenemos las siguientes equivalencias (los paréntesis son obligatorios):

Secciones de operadores

$(x\ \star)$	≈≈≈>	$\lambda\ y \rightarrow x \star y$
$(\star\ y)$	≈≈≈>	$\lambda\ x \rightarrow x \star y$
$(\star)$	≈≈≈>	$\lambda\ x\ y \rightarrow x \star y$

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:

$$
\begin{array}{lll}
\mathit{factorial} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{factorial}\ 0 & = & 1 \\
\mathit{factorial}\ m@(n+1) & = & (*)\ m\ (\mathit{factorial}\ n)
\end{array}
$$

$$
\begin{array}{lll}
\mathit{sumatorio} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{sumatorio}\ 0 & = & 0 \\
\mathit{sumatorio}\ m@(n+1) & = & (+)\ m\ (\mathit{sumatorio}\ n)
\end{array}
$$

Ambas siguen el esquema:

$$
\begin{array}{lll}
\mathit{fun} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{fun}\ 0 & = & \boxed{e} \\
\mathit{fun}\ m@(n+1) & = & \boxed{f}\ m\ (\mathit{fun}\ n)
\end{array}
$$

Función de orden superior que lo captura:

$$
\begin{array}{lll}
\mathit{iter} & :: & (\mathit{Integer} \rightarrow \mathit{Integer} \rightarrow \mathit{Integer}) \rightarrow \mathit{Integer} \rightarrow \\
 & & (\mathit{Integer} \rightarrow \mathit{Integer}) \\
\mathit{iter}\ f\ e & = & \mathit{fun} \\
\quad \textbf{where} & & \\
\qquad \mathit{fun} & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\qquad \mathit{fun}\ 0 & = & e \\
\qquad \mathit{fun}\ m@(n+1) & = & f\ m\ (\mathit{fun}\ n)
\end{array}
$$

Ahora es posible una definición más compacta:

$$
\begin{array}{lll}
\mathit{factorial}' & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{factorial}' & = & \mathit{iter}\ (*)\ 1
\end{array}
$$

$$
\begin{array}{lll}
\mathit{sumatorio}' & :: & \mathit{Integer} \rightarrow \mathit{Integer} \\
\mathit{sumatorio}' & = & \mathit{iter}\ (+)\ 0
\end{array}
$$

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*)

Tema 5. Polimorfismo

5.1 Funciones Polimórficas

La función identidad

Polimorfismo en Tuplas

Polimorfismo en Listas

Composición de funciones

El operador ($)

5.1 Funciones Polimórficas

- ✓ Tienen sentido independientemente del tipo
- ✓ Ventaja: código más reutilizable y fácil de mantener

La función identidad

Ejemplo simple: La función predefinida *identidad*

$$
\begin{array}{lcl}
id & :: & a \rightarrow a \\
id\ x & = & x
\end{array}
$$

Uso:

```
? id 'd'
'd' :: Char

? id 120
120 :: Integer

? id [1,2,0]
[1,2,0] :: [Integer]
```

- ✓ La a en el tipo es una *variable de tipo* (en minúscula): denota un tipo arbitrario
- ✓ El tipo del argumento, a, indica que id puede tomar argumentos de cualquier tipo
- ✓ El tipo del resultado, a, indica que id devuelve un valor cuyo tipo coincide con el del argumento

Polimorfismo en Tuplas

Los selectores predefinidos *fst* y *snd* permiten extraer componentes:

$$fst :: (a, b) \rightarrow a$$
$$fst\ (x\ ,\ _\ ) = x$$

$$snd :: (a, b) \rightarrow b$$
$$snd\ (\ _\ ,\ y) = y$$

Uso:

```
? fst (1 , 'd')
1 :: Integer

? snd (1 , 'd')
'd' :: Char

? snd (1 , 2)
2 :: Integer
```

- ✓ Se usan dos variables de tipo distintas: a y b
- ✓ Esto indica que los tipos de ambas componentes pueden ser distintos
- ✓ El resultado de *fst* tiene siempre el mismo tipo que la primera componente del argumento
- ✓ Dos variables de tipo distintas pueden corresponder a dos tipos distintos, aunque no es obligatorio (p. ej. $snd(1, 2)$)

Polimorfismo en Listas

La función predefinida *length* calcula la longitud de listas de cualquier tipo:

$$
\begin{array}{lll}
length & :: & [a] \rightarrow Int \\
length\ [\,] & = & 0 \\
length\ (_ : xs) & = & 1 + length\ xs
\end{array}
$$

Uso:

$$
\begin{array}{l}
?\ length\ [10, 11, 12] \\
3\ ::\ Int \\
\\
?\ length\ [True, False] \\
2\ ::\ Int \\
\\
?\ length\ [\,[10, 11, 12],\ [13, 14, 15, 16]\,] \\
2\ ::\ Int
\end{array}
$$

Los selectores predefinidos pueden ser utilizados con listas de cualquier tipo:

$$
\begin{array}{llll}
head & :: & [a] \rightarrow a & \text{-- Cabeza de la lista} \\
head\ (x : _) & = & x & \\
tail & :: & [a] \rightarrow [a] & \text{-- Cola de la lista} \\
tail\ (_ : xs) & = & xs & \\
last & :: & [a] \rightarrow a & \text{-- Último elmento de la lista} \\
last\ [x] & = & x & \\
last\ (_ : xs) & = & last\ xs & \\
init & :: & [a] \rightarrow [a] & \text{-- Toda menos el último elemento} \\
init\ [x] & = & [\,] & \\
init\ (x : xs) & = & x : init\ xs &
\end{array}
$$

Polimorfismo en Listas (2)

Concatenación de listas:

$$\begin{array}{lll}
\textbf{infixr}\ 5\ ++ & & \\
(++) & :: & [a] \to [a] \to [a] \\
[\,]\ \ ++\ ys & = & ys \\
(x : xs)\ ++\ ys & = & x\ :\ (xs\ ++\ ys)
\end{array}$$

La función map:

$$\begin{array}{lll}
map & :: & (a \to b) \to [a] \to [b] \\
map\ f\ [\,] & = & [\,] \\
map\ f\ (x : xs) & = & f\ x\ \ :\ \ map\ f\ xs
\end{array}$$

La función $filter$:

$$\begin{array}{lll}
filter & :: & (a \to Bool) \to [a] \to [a] \\
filter\ p\ [\,] & = & [\,] \\
filter\ p\ (x : xs) & & \\
\quad |\ p\ x & = & x\ :\ filter\ p\ xs \\
\quad |\ otherwise & = & filter\ p\ xs
\end{array}$$

Usos:

? $[1,3,5]\ ++\ [2,4]$
$[1,3,5,2,4]\ ::\ [Integer]$

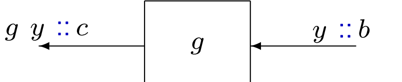
? $map\ (+1)\ [1,2,3]$
$[2,3,4]\ ::\ [Integer]$

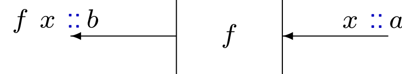
? $map\ even\ [1,2,3,4]$
$[False,\ True,\ False,\ True]\ ::\ [Bool]$

? $filter\ even\ [1,2,3,4]$
$[2,4]\ ::\ [Integer]$

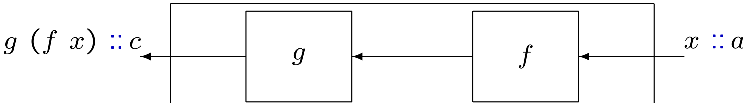
Composición de funciones

Si $f \ :: \ a \rightarrow b$ y $g \ :: \ b \rightarrow c$





se define $g \, . \, f$:



✓ El resultado de la composición es otra función con tipo $g.f \ :: \ a \rightarrow c$

✓ Si el tipo del resultado de f no coincide con el del argumento de g las funciones no se pueden componer.

✓ En Haskell, (.) está predefinido como

```
infixr 9 .
(.)   ::  (b → c) → (a → b) → (a → c)
g . f  =  λ x → g (f x)
```

Ejemplos:

```
esPar    ::  Integer → Bool
esPar x  =   (x ‘mod‘ 2 == 0)

esImpar  ::  Integer → Bool     -- Recordemos que not :: Bool → Bool
esImpar  =   not . esPar

fun  ::  Integer → Integer
fun  =   (+1) . (*2) . (+2)

  ? esImpar 5
  True :: Bool

  ? fun 10
  25 :: Integer
```

El operador ($)

Operador polimórfico predefinido, que permite aplicar una función a su argumento:

$$\textbf{infixr}\ 0\ \$$$
$$(\$) \quad :: \quad (a \rightarrow b) \rightarrow a \rightarrow b$$
$$f\ \$\ x = f\ x$$

Uso:

$$\textbf{?}\ f\ 5\ \textbf{where}\ f\ x = 2 * x$$
$$10\ ::\ \mathit{Integer}$$

$$\textbf{?}\ f\ \$\ 5\ \textbf{where}\ f\ x = 2 * x$$
$$10\ ::\ \mathit{Integer}$$

Su baja prioridad (mínima) lo hace útil para evitar paréntesis:

$$\textbf{?}\ f\ 5 + 3\ \textbf{where}\ f\ x = 2 * x$$
$$13\ ::\ \mathit{Integer}$$

$$\textbf{?}\ f\ (5 + 3)\ \textbf{where}\ f\ x = 2 * x$$
$$16\ ::\ \mathit{Integer}$$

$$\textbf{?}\ f\ \ \$\ \ 5 + 3\ \ \textbf{where}\ f\ x = 2 * x$$
$$16\ ::\ \mathit{Integer}$$

$$\textbf{?}\ (+1)\ .\ (*2)\ .\ (+2)\ \ \$\ \ 10$$
$$25\ ::\ \mathit{Integer}$$

Objetivos del tema

El alumno debe:

- ✓ Comprender las definiciones de funciones y tipos polimórficos
- ✓ Saber definir y utilizar funciones polimórficas
- ✓ Conocer algunos de los operadores y funciones polimóficas predefinidas
- ✓ Saber utilizar el operador de composición de funciones para definir nuevas funciones a partir de otras.
- ✓ Saber el tipo de las funciones que se obtienen por composición de otras.

Tema 6. Definiciones de tipos

6.1 Sinónimos de tipo

6.2 Definiciones de tipos de datos

Tipos enumerados

Uniones

Productos

Registros variantes

Tipos recursivos

6.3 Tipos Polimórficos

Either

Maybe

Listas

6.1 Sinónimos de tipo

✓ Introducen un nuevo nombre para un tipo existente.

✓ Se usa la palabra clave **type** :

Ejemplos:

$$\textbf{type}\ \underbrace{Entero}_{Nuevo\ \ Nombre} = \underbrace{Integer}_{Tipo\ \ Existente}$$

$$\begin{array}{lll} uno & :: & Entero \\ uno & = & 1 \end{array}$$

$$\textbf{type}\ DeEnteroEnEntero = Entero \rightarrow Entero$$

$$\begin{array}{lll} sucesor & :: & DeEnteroEnEntero \\ sucesor\ x & = & x + 1 \end{array}$$

$$\textbf{type}\ ParFlotantes = (Float, Float)$$

$$\begin{array}{lll} parCeros & :: & ParFlotantes \\ parCeros & = & (0.0,\ 0.0) \end{array}$$

$\textbf{type}\ String = [Char]$ −− Predefinido en Prelude

✓ El nuevo nombre del tipo debe comenzar con mayúscula.

6.2 Definiciones de tipos de datos

✓ El programador puede definir nuevos tipos (palabra reservada **data**)

✓ Todos los nombres de tipo deben comenzar con mayúscula

Tipos enumerados

✓ Constan de un número finito de valores que se enumeran en la definición.

Ejemplo:

$$\begin{aligned} \textbf{data}\ \textit{DíaSemana} &= \textit{Lunes} \mid \textit{Martes} \mid \textit{Miércoles} \mid \textit{Jueves} \mid \textit{Viernes} \\ &\mid \textit{Sábado} \mid \textit{Domingo}\ \ \textbf{deriving}\ \textit{Show} \end{aligned}$$

$$\begin{aligned} \textit{unDía} &:: \textit{DíaSemana} \\ \textit{unDía} &= \textit{Lunes} \end{aligned}$$

$$\begin{aligned} \textit{laborables} &:: [\textit{DíaSemana}] \\ \textit{laborables} &= [\textit{Lunes}, \textit{Martes}, \textit{Miércoles}, \textit{Jueves}, \textit{Viernes}] \end{aligned}$$

✓ *DíaSemana* es un *constructor de tipo* (nombre del tipo definido).

✓ Los valores que puede tomar una variable del tipo *DíaSemana* son *Lunes*, *Martes*, . . . o *Domingo*.

✓ Son los *constructores de datos*. También deben empezar con mayúscula.

✓ La cláusula **deriving** *Show* es necesaria para poder mostrar por pantalla los valores del tipo.

✓ Un mismo constructor de datos no puede aparecer en dos tipos distintos.

Tipos enumerados (2)

Los constructores de datos se pueden usar como patrones:

$$
\begin{array}{lll}
esFinSemana & :: & DíaSemana \to Bool \\
esFinSemana\ Sábado & = & True \\
esFinSemana\ Domingo & = & True \\
esFinSemana\ _ & = & False
\end{array}
$$

Otro ejemplo (predefinido):

$$\textbf{data}\ Bool\ =\ False \mid True\quad \textbf{deriving}\ (Show, \ldots)$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{infixr}\ 3\ \&\&} \\
(\&\&) & :: & Bool \to Bool \to Bool \\
False\ \&\&\ x & = & False \\
True\ \&\&\ x & = & x
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{infixr}\ 2\ ||} \\
(||) & :: & Bool \to Bool \to Bool \\
False\ ||\ x & = & x \\
True\ ||\ x & = & True
\end{array}
$$

Uniones

✓ Unión de varios tipos existentes:

$$\textbf{data}\ LetraOEntero = Letra\ Char \mid Entero\ Integer\ \ \textbf{deriving}\ Show$$

✓ Los valores del tipo *LetraOEntero* son:

- ◇ Los valores del tipo *Char* precedidos del constructor *Letra*
- ◇ Los valores del tipo *Integer* precedidos del constructor *Entero*

```
unValor :: LetraOEntero
unValor = Letra 'x'

otroValor :: LetraOEntero
otroValor = Entero 15

listaMixta :: [LetraOEntero]
listaMixta = [ Letra 'a', Entero 10, Entero 12, Letra 'b']
```

✓ Los constructores los elige el programador pero **son obligatorios**

✓ Cada constructor introducido tiene un tipo (no hay que declararlo)

```
Letra  :: Char → LetraOEntero
Entero :: Integer →LetraOEntero
```

✓ Los constructores de datos pueden actuar como patrones y funciones:

```
incLoE            :: LetraOEntero → LetraOEntero
incLoE (Entero n) = Entero (n + 1)
incLoE (Letra c)  = Letra (chr (1 + ord c))

   ? incLoE (Letra 'a')
   Letra 'b' :: LetraOEntero

   ? incLoE (Entero 10)
   Entero 11 :: LetraOEntero
```

Productos

✓ Tipos con un único constructor y varias componentes

$$\textbf{data}\ Racional\ =\ Par\ \underbrace{Integer}_{Numerador}\ \underbrace{Integer}_{Denom.}\ \ \textbf{deriving}\ Show$$

✓ Los valores del tipo $Racional$ son cualesquiera dos valores de tipo $Integer$ precedidos del constructor Par:

$$\begin{aligned} &unMedio\ ::\ Racional \\ &unMedio\ =\ Par\ 1\ 2 \end{aligned}$$

✓ Tipo del constructor (no hay que declararlo):

$$Par\ ::\ Integer \rightarrow Integer \rightarrow Racional$$

Ejemplos:

$$\begin{aligned} &numerador\ ::\ Racional \rightarrow Integer \\ &numerador\ (Par\ x\ _)\ =\ x \end{aligned}$$

$$\begin{aligned} &denominador\ ::\ Racional \rightarrow Integer \\ &denominador\ (Par\ _\ y)\ =\ y \end{aligned}$$

$$\begin{aligned} &\textbf{infixl}\ 7\ \ {>}{*}{<} \\ &({>}{*}{<})\ ::\ Racional \rightarrow Racional \rightarrow Racional \\ &(Par\ a\ b)\ {>}{*}{<}\ (Par\ c\ d)\ =\ Par\ (a * c)\ (b * d) \end{aligned}$$

? $numerador\ (Par\ 1\ 3)$
$1\ ::\ Integer$

? $(Par\ 1\ 2)\ {>}{*}{<}\ (Par\ 1\ 3)$
$Par\ 1\ 6\ ::\ Racional$

Constructores simbólicos

- ✓ Si un constructor de datos es binario su nombre puede ser simbólico
- ✓ Pueden mejorar legibilidad
- ✓ Ha de comenzar por el carácter dos puntos (:)
- ✓ El constructor se escribe entre las dos componentes del tipo (infijo)

$$\textbf{infix}\ 9\ :/$$
$$\textbf{data}\ Racional = \underbrace{Integer}_{Num.}\ :/\ \underbrace{Integer}_{Denom.}\quad \textbf{deriving}\ Show$$

- ✓ Los valores del tipo *Racional* son dos valores de tipo *Integer* con el constructor (:/) infijo:

$$unMedio :: Racional$$
$$unMedio = 1 :/ 2$$

- ✓ Tipo del constructor (no hay que declararlo):

$$(:/) :: Integer \rightarrow Integer \rightarrow Racional$$

Ejemplos:

$$numerador :: Racional \rightarrow Integer$$
$$numerador\ (x\ :/\ _) = x$$

$$denominador :: Racional \rightarrow Integer$$
$$denominador\ (_\ :/\ y) = y$$

$$\textbf{infixl}\ 7\ >*<$$
$$(>*<) :: Racional \rightarrow Racional \rightarrow Racional$$
$$a :/ b\ >*<\ c :/ d = (a * c)\ :/\ (b * d)$$

? $numerador\ (1 :/ 3)$
$1 :: Integer$

? $1 :/ 2\ >*<\ 1 :/ 3$
$1 :/ 6 :: Racional$

Registros variantes

- ✓ Mezcla de Uniones, Productos y Enumerados
- ✓ Permiten expresar distintas formas para valores de un mismo tipo
- ✓ Cada forma puede tener un número distinto de componentes

Ejemplo: Tipo para representar cuatro clases de figuras

```
type Radio  = Float
type Lado   = Float
type Base   = Float
type Altura = Float

data Figura = Círculo Radio
            | Cuadrado Lado
            | Rectángulo Base Altura
            | Punto
              deriving Show

unCírculo :: Figura
unCírculo = Círculo 25

unRectángulo :: Figura
unRectángulo = Rectángulo 10 15

listaFiguras :: [Figura]
listaFiguras = [Círculo 15, Cuadrado 3, Rectángulo 5 6]

área                  :: Figura → Float
área (Círculo r)      = pi * r ^ 2
área (Cuadrado l)     = l ^ 2
área (Rectángulo b h) = b * h
área Punto            = 0
```

Tipos recursivos

- ✓ Alguna componente de un constructor puede tener el tipo que se está definiendo

- ✓ Permiten definir tipos con cardinalidad infinita

Ejemplo: Los naturales

$$\textbf{data}\ \mathit{Nat} = \mathit{Cero} \mid \mathit{Suc}\ \mathit{Nat}\ \ \textbf{deriving}\ \mathit{Show}$$

- ✓ Un valor de tipo Nat puede tener dos formas
  - ◇ La constante Cero
  - ◇ El constructor Suc seguido de otro valor de tipo Nat

- ✓ Iterando la segunda forma se generan los naturales distintos a Cero:

$$\underbrace{\mathit{Suc}\ \mathit{Cero}}_{\mathit{uno}} \quad \underbrace{\mathit{Suc}\ (\mathit{Suc}\ \mathit{Cero})}_{\mathit{dos}} \quad \underbrace{\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ \mathit{Cero}))}_{\mathit{tres}} \quad \ldots$$

- ✓ Incluso se puede definir ∞

$$\begin{array}{lcl} \mathit{inf} & :: & \mathit{Nat} \\ \mathit{inf} & = & \mathit{Suc}\ \mathit{inf} \end{array}$$

? inf
$\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ \ldots$

Tipos recursivos (2)

✓ La recursión permite definir funciones que actúen sobre el tipo Nat de un modo elegante.

Ejemplos:

$$
\begin{array}{lll}
esPar & :: & Nat \rightarrow Bool \\
esPar\ Cero & = & True \\
esPar\ (Suc\ n) & = & not\ (esPar\ n)
\end{array}
$$

$$
\begin{array}{lll}
\textbf{infixl}\ 6\ <+> & & \\
(<+>) & :: & Nat \rightarrow Nat \rightarrow Nat \\
Cero\ \ <+>\ m & = & m \\
(Suc\ n)\ <+>\ m & = & Suc\ (n\ <+>\ m) \\
& & \text{-- ya que } (n+1)+m = (n+m)+1
\end{array}
$$

$$
\begin{array}{lll}
\textbf{infixl}\ 7\ <*> & & \\
(<*>) & :: & Nat \rightarrow Nat \rightarrow Nat \\
Cero\ \ <*>\ m & = & Cero \\
(Suc\ n)\ <*>\ m & = & n\ <*>\ m\ \ <+>\ m \\
& & \text{-- ya que } (n+1)*m = n*m+m
\end{array}
$$

Ejemplo de evaluación:

$$
\begin{array}{l}
\underline{Suc\ (Suc\ Cero)\ <+>\ Suc\ Cero} \\
\Longrightarrow \{\text{segunda ecuación de } (<+>)\} \\
Suc\ (\underline{(Suc\ Cero)\ <+>\ Suc\ Cero}) \\
\Longrightarrow \{\text{segunda ecuación de } (<+>)\} \\
Suc\ (Suc\ (\underline{Cero\ <+>\ Suc\ Cero})) \\
\Longrightarrow \{\text{primera ecuación de } (<+>)\} \\
Suc\ (Suc\ (Suc\ Cero))
\end{array}
$$

Plegado para el tipo Nat

✓ Muchas funciones sobre el tipo Nat siguen el mismo esquema:

$$\begin{array}{lcl} esPar & :: & Nat \to Bool \\ esPar\ Cero & = & True \\ esPar\ (Suc\ n) & = & not\ (esPar\ n) \end{array}$$

$$\begin{array}{lcl} aInteger & :: & Nat \to Integer \\ aInteger\ Cero & = & 0 \\ aInteger\ (Suc\ n) & = & 1 + (aInteger\ n) \end{array}$$

✓ El esquema común es:

$$\begin{array}{lcl} fun & :: & Nat \to a \\ fun\ Cero & = & \boxed{e} \\ fun\ (Suc\ n) & = & \boxed{f}\ (fun\ n) \end{array}$$

✓ Una función de orden superior para este esquema

$$\begin{array}{lcl} foldNat & :: & (a \to a) \to a \to (Nat \to a) \\ foldNat\ f\ e & = & fun \\ \quad \mathbf{where} & & \\ \qquad fun\ Cero & = & e \\ \qquad fun\ (Suc\ n) & = & f\ (fun\ n) \end{array}$$

✓ O equivalentemente (ya que $fun \equiv foldNat\ f\ e$)

$$\begin{array}{lcl} foldNat & :: & (a \to a) \to a \to Nat \to a \\ foldNat\ f\ e\ Cero & = & e \\ foldNat\ f\ e\ (Suc\ n) & = & f\ (foldNat\ f\ e\ n) \end{array}$$

✓ Las funciones originales como concreción de $foldNat$:

$$\begin{array}{lcl} esPar & :: & Nat \to Bool \\ esPar & = & foldNat\ not\ True \end{array}$$

$$\begin{array}{lcl} aInteger & :: & Nat \to Integer \\ aInteger & = & foldNat\ (1+)\ 0 \end{array}$$

6.3 Tipos Polimórficos

Los tipos también pueden ser polimórficos.

Either

Tipo predefinido para representar la unión de otros dos tipos arbitrarios:

$$\textbf{data}\ \ Either\ a\ b\ =\ \ Left\ a\ |\ Right\ b\ \ \textbf{deriving}\ Show$$

✓ Los valores de tipo $Either\ a\ b$ son los valores de tipo a precedidos del constructor $Left$ y los valores de tipo b precedidos de $Right$.

Ejemplo: Listas con enteros y booleanos:

$$\begin{aligned} l1 &:: [Either\ Integer\ Bool] \\ l1 &= [Left\ 1,\ Right\ True,\ Left\ 3,\ Left\ 5] \\ l2 &:: [Either\ Bool\ Integer] \\ l2 &= [Rigth\ 2,\ Left\ False,\ Right\ 5] \end{aligned}$$

Maybe

Tipo predefinido para representar valores parciales:

$$\textbf{data}\ Maybe\ a\ =\ \ Nothing\ |\ Just\ a\ \ \textbf{deriving}\ Show$$

✓ Los valores de tipo $Maybe\ a$ son los valores de tipo a precedidos de $Just$ y además un valor especial que se escribe $Nothing$

✓ $Nothing$ se suele usar para representar *no definido*:

$$\begin{aligned} recíproco &:: Float \to Maybe\ Float \\ recíproco\ 0 &= Nothing \\ recíproco\ x &= Just\ (1/x) \end{aligned}$$

Listas

Podemos definir una lista polimórfica homogénea (todos los elementos tienen el mismo tipo):

$$\textbf{data}\ \ \mathit{Lista}\ a\ =\ \ \mathit{Vacía}\ |\ \mathit{Cons}\ \underbrace{a}_{cabeza}\ \underbrace{(\mathit{Lista}\ a)}_{cola}\ \ \textbf{deriving}\ \mathit{Show}$$

Las listas definidas tienen dos formas posibles:

✓ Puede ser la lista vacía, representada por *Vacía*

✓ Puede ser una lista no vacía, representada por *Cons cabeza cola* donde la cabeza ha de tener tipo a y la cola tipo *Lista a*

Ejemplos:

```
l3 :: Lista Integer
l3 = Cons 1 (Cons 2 (Cons 3 Vacía))

l4 :: Lista Bool
l4 = Cons True (Cons True (Cons False Vacía))
```

Para estructuras lineales es mejor usar un constructor simbólico:

$$\textbf{infixr}\ 5\ :>$$
$$\textbf{data}\ \ \mathit{Lista}\ a\ =\ \ \mathit{Vacía}\ |\ \underbrace{a}_{cabeza}\ :>\ \underbrace{(\mathit{Lista}\ a)}_{cola}\ \ \textbf{deriving}\ \mathit{Show}$$

```
l3 :: Lista Integer
l3 = 1 :> 2 :> 3 :> Vacía
```

Ejemplo poco práctico, ya que las listas están predefinidas

Objetivos del tema

El alumno debe:

- ✓ Saber definir y utilizar sinónimos de tipos
- ✓ Saber definir tipos enumerados, uniones, productos, registros variantes y tipos recursivos
- ✓ Entender las definiciones de tipo
- ✓ Saber definir funciones sobre tipos definidos
- ✓ Saber reducir expresiones en las que aparezcan tipos definidos
- ✓ Entender la función de plegado $foldNat$ y saber definir otras funciones como concreciones de ésta
- ✓ Saber definir y utilizar tipos polimórficos

Tema 7. El sistema de clases

7.1 Funciones Sobrecargadas

Clases e Instancias

7.2 Algunas Clases e Instancias predefinidas

La clase Eq

La clase Ord

Las clases Show y Read

Las clases Num y Fractional

7.3 Derivación de instancias

7.4 Tipos sobrecargados: Contextos

7.1 Funciones Sobrecargadas

- ✓ Tienen sentido para algunos tipos, pero no todos
- ✓ Pueden tener definiciones distintas para cada tipo

Ejemplo: Consideremos los tipos

$$
\begin{array}{lcl}
\textbf{type}\ Lado & = & Float \\
\textbf{type}\ Radio & = & Float \\
\textbf{type}\ Área & = & Float \\
\textbf{data}\ Cuadrado & = & UnCuadrado\ Lado\ \ \textbf{deriving}\ Show \\
\textbf{data}\ Círculo & = & UnCírculo\ Radio\ \ \textbf{deriving}\ Show
\end{array}
$$

Tiene sentido definir una función para calcular el área de un *Cuadrado*:

$$
\begin{array}{lcl}
área & :: & Cuadrado \to Área \\
área\ (UnCuadrado\ l) & = & l\ *\ l
\end{array}
$$

O para un *Círculo*:

$$
\begin{array}{lcl}
área & :: & Círculo \to Área \\
área\ (UnCírculo\ r) & = & pi\ *\ r\ \wedge\ 2
\end{array}
$$

¿ Es el tipo de $área\ ::\ a \to Área$?

NO, no tiene sentido, p. ej., calcular el área de un *Bool*

área tiene sentido para los tipos *Cuadrado* y *Círculo*, pero no para *Bool*, luego no es polimórfica

Clases e Instancias

- ✓ *área* tiene sentido para varios tipos pero *NO* para todos
- ✓ *área* tiene una definición *DISTINTA* para cada tipo

área es un ejemplo de función *Sobrecargada*

En Haskell, para definir función sobrecargada hay que crear una *clase* (conjunto de tipos que implementan la función):

```
class TieneÁrea t where
  área :: t → Área
```

- ✓ *TieneÁrea* es el nombre de la clase (empieza por mayúscula)
- ✓ t es una variable de tipo que representa los tipos de la clase
- ✓ El *método* *área* solo estará definido para los t que pertenezcan a la clase

Para incluir un tipo en una clase se realiza una *instancia*

```
instance TieneÁrea Cuadrado where
  área (UnCuadrado l) = l * l

instance TieneÁrea Círculo where
  área (UnCírculo r) = pi * r ^ 2
```

Uso:

```
? área (UnCuadrado 3)      -- Se usa primera instancia
9.0 :: Área

? área (UnCírculo 3)       -- Se usa segunda instancia
28.2743 :: Área

? área True                -- No existe instancia adecuada
ERROR ...
```

7.2 Algunas Clases e Instancias predefinidas

✓ Haskell organiza los tipos predefinidos en clases de tipos.

Clase: conjunto de tipos para los que tiene sentido una serie de operaciones sobrecargadas.

✓ Algunas de las clases predefinidas:

- Eq tipos que definen igualdad: $(==)$ y $(/=)$
- Ord tipos que definen un orden: $(<=)$, $(<)$, $(>=)$, ...
- Num tipos numéricos: $(+)$, $(-)$, $(*)$, ...

Instancias: conjunto de tipos pertenecientes a una clase.

✓ Algunas instancias predefinidas:

- Eq tipos que definen igualdad: *Bool*, *Char*, *Int*, *Integer*, *Float*, *Double*, ...
- Ord tipos que definen un orden: *Bool*, *Char*, *Int*, *Integer*, *Float*, *Double*, ...
- Num tipos numéricos: *Int*, *Integer*, *Float* y *Double*

La clase Eq

- Tipos *igualables*

$$
\begin{array}{l}
\textbf{class } Eq\ a \textbf{ where} \\
\quad (==) \quad :: \ a \rightarrow a \rightarrow Bool \\
\quad (/=) \quad :: \ a \rightarrow a \rightarrow Bool
\end{array}
$$

−− Mínimo a implementar: (==) o bien (/=)

$$
\begin{array}{lll}
x == y & = & not\ (x \mathrel{/=} y) \\
x \mathrel{/=} y & = & not\ (x == y)
\end{array}
$$

- Las definiciones en la clase de (==) y (/=) son *Métodos por defecto*
- Se usan si no se definen en las instancias
- Basta con definir uno de los dos

La clase Ord

✓ Tipos *ordenables*

```
class Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a
  ...

  -- Mínimo a implementar: (<=)

  x <  y = x <= y && x /= y
  x >= y = y <= x
  x >  y = y <  x

  max x y
    | x >= y    = x
    | otherwise = y

  min x y
    | x <= y    = x
    | otherwise = y
  ...
```

✓ Cualquier tipo instancia de Ord debe ser instancia de Eq (no necesariamente lo contrario)

Las clases Show y Read

✓ Tipos mostrables

$$\textbf{class}\ \ Show\ a\ \textbf{where}$$
$$\quad show\ :: a \to String$$
$$\quad \ldots$$

✓ Tipos leíbles

$$\textbf{class}\ \ Read\ a\ \textbf{where}$$
$$\quad \ldots$$

$$read\ \ :: Read\ a \Rightarrow String \to a$$
$$read\ s = \ \ldots$$

Las clases Num y Fractional

$$\textbf{class}\ \ Num\ a\ \textbf{where}$$
$$\quad (+),\ (-),\ (*)\ :: a \to a \to a$$
$$\quad negate\ :: a \to a$$
$$\quad abs,\ signum\ :: a \to a$$
$$\quad fromInteger\ :: Integer \to a$$

−− Mínimo a implementar: todos, excepto *negate* o $(-)$

$$x - y = x + negate\ y$$
$$negate\ x = 0 - x$$

$$\textbf{class}\ \ Fractional\ a\ \textbf{where}$$
$$\quad (/)\ :: a \to a \to a$$
$$\quad recip\ :: a \to a$$
$$\quad \ldots$$

$$recip\ x = 1\ /\ x$$
$$x\ /\ y = x * recip\ y$$
$$\ldots$$

7.3 Derivación de instancias

✓ La claúsula **deriving** permite generar instancias de ciertas clases predefinidas de forma automática.

✓ Aparece al final de una declaración de tipo

data *Color* = *Rojo* | *Amarillo* | *Azul* | *Verde* **deriving** (*Eq*, *Ord*, *Show*, *Read*)

? *Rojo* == *Verde*
False :: *Bool*

? *Rojo* < *Verde*
True :: *Bool*

? *show* *Rojo*
"Rojo" :: *String*

? *read* "Rojo" :: *Color*
Rojo :: *Color*

✓ Al derivar la clase *Eq* se usa *igualdad estructural*:

- Dos valores son iguales si tienen la misma forma

EJEMPLO:

infix 9 :/
data *Racional* = *Integer* :/ *Integer* **deriving** *Eq*

genera

instance *Eq* *Racional* **where**
$x :/ y \;==\; x' :/ y' \;=\; (x == x') \;\&\&\; (y == y')$

? 1 :/2 == 2 :/4
False :: *Bool*

✓ La igualdad estructural no es adecuada en este caso

Derivación de instancias (2)

EJEMPLO:

$$\textbf{data}\ \ Nat = \ \ Cero \mid Suc\ Nat\ \ \textbf{deriving}\ Eq$$

genera

$$
\begin{array}{l}
\textbf{instance}\ \ Eq\ Nat\ \textbf{where} \\
\quad \begin{array}{lcll}
Cero & == & Cero & = \ \ True \\
Suc\ x & == & Suc\ y & = \ \ (x == y) \\
_ & == & _ & = \ \ False
\end{array}
\end{array}
$$

✓ La igualdad estructural es adecuada en este caso

Al derivar la clase Ord se usa *orden estructural*.

- ◇ Es adecuado para tipos enumerados
- ◇ No es adecuado en todos los casos.

7.4 Tipos sobrecargados: Contextos

✓ Los métodos de una clase solo pueden ser usados con tipos instancia de dicha clase

✓ Ésto queda reflejado en el tipo de los métodos:

```
? :t (==)
(==) :: Eq a ⇒ a → a → Bool

? :t (+)
(+) :: Num a ⇒ a → a → a

? True + False
ERROR – Illegal Haskell 98 class constraint in inferred type
*** Expression : True + False
*** Type       : Num Bool ⇒ Bool
```

✓ El *contexto* establece una restricción sobre el polimorfismo de la variable.

✓ *Propagación de contextos*:

Si función polimórfica usa una sobrecargada se convierte en sobrecargada.

```
doble   :: Num a ⇒ a → a
doble x = x + x

elem              :: Eq a ⇒ a → [a] → Bool
x `elem` []       = False
x `elem` (y : ys) = x == y || x `elem` ys

f               :: (Ord a, Num a) ⇒ a → a → a
f x y
  | x < y     = x
  | otherwise = x + y
```

Objetivos del tema

El alumno debe:

- ✓ Comprender el concepto de sobrecarga y diferenciarlo del concepto de polimorfismo
- ✓ Saber definir clases e instancias para sobrecargar funciones
- ✓ Conocer las clases predefinidas expuestas
- ✓ Conocer la posibilidad de derivar instancias.
- ✓ Conocer la igualdad estructural generado para instancias derivadas
- ✓ Saber si una instancia derivada es adecuada
- ✓ Entender los tipos sobrecargados (*contextos*)
- ✓ Saber cuál es el tipo de una función sobrecargada a partir de su definición

Tema 8. Listas

8.1 Secuencias aritméticas

8.2 Algunas funciones predefinidas

8.3 Listas por comprensión

Ejemplo: QuickSort

8.4 Funciones de plegado

8.5 Listas infinitas

Ejemplo: Raíz cuadrada de un número

Ejemplo: La criba de Eratóstenes

8.1 Secuencias aritméticas

✓ Sintaxis para definir listas que se puede usar con los tipos predefinidos.

Ejemplos:

✓ Lista con enteros entre 1 y 10 (de uno en uno)

? $[1 \,..\, 10]$
$[1,2,3,4,5,6,7,8,9,10] \; :: \; [Integer]$

✓ De dos en dos (se especifican los dos primeros elementos)

? $[1,3 \,..\, 11]$
$[1,3,5,7,9,11] \; :: \; [Integer]$

✓ En orden decreciente.

? $[10,9 \,..\, 1]$
$[10,9,8,7,6,5,4,3,2,1] \; :: \; [Integer]$

✓ Si no se especifica el elemento final, se pueden obtener listas infinitas:

? $[1 \,..\,]$
$[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 \ldots$

8.2 Algunas funciones predefinidas

✓ Selectores básicos

```
? head [1 .. 5]                -- head :: [a] → a
1 :: Integer                   -- cabeza de la lista

? tail [1 .. 5]                -- tail :: [a] → [a]
[2, 3, 4, 5] :: [Integer]      -- cola de la lista

? last [1 .. 5]                -- last :: [a] → a
5 :: Integer                   -- último de la lista

? init [1 .. 5]                -- init :: [a] → [a]
[1, 2, 3, 4] :: [Integer]      -- inicio de la lista
```

✓ Más selectores

```
? take 3 [1 .. 5]              -- take :: Int → [a] → [a]
[1, 2, 3] :: [Integer]         -- toma

? drop 3 [1 .. 5]              -- drop :: Int → [a] → [a]
[4, 5] :: [Integer]            -- quita

? [1 .. 5] !! 3                -- (!!) :: [a] → Int → a
4 :: Integer                   -- selecciona
```

✓ *map* y filtros

```
? map (*10) [2, 3, 4, 6]                      -- map :: (a → b) → [a] → [b]
[20, 30, 40, 60] :: [Integer]                 -- aplicar a todos

? filter even [2, 4, 8, 9, 10, 11, 12]        -- filter :: (a → Bool) → [a] → [a]
[2, 4, 8, 10, 12] :: [Integer]                -- filtrar

? takeWhile even [2, 4, 8, 9, 10, 11, 12]     -- takeWhile :: (a → Bool) → [a] → [a]
[2, 4, 8] :: [Integer]                        -- mayor segmento inicial
```

Algunas funciones predefinidas (2)

✓ Concatenación

```
? [1 .. 5] ++ [10 .. 13]                      -- (++) :: [a] → [a] → [a]
[1,2,3,4,5,10,11,12,13] :: [Integer]          -- de dos listas

? concat [ [1,2,3], [6,7], [9,10,11,12] ]     -- concat :: [[a]] → [a]
[1,2,3,6,7,9,10,11,12] :: [Integer]           -- de lista de listas
```

✓ Numéricas

```
? sum [1 .. 5]          -- sum :: Num a ⇒ [a] → a
15 :: Integer           -- sumar elementos

? product [1 .. 5]      -- product :: Num a ⇒ [a] → a
120 :: Integer          -- multiplicar elementos
```

✓ Orden

```
? maximum [10,4,15,2]   -- maximum :: Ord a ⇒ [a] → a
15 :: Integer           -- máximo

? minimum [10,4,15,2]   -- minimum :: Ord a ⇒ [a] → a
2 :: Integer            -- mínimo
```

✓ Emparejamiento

```
? zip [1,2,3,4] ['a', 'b', 'c']                       -- zip :: [a]→[b]→[(a, b)]
[(1, 'a'), (2, 'b'), (3, 'c')] :: [ (Integer, Char) ] -- emparejar

? unzip [ (1, 'a'), (2, 'b'), (3, 'c') ]              -- unzip :: [(a, b)]→([a], [b])
( [1,2,3], ['a', 'b', 'c'] ) :: ( [Integer], [Char] ) -- desemparejar

? zipWith (+) [1,2,3] [10,20,30]          -- zipWith :: (a→ b→ c) → [a]→[b]→[c]
[11,22,33] :: [Integer]                   -- emparejar con
```

8.3 Listas por comprensión

✓ Similar a los *conjuntos por comprensión* en *matemáticas*

✓ Sintaxis:

$[\ expr \mid qual_1,\ qual_2,\ \ldots\ ,\ qual_n\ ]$

✓ Un cualificador puede ser un:

◇ Un *generador* ($patrón \leftarrow expr$) con $expr$ de tipo lista:

? $[\ x \wedge 2 \mid x \leftarrow [1\ ..\ 5]\ ]$
$[1, 4, 9, 16, 25]\ ::\ [Integer]$

◇ Un *filtro* o *guarda* (expresión de tipo $Bool$):

? $[\ x \mid x \leftarrow [1\ ..\ 10],\ even\ x\ ]$
$[2, 4, 6, 8, 10]\ ::\ [Integer]$

◇ Una *definición local* ($\textbf{let}\ patrón = expr$):

? $[\ (x, y) \mid x \leftarrow [1\ ..\ 5],\ \textbf{let}\ y = 2 * x\ ]$
$[\ (1, 2),\ (2, 4),\ (3, 6),\ (4, 8),\ (5, 10)\ ]\ ::\ [(Integer, Integer)]$

✓ Varios *generadores* (los últimos cambian más rápido)

? $[\ (x, y) \mid x \leftarrow [1\ ..\ 3],\ y \leftarrow [10, 20]\ ]$
$[\ (1, 10),\ (1, 20),\ (2, 10),\ (2, 20),\ (3, 10),\ (3, 20)\ ]\ ::\ [(Integer, Integer)]$

? $[\ (x, y) \mid y \leftarrow [10, 20],\ x \leftarrow [1\ ..\ 3]\ ]$
$[\ (1, 10),\ (2, 10),\ (3, 10),\ (1, 20),\ (2, 20),\ (3, 20)\ ]\ ::\ [(Integer, Integer)]$

✓ Un *generador* o *def. local* puede depender de otro previo:

? $[\ (x, y) \mid x \leftarrow [1, 2],\ y \leftarrow [x\ ..\ 3]\ ]$
$[\ (1, 1),\ (1, 2),\ (1, 3),\ (2, 2),\ (2, 3)\ ]\ ::\ [(Integer, Integer)]$

? $[\ (x, y) \mid x \leftarrow [y, 2],\ y \leftarrow [1\ ..\ 3]\ ]$
$ERROR - Undefined\ variable$ "y"

Listas por comprensión (2)

Algunos ejemplos:

✓ La función *map*:

$$\begin{aligned} map &:: (a \to b) \to [a] \to [b] \\ map\ f\ xs &= [\ f\ x \mid x \leftarrow xs\ ] \end{aligned}$$

✓ La función *filter*:

$$\begin{aligned} filter &:: (a \to Bool) \to [a] \to [a] \\ filter\ p\ xs &= [\ x \mid x \leftarrow xs,\ p\ x\ ] \end{aligned}$$

✓ Divisores de un número natural:

$$\begin{aligned} divideA &:: Integer \to Integer \to Bool \\ d\ `divideA`\ n &= (n\ `mod`\ d == 0) \end{aligned}$$

$$\begin{aligned} divisores &:: Integer \to [Integer] \\ divisores\ n &= [\ x \mid x \leftarrow [1\ ..\ n],\ x\ `divideA`\ n\ ] \end{aligned}$$

✓ Máximo común divisor de dos números:

$$\begin{aligned} mcd &:: Integer \to Integer \to Integer \\ mcd\ x\ y &= maximum\ [\ n \mid n \leftarrow divisores\ x,\ n\ `divideA`\ y\ ] \end{aligned}$$

✓ Posiciones de un dato en una lista:

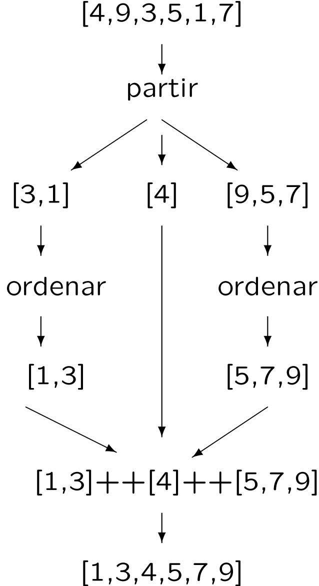
$$\begin{aligned} posiciones &:: Eq\ a \Rightarrow a \to [a] \to [Integer] \\ posiciones\ x\ xs &= [\ p \mid (p, y) \leftarrow zip\ [0\ ..\ ]\ xs,\ x == y\ ] \end{aligned}$$

```
? posiciones 'a' "la casa"
[1, 4, 6] :: [Integer]
```

Ejemplo: QuickSort

✓ Método para ordenar una lista

- ◇ Tomar el primer elemento de la lista (*pivote*).
- ◇ Partir la cola de la lista en dos: los elementos menores al pivote y los demás.
- ◇ Ordenar cada una de estas listas.
- ◇ A partir de las dos listas ordenadas, obtener la lista original ordenada concatenando la primera con el pivote y la segunda.



✓ En Haskell:

$$
\begin{array}{lcl}
qSort & :: & Ord\ a \Rightarrow [a] \to [a] \\
qSort\ [\,] & = & [\,] \\
qSort\ (p : xs) & = & qSort\ menores +\!\!+ [p] +\!\!+ qSort\ mayores \\
\quad \textbf{where} & & \\
\qquad menores & = & [\ x \mid x \leftarrow xs,\ x < p\ ] \\
\qquad mayores & = & [\ x \mid x \leftarrow xs,\ x >= p\ ]
\end{array}
$$

8.4 Funciones de plegado

- ✓ *foldr* captura un patrón recursivo habitual sobre listas
- ✓ Consideremos

$$\begin{array}{lcl} suma & :: & [Integer] \rightarrow Integer \\ suma\ [\,] & = & 0 \\ suma\ (x : xs) & = & (+)\ x\ (suma\ xs) \end{array}$$

$$\begin{array}{lcl} producto & :: & [Integer] \rightarrow Integer \\ producto\ [\,] & = & 1 \\ producto\ (x : xs) & = & (*)\ x\ (producto\ xs) \end{array}$$

- ✓ Ambas siguen el mismo patrón:

$$\begin{array}{lcl} fun & :: & [a] \rightarrow b \\ fun\ [\,] & = & \boxed{e} \\ fun\ (x : xs) & = & \boxed{f}\ x\ (fun\ xs) \end{array}$$

- ✓ Una función de orden superior para este esquema

$$\begin{array}{lcl} foldr & :: & (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow ([a] \rightarrow b) \\ foldr\ f\ e & = & fun \\ \quad \mathbf{where} & & \\ \qquad fun\ [\,] & = & e \\ \qquad fun\ (x : xs) & = & f\ x\ (fun\ xs) \end{array}$$

- ✓ O equivalentemente (ya que $fun \equiv foldr\ f\ e$)

$$\begin{array}{lcl} foldr & :: & (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ foldr\ f\ e\ [\,] & = & e \\ foldr\ f\ e\ (x : xs) & = & f\ x\ (foldr\ f\ e\ xs) \end{array}$$

- ✓ Las funciones originales como concreción de *foldr*:

$$\begin{array}{lcl} suma & :: & [Integer] \rightarrow Integer \\ suma & = & foldr\ (+)\ 0 \end{array}$$

$$\begin{array}{lcl} producto & :: & [Integer] \rightarrow Integer \\ producto & = & foldr\ (*)\ 1 \end{array}$$

Funciones de plegado (2)

✓ Es más fácil ver el comportamiento de *foldr* del siguiente modo:

Comportamiento de *foldr*

$$foldr\ (\otimes)\ z\ [x_1, x_2, \ldots, x_n] \Longrightarrow x_1 \otimes (x_2 \otimes (\ldots \otimes (x_n \otimes z)))$$

Por ejemplo

$\underline{suma}\ [1,2,3]$

$\Longrightarrow$ {definición de *suma*}

$\underline{foldr\ (+)\ 0\ [1,2,3]}$

$\Longrightarrow$ {comportamiento de *foldr*}

...

$1 + (2 + (3 + 0))$

$\Longrightarrow$ {por (+)}

...

6

✓ Más ejemplos:

```
and :: [Bool] → Bool        -- Conjunción de booleanos
and = foldr (&&) True

or :: [Bool] → Bool         -- Disyunción de booleanos
or = foldr (||) False

concat :: [[a]] → [a]       -- Concatenación de lista de listas
concat = foldr (++) []
```

Funciones de plegado (3)

✓ *foldl* pliega la lista de izquierda a derecha

$$
\begin{array}{lcl}
foldl & :: & (b \to a \to b) \to b \to [a] \to b \\
foldl\ f\ e\ [\,] & = & e \\
foldl\ f\ e\ (x : xs) & = & foldl\ f\ (f\ e\ x)\ xs
\end{array}
$$

Comportamiento de *foldl*

$$foldl\ (\otimes)\ z\ [x_1, x_2, \ldots, x_n] \Longrightarrow (((z \otimes x_1) \otimes x_2) \ldots \otimes x_{n-1}) \otimes x_n$$

✓ Por ejemplo:

$$
\begin{array}{lcl}
suma & :: & [Integer] \to Integer \\
suma & = & foldl\ (+)\ 0
\end{array}
$$

$suma\ [1,2,3]$

$\Longrightarrow$ {definición de *suma*}

$foldl\ (+)\ 0\ [1,2,3]$

$\Longrightarrow$ {comportamiento de *foldl*}

...

$((0+1)+2)+3$

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

...

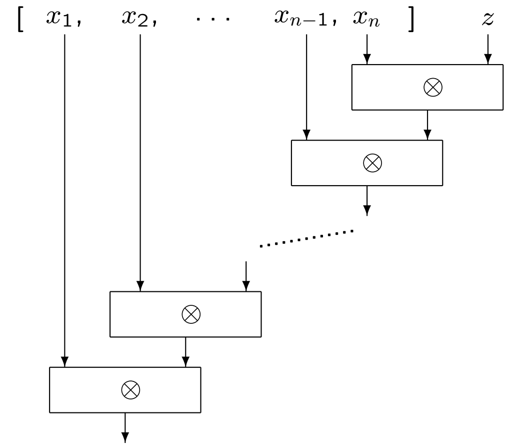
6

Funciones de plegado (4)

✓ Gráficamente

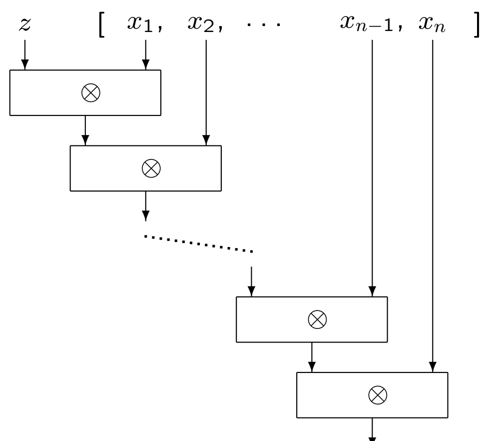
Comportamiento de *foldr*

$$foldr\ (\otimes)\ z\ [x_1, x_2, \ldots, x_n] \Longrightarrow x_1 \otimes (x_2 \otimes (\ldots \otimes (x_n \otimes z)))$$



Comportamiento de *foldl*

$$foldl\ (\otimes)\ z\ [x_1, x_2, \ldots, x_n] \Longrightarrow (((z \otimes x_1) \otimes x_2) \ldots \otimes x_{n-1}) \otimes x_n$$



Funciones de plegado (5)

- ✓ No todas las funciones se definen igual usando *foldr* y *foldl*

- ✓ Para resolver un problema usando *foldr f z*
  - ◇ z será la solución para la lista vacía
  - ◇ f tomará
    - ○ como primer argumento la cabeza de la lista y
    - ○ como segundo argumento la solución del problema para la cola

$$\begin{aligned} reverse &:: [a] \rightarrow [a] \\ reverse &= foldr\ (\lambda\ x\ xs \rightarrow xs +\!\!+ [x])\ [\,] \end{aligned}$$

- ✓ Para resolver un problema usando *foldl f z*
  - ◇ z será la solución para la lista vacía
  - ◇ f tomará
    - ○ como primer argumento la solución para el inicio de la lista y
    - ○ como segundo argumento el último elemento de la lista

$$\begin{aligned} reverse &:: [a] \rightarrow [a] \\ reverse &= foldl\ (\lambda\ xs\ x \rightarrow x : xs)\ [\,] \end{aligned}$$

8.5 Listas infinitas

✓ Algunas funciones predefinidas construyen listas infinitas:

$-- \; \mathit{repeat}\ v \Longrightarrow [v,\ v,\ v,\ \ldots]$

$$\begin{aligned} \mathit{repeat} &:: a \rightarrow [a] \\ \mathit{repeat}\ x &= \mathit{xs}\ \mathbf{where}\ \mathit{xs} = x : \mathit{xs} \end{aligned}$$

$-- \; \mathit{cycle}\ [v_1, v_2, \ldots, v_n] \Longrightarrow [v_1, v_2, \ldots, v_n, v_1, v_2, \ldots, v_n,\ \ldots]$

$$\begin{aligned} \mathit{cycle} &:: [a] \rightarrow [a] \\ \mathit{cycle}\ [\,] &= \mathit{error}\ "\mathit{Prelude.cycle}:\ \mathit{empty\ list}" \\ \mathit{cycle}\ \mathit{xs} &= \mathit{xs}'\ \mathbf{where}\ \mathit{xs}' = \mathit{xs} +\!\!+\ \mathit{xs}' \end{aligned}$$

$-- \; \mathit{iterate}\ f\ x \Longrightarrow [x,\ f\ x,\ f\ (f\ x),\ f\ (f\ (f\ x)),\ \ldots]$

$$\begin{aligned} \mathit{iterate} &:: (a \rightarrow a) \rightarrow a \rightarrow [a] \\ \mathit{iterate}\ f\ x &= x\ :\ \mathit{iterate}\ f\ (f\ x) \end{aligned}$$

✓ Ejemplos

-- Lista infinita de los números naturales

$$\begin{aligned} \mathit{losNaturales} &:: [\mathit{Integer}] \\ \mathit{losNaturales} &= \mathit{iterate}\ (+1)\ 0 \end{aligned}$$

-- Lista infinita con los múltiplos de un número

$$\begin{aligned} \mathit{múltiplosDe} &:: \mathit{Integer} \rightarrow [\mathit{Integer}] \\ \mathit{múltiplosDe}\ x &= \mathit{iterate}\ (+x)\ 0 \end{aligned}$$

-- Lista infinita de las potencias de un número

$$\begin{aligned} \mathit{potenciasDe} &:: \mathit{Integer} \rightarrow [\mathit{Integer}] \\ \mathit{potenciasDe}\ x &= \mathit{iterate}\ (*x)\ 1 \end{aligned}$$

Ejemplo: Raíz cuadrada de un número

✓ Método numérico para calcular la raíz cuadrada de un número.

✓ Basado en

Si x_i es una aproximación a $\sqrt{n}$,
entonces $x_{i+1} = \frac{1}{2}(x_i + \frac{n}{x_i})$ es una aproximación mejor

✓ Algoritmo para calcular $\sqrt{n}$:

1. Partir de cualquier aproximación inicial, por ejemplo $x_0 = \frac{n}{2}$
2. Obtener sucesivas aproximaciones $[x_0, x_1, x_2, \ldots]$ utilizando la fórmula anterior
3. Quedarse con la primera aproximación x_i tal que $x_i^2 \simeq n$

✓ Programa:

```
infix 4 ~=    -- Comprueba si dos números son aproximadamente iguales
(~=)      :: Double → Double → Bool
a ~= b =  abs (a − b) < precisión
  where
    precisión = 1/1000


raíz   :: Double → Double
raíz n = primeraQue esBuena aproxs
  where
    x0            = n / 2
    mejorar x     = 0.5 ∗ (x + n/x)
    aproxs        = iterate mejorar x0
    esBuena x     = (x ^ 2 ~= n)
    primeraQue p  = head . filter p

  ? raíz 100
  10.0 :: Double

  ? raíz 9
  3.00002 :: Double
```

Ejemplo: La criba de Eratóstenes

✓ Eratóstenes propuso un método para calcular todos los números primos

✓ Partir de la lista $l_0 = [2\,..\,]$

$$[\,\boxed{2}\,, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, \ldots]$$

✓ El primer elemento ($\boxed{2}$) es primo.

✓ Calcular l_1 eliminando de l_0 los múltiplos de 2:

l_1

$\Longrightarrow$ {eliminando múltiplos de 2 en l_0}

$[\,3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, 15, \cancel{16}, 17, \cancel{18}, 19, \cancel{20}, 21, \ldots]$

$\Longrightarrow$

$[\,\boxed{3}\,, 5, 7, 9, 11, 13, 15, 17, 19, 21, \ldots]$

✓ El primer elemento ($\boxed{3}$) es primo.

✓ Calcular l_2 eliminando de l_1 los múltiplos de 3:

l_2

$\Longrightarrow$ {eliminando múltiplos de 3 en l_1}

$[\,5, 7, \cancel{9}, 11, 13, \cancel{15}, 17, 19, \cancel{21}, \ldots]$

$\Longrightarrow$

$[\,\boxed{5}\,, 7, 11, 13, 17, 19, \ldots]$

✓ Repetir indefinidamente el proceso.

✓ Los elementos que aparecen al inicio de las listas $[l_0, l_1, l_2, \ldots]$ son los números primos: [2,3,5,. . .]

La criba de Eratóstenes (2)

✓ Eliminar de una lista el primer elemento y todos sus múltiplos:

$$
\begin{array}{lcl}
cribar & :: & [Integer] \to [Integer] \\
cribar\ [\,] & = & [\,] \\
cribar\ (x : xs) & = & [\ y \mid y \leftarrow xs,\ y\ \text{`}noEsMúltiploDe\text{`}\ x\ ]
\end{array}
$$

where

$$a\ \text{`}noEsMúltiploDe\text{`}\ b\ =\ (mod\ a\ b\ /=\ 0)$$

? $cribar\ [2\ ..\ 10]$
$[3, 5, 7, 9]\ ::\ [Integer]$

✓ Observación:

$$
\begin{array}{lclcl}
l_0 & \equiv & [2\ ..\ ] & & \\
l_1 & \equiv & cribar\ l_0 & & \\
l_2 & \equiv & cribar\ l_1 & \equiv & cribar\ (cribar\ l_0) \\
l_3 & \equiv & cribar\ l_2 & \equiv & cribar\ (cribar\ (cribar\ l_0)) \\
\ldots & & & &
\end{array}
$$

✓ Una función que devuelva la lista $[\ l_0,\ l_1,\ l_2,\ l_3,\ \ldots\ ]$.

$$
\begin{array}{lcl}
cribas & :: & [[Integer]] \\
cribas & = & iterate\ cribar\ [2\ ..\ ]
\end{array}
$$

✓ El primer primo está al inicio de l_0, el segundo al inicio de l_1, el tercero al inicio de l_2, etc, ...

✓ La lista infinita de los número primos es:

$$[\ head\ l_0,\ head\ l_1,\ head\ l_2,\ \ldots\ ]\ \equiv\ map\ head\ cribas$$

✓ Podemos definir:

$$
\begin{array}{lcl}
losPrimos & :: & [Integer] \\
losPrimos & = & map\ head\ cribas
\end{array}
$$

? $take\ 10\ losPrimos$
$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]\ ::\ [Integer]$

La criba de Eratóstenes (3)

✓ Algunos ejemplos.

✓ ¿Cuantos primos hay menores que 100?

> **?** $takeWhile\ (< 100)\ losPrimos$
> $[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,$
> $61,67,71,73,79,83,89,97]\ ::\ [Integer]$

> **?** $length\ (takeWhile\ (< 100)\ losPrimos)$
> $25\ ::\ Int$

✓ ¿Cuánto suman?

> **?** $sum\ (takeWhile\ (< 100)\ losPrimos)$
> $1060\ ::\ Integer$

✓ ¿Cuál es el primer primo mayor que 100?

$$
\begin{array}{lcl}
primeroQue & :: & (a \to Bool) \to [a] \to a \\
primeroQue\ p & = & head\ .\ filter\ p
\end{array}
$$

> **?** $primeroQue\ (> 100)\ losPrimos$
> $101\ ::\ Integer$

✓ ¿Cuál es el primer primo que acaba en 9?

$$
\begin{array}{lcl}
acabaEn & :: & Integer \to Integer \to Bool \\
n\ \text{`}acabaEn\text{`}\ d & = & (n\ \text{`}mod\text{`}\ 10 == d)
\end{array}
$$

> **?** $primeroQue\ (\text{`}acabaEn\text{`}\ 9)\ losPrimos$
> $19\ ::\ Integer$

Objetivos del tema

El alumno debe:

- ✓ Conocer la notación de secuencias aritméticas para definir listas
- ✓ Conocer las funciones predefinidas para listas comentadas en el tema
- ✓ Conocer la notación de listas por comprensión. Debe saber calcular el resultado de este tipo de expresiones y debe saber definir funciones usando esta notación
- ✓ Conocer las funciones de plegado predefinidas $foldr$ y $foldl$
- ✓ Saber reducir expresiones en las que aparezcan funciones de plegado
- ✓ Saber definir funciones sobre listas como concreciones de las funciones de plegado
- ✓ Conocer las funciones para definir listas infinitas y saber utilizarlas

Tema 9. Árboles

9.1 Árboles generales

9.2 Árboles binarios

9.3 Árboles de búsqueda

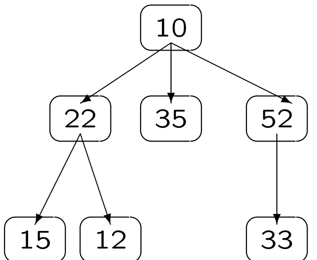
9.1 Árboles generales

✓ Un árbol es una estructura no lineal acíclica utilizada para organizar información de forma eficiente.

✓ La definición es recursiva:

✓ Un árbol es una colección de valores $\{v_1, v_2, \ldots v_n\}$ tales que

- ◇ Si $n = 0$ el árbol se dice vacío.
- ◇ En otro caso, existe un valor destacado que se denomina *raíz* (p.e. v_1), y los demás elementos forman parte de colecciones disjuntas que a su vez son árboles. Estos árboles se llaman subárboles del raíz.



✓ Las estructuras tipo árbol se usan principalmente para representar datos con una relación jerárquica entre sus elementos, como árboles genealógicos, tablas, etc.

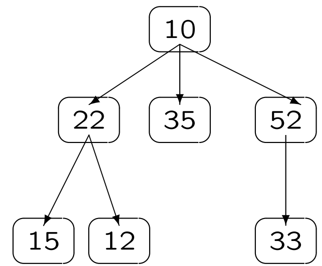
✓ La terminología de los árboles se realiza con las típicas notaciones de las relaciones familiares en los árboles genealógicos: padre, hijo, hermano, ascendente, descendente, etc.

Algunas definiciones

- ✓ Nodo, son los elementos del árbol.
- ✓ Raíz del árbol: todos los árboles que no están vacíos tienen un único nodo raíz. Todos los demás elementos o nodos se derivan o descienden de él.
- ✓ Nodo hoja es aquel nodo que no contiene ningún subárbol.
- ✓ Tamaño de un árbol es su número de nodos.
- ✓ A cada nodo que no es hoja se le asocia uno o varios subárboles llamados descendientes o hijos.
- ✓ De igual forma, cada nodo tiene asociado un antecesor o ascendiente llamado padre.
- ✓ Todos los nodos tienen un solo padre excepto el raíz que no tiene padre.
- ✓ Cada nodo tiene asociado un número de nivel que se determina por la longitud del camino desde el raíz al nodo específico.
- ✓ La altura o profundidad de un árbol es el nivel más profundo más uno.

Ejemplo

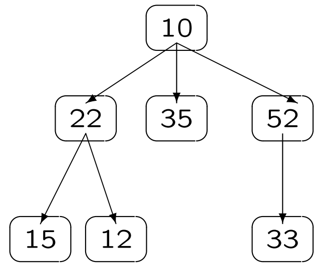
✓ Para ilustrar las definiciones se considera el siguiente árbol general:



✓ Por ejemplo, en la figura:

- Raíz: 10
- Nodos: 10, 22, 35, 52, 15, 12, 33
- Tamaño: 7
- Nivel 0: 10
- Nivel 1: 22, 35, 52
- Nivel 2: 15, 12, 33
- Altura o profundidad: 3
- Hojas: 15, 12, 35, 33

Representación en Haskell

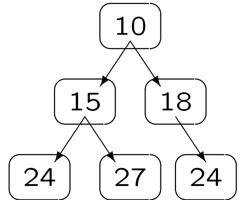


$$\textbf{data}\ \acute{A}rbol\ a = Vac\acute{\imath}o \mid Nodo\ \underbrace{a}_{ra\acute{\imath}z}\ \underbrace{[\acute{A}rbol\ a]}_{hijos}\ \textbf{deriving}\ Show$$

```
a1 :: Árbol Integer
a1 = Nodo 10 [a11, a12, a13]
  where
    a11 = Nodo 22 [hoja 15, hoja 12]
    a12 = hoja 35
    a13 = Nodo 52 [hoja 33]

hoja   :: a → Árbol a
hoja x = Nodo x [ ]

raíz            :: Árbol a → a
raíz Vacío      = error "raíz de árbol vacío"
raíz (Nodo x _) = x

tamaño             :: Árbol a → Integer
tamaño Vacío       = 0
tamaño (Nodo _ xs) = 1 + sum (map tamaño xs)

profundidad             :: Árbol a → Integer
profundidad Vacío       = 0
profundidad (Nodo _ []) = 1
profundidad (Nodo _ xs) = 1 + maximum (map profundidad xs)
```

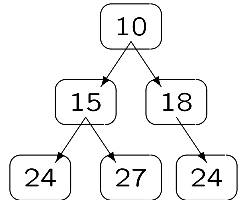
9.2 Árboles binarios

Un árbol binario es árbol tal que cada nodo tiene como máximo dos subárboles.



$$\textbf{data}\ \ \mathit{\acute{A}rbolB}\ a = \mathit{Vac\acute{\imath}oB} \mid \mathit{NodoB}\ \overbrace{(\mathit{\acute{A}rbolB}\ a)}^{hijo\ izq}\ \overbrace{a}^{dato\ en\ nodo}\ \overbrace{(\mathit{\acute{A}rbolB}\ a)}^{hijo\ der}\ \ \textbf{deriving}\ \mathit{Show}$$

Consideraremos que las tres componentes del constructor *NodoB* son el subárbol izquierdo, el dato raíz y el subárbol derecho respectivamente.




```
a2 :: ÁrbolB Integer
a2 = NodoB aI 10 aD
  where
    aI  = NodoB aII 15 aID
    aD  = NodoB VacíoB 18 aDD
    aII = hojaB 24
    aID = hojaB 27
    aDD = hojaB 24

hojaB   :: a → ÁrbolB a
hojaB x =  NodoB VacíoB x VacíoB
```

Árboles binarios (II)

$$
\begin{array}{lcl}
raízB & :: & ÁrbolB\ a \rightarrow a \\
raízB\ VacíoB & = & error\ \text{"raíz de árbol vacío"} \\
raízB\ (NodoB\ _\ x\ _) & = & x
\end{array}
$$

$$
\begin{array}{lcl}
tamañoB & :: & ÁrbolB\ a \rightarrow Integer \\
tamañoB\ VacíoB & = & 0 \\
tamañoB\ (NodoB\ i\ _\ d) & = & 1 + tamañoB\ i + tamañoB\ d
\end{array}
$$

$$
\begin{array}{lcl}
profundidadB & :: & ÁrbolB\ a \rightarrow Integer \\
profundidadB\ VacíoB & = & 0 \\
profundidadB\ (NodoB\ i\ _\ d) & = & 1 + max\ (profundidadB\ i)\ (profundidadB\ d)
\end{array}
$$

Recorrido de árboles binarios (I)

- ✓ Se llama recorrido de un árbol al proceso que permite acceder una sola vez a cada uno de los nodos del árbol para examinar el conjunto completo de nodos.

- ✓ Los algoritmos de recorrido de un árbol binario presentan tres tipos de actividades comunes:
  - ◇ visitar el nodo raíz
  - ◇ recorrer el subárbol izquierdo
  - ◇ recorrer el subárbol derecho

- ✓ Estas tres acciones llevadas a cabo en distinto orden proporcionan los distintos recorridos del árbol.

- ✓ Recorrido en PRE-ORDEN:
  - ◇ Visitar el raíz
  - ◇ Recorrer el subárbol izquierdo en pre-orden
  - ◇ Recorrer el subárbol derecho en pre-orden

- ✓ Recorrido EN-ORDEN
  - ◇ Recorrer el subárbol izquierdo en en-orden
  - ◇ Visitar el raíz
  - ◇ Recorrer el subárbol derecho en en-orden

- ✓ Recorrido en POST-ORDEN
  - ◇ Recorrer el subárbol izquierdo en post-orden
  - ◇ Recorrer el subárbol derecho en post-orden
  - ◇ Visitar el raíz

Recorrido de árboles binarios (II)

$$
\begin{array}{lcl}
enOrdenB & :: & ÁrbolB\ a \rightarrow [a] \\
enOrdenB\ VacíoB & = & [\,] \\
enOrdenB\ (NodoB\ i\ r\ d) & = & enOrdenB\ i\ +\!\!+\ [r]\ +\!\!+\ enOrdenB\ d
\end{array}
$$

$$
\begin{array}{lcl}
preOrdenB & :: & ÁrbolB\ a \rightarrow [a] \\
preOrdenB\ VacíoB & = & [\,] \\
preOrdenB\ (NodoB\ i\ r\ d) & = & [r]\ +\!\!+\ preOrdenB\ i\ +\!\!+\ preOrdenB\ d
\end{array}
$$

$$
\begin{array}{lcl}
postOrdenB & :: & ÁrbolB\ a \rightarrow [a] \\
postOrdenB\ VacíoB & = & [\,] \\
postOrdenB\ (NodoB\ i\ r\ d) & = & postOrdenB\ i\ +\!\!+\ postOrdenB\ d\ +\!\!+\ [r]
\end{array}
$$

? $enOrdenB\ a2$
$[24, 15, 27, 10, 18, 24]$ $::$ $[Integer]$

? $preOrdenB\ a2$
$[10, 15, 24, 27, 18, 24]$ $::$ $[Integer]$

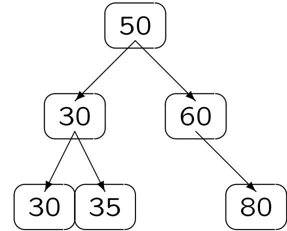
? $postOrdenB\ a2$
$[24, 27, 15, 24, 18, 10]$ $::$ $[Integer]$

9.3 Árboles de búsqueda

✓ Un árbol de búsqueda es un árbol binario tal que

- ◇ O bien es vacío
- ◇ O no es vacío y para cualquier nodo se cumple que:
  - ∘ los elementos del subárbol izquierdo son menores o iguales al almacenado en el nodo
  - ∘ y los elementos del subárbol derecho son estrictamente mayores al almacenado en el nodo

Ejemplo



✓ La siguiente función puede ser utilizada para comprobar si un árbol binario es de búsqueda:

$$
\begin{array}{lll}
es\acute{A}rbolBB & :: & Ord\ a \Rightarrow \acute{A}rbolB\ a \rightarrow Bool \\
es\acute{A}rbolBB\ Vac\acute{i}oB & = & True \\
es\acute{A}rbolBB\ (NodoB\ i\ r\ d) & = & todos\acute{A}rbolB\ (<=\ r)\ i \\
 & & \&\&\ todos\acute{A}rbolB\ (>\ r)\ d \\
 & & \&\&\ es\acute{A}rbolBB\ i \\
 & & \&\&\ es\acute{A}rbolBB\ d
\end{array}
$$

$$
\begin{array}{lll}
todos\acute{A}rbolB & :: & (a \rightarrow Bool) \rightarrow \acute{A}rbolB\ a \rightarrow Bool \\
todos\acute{A}rbolB\ p\ Vac\acute{i}oB & = & True \\
todos\acute{A}rbolB\ p\ (NodoB\ i\ r\ d) & = & p\ r\ \&\& \\
 & & todos\acute{A}rbolB\ p\ i\ \&\&\ todos\acute{A}rbolB\ p\ d
\end{array}
$$

Árboles de búsqueda (2)

✓ Pertenencia a un árbol de búsqueda

$$
\begin{array}{lcl}
\mathit{perteneceBB} & :: & \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{ÁrbolB}\ a \rightarrow \mathit{Bool} \\
\mathit{perteneceBB}\ x\ \mathit{VacíoB} & = & \mathit{False} \\
\mathit{perteneceBB}\ x\ (\mathit{NodoB}\ i\ r\ d) & & \\
\quad \mid\ x == r & = & \mathit{True} \\
\quad \mid\ x < r & = & \mathit{perteneceBB}\ x\ i \\
\quad \mid\ \mathit{otherwise} & = & \mathit{perteneceBB}\ x\ d
\end{array}
$$

✓ Inserción en un árbol de búsqueda

$$
\begin{array}{lcl}
\mathit{insertarBB} & :: & \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{ÁrbolB}\ a \rightarrow \mathit{ÁrbolB}\ a \\
\mathit{insertarBB}\ x\ \mathit{VacíoB} & = & \mathit{NodoB}\ \mathit{VacíoB}\ x\ \mathit{VacíoB} \\
\mathit{insertarBB}\ x\ (\mathit{NodoB}\ i\ r\ d) & & \\
\quad \mid\ x <= r & = & \mathit{NodoB}\ (\mathit{insertarBB}\ x\ i)\ r\ d \\
\quad \mid\ \mathit{otherwise} & = & \mathit{NodoB}\ i\ r\ (\mathit{insertarBB}\ x\ d)
\end{array}
$$

✓ Construcción de un árbol de búsqueda a partir de una lista

$$
\begin{array}{lcl}
\mathit{listaAÁrbolBB} & :: & \mathit{Ord}\ a \Rightarrow [a] \rightarrow \mathit{ÁrbolB}\ a \\
\mathit{listaAÁrbolBB} & = & \mathit{foldr}\ \mathit{insertarBB}\ \mathit{VacíoB}
\end{array}
$$

✓ El recorrido en orden genera una lista ordenada (*tree sort*)

$$
\begin{array}{lcl}
\mathit{treeSort} & :: & \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a] \\
\mathit{treeSort} & = & \mathit{enOrdenB}\ .\ \mathit{listaAÁrbolBB}
\end{array}
$$

? $\mathit{treeSort}\ [4, 7, 1, 2, 9]$
$[1, 2, 4, 7, 9]\ ::\ [\mathit{Integer}]$

Objetivos del tema

El alumno debe:

- ✓ Conocer el concepto de árbol y la terminología asociada
- ✓ Conocer las definiciones de tipo para representar árboles en Haskell
- ✓ Saber definir funciones sobre árboles
- ✓ Conocer la implementación de los árboles de búsqueda en Haskell

Tema 10. Razonamiento ecuacional

10.1 Pruebas directas

10.2 Pruebas por casos

10.3 Pruebas por inducción

10.1 Pruebas directas

✓ El razonamiento formal con programas funcionales es simple

✓ Gracias a la *transparencia referencial* podemos sustituir términos *equivalentes*

✓ Consideraremos las ecuaciones en una definición de función como equivalencias (*Axiomas*)

EJEMPLO

$$\begin{array}{lcl} cambio & :: & (a, b) \rightarrow (b, a) \\ cambio\ (x, y) & = & (y, x) \end{array}$$

Demostrar que

$$\forall\ m::a,\ n::b\ .\ cambio\ (cambio\ (m, n)) \equiv (m, n)$$

Axiomas

$$\forall\ x::a,\ \forall\ y::b\ .\ cambio\ (x, y) \equiv (y, x) \quad -- \ Ax_1$$

Demostración

$$\begin{array}{ll} & cambio\ (\underline{cambio\ (m, n)}) \\ \equiv & \quad \{\text{por } Ax_1\} \\ & \underline{cambio\ (n, m)} \\ \equiv & \quad \{\text{por } Ax_1\} \\ & (m, n) \end{array}$$

10.2 Pruebas por casos

✓ Para tipos no recursivos, basta con probar la propiedad para cada constructor

EJEMPLO

$$\textbf{data}\ Bool = False \mid True$$

$$\begin{aligned} not &:: Bool \to Bool \\ not\ True &= False \\ not\ False &= True \end{aligned}$$

Demostrar que

$$\forall\, x{::}Bool\ .\ not\ (not\ x) \equiv x$$

Axiomas

$$\begin{aligned} not\ True &\equiv False \quad \text{-- } Ax_1 \\ not\ False &\equiv True \quad \text{-- } Ax_2 \end{aligned}$$

Demostración

✓ Si x es $False$, hay que demostrar

$$not\ (not\ False) \equiv False$$

$$\begin{aligned} & not\ (\underline{not\ False}) \\ \equiv &\quad \{\text{por } Ax_2\} \\ & \underline{not\ True} \\ \equiv &\quad \{\text{por } Ax_1\} \\ & False \end{aligned}$$

✓ Si x es $True$, hay que demostrar

$$not\ (not\ True) \equiv True$$

$$\begin{aligned} & not\ (\underline{not\ True}) \\ \equiv &\quad \{\text{por } Ax_1\} \\ & \underline{not\ False} \\ \equiv &\quad \{\text{por } Ax_2\} \\ & True \end{aligned}$$

✓ La propiedad queda demostrada para cualquier $x{::}Bool$

10.3 Pruebas por inducción

✓ Para tipos recursivos, el número de casos a considerar es infinito

✓ No podemos demostrar todos los casos

✓ Se usa la *inducción*

EJEMPLO

$\textbf{data}\ Nat\ =\ Cero\ |\ Suc\ Nat\ \ \textbf{deriving}\ Show$

$$
\begin{array}{lll}
(<\!+\!>) & :: & Nat \rightarrow Nat \rightarrow Nat \\
m\ <\!+\!>\ Cero & = & m \\
m\ <\!+\!>\ Suc\ n & = & Suc\ (m\ <\!+\!>\ n)
\end{array}
$$

Demostrar que

$$\forall\ x::Nat\ .\ Cero\ <\!+\!>\ x\ \equiv\ x$$

Principio de inducción para el tipo Nat

$$\forall x :: Nat\ .\ P(x) \Leftrightarrow \left\{ \begin{array}{l} P(Cero) \\ \wedge \\ \forall x :: Nat\ .\ P(x) \Rightarrow P(Suc\ x) \end{array} \right.$$

✓ **Caso base:** Hay que demostrar $P(Cero)$

✓ **Paso inductivo:** Hay que demostrar $P(Suc\ x)$ supuesto $P(x)$

Pruebas por inducción (2)

Propiedad

$$\forall\, x::Nat \,.\; Cero \;<+>\; x \;\equiv\; x$$

Axiomas

$$\forall\, m::Nat \;.\; m \;<+>\; Cero \;\equiv\; m \qquad \text{-- } Ax_1$$
$$\forall\, m,\; n::Nat \;.\; m \;<+>\; Suc\; n \;\equiv\; Suc\;(m \;<+>\; n) \qquad \text{-- } Ax_2$$

✓ **Caso base:** Hay que demostrar

$$Cero \;<+>\; Cero \;\equiv\; Cero$$

Demostración

$$\underline{Cero \;<+>\; Cero}$$
$\equiv$ {por Ax_1}
$$Cero$$

✓ **Paso inductivo:** Hay que demostrar

$$\forall\, x::Nat \;.\; \underbrace{(Cero \;<+>\; x \;\equiv\; x)}_{Hipótesis\ de\ Inducción} \Rightarrow (Cero \;<+>\; Suc\; x \;\equiv\; Suc\; x)$$

Demostración

$$\underline{Cero \;<+>\; Suc\; x}$$
$\equiv$ {por Ax_2}
$$Suc\;(\underline{Cero \;<+>\; x})$$
$\equiv$ {por hipótesis de inducción}
$$Suc\; x$$

✓ La propiedad queda demostrada para cualquier $x::Nat$

Pruebas por inducción (3)

✓ Las listas también son un tipo recursivo

$$\textbf{data}\ [a] = [\,] \mid a : [a]$$

Principio de inducción para listas

$$\forall ls :: [a]\ .\ P(ls) \Leftrightarrow \begin{cases} P([\,]) \\ \wedge \\ \forall xs :: [a], \forall x :: a\ .\ \ P(xs) \Rightarrow P(x : xs) \end{cases}$$

✓ **Caso base:** Hay que demostrar $P([\,])$

✓ **Paso inductivo:** Hay que demostrar $P(x : xs)$ supuesto $P(xs)$

EJEMPLO

$$\begin{array}{lcl} suma & :: & [Int] \rightarrow Int \\ suma\ [\,] & = & 0 \\ suma\ (x : xs) & = & x + suma\ xs \end{array}$$

$$\begin{array}{lcl} doble & :: & [Int] \rightarrow [Int] \\ doble\ [\,] & = & [\,] \\ doble\ (x : xs) & = & 2 * x\ :\ doble\ xs \end{array}$$

Demostrar

$$\forall\ ls::[Int]\ .\ suma\ (doble\ ls) \equiv 2 * suma\ ls$$

✓ **Caso base:** Hay que demostrar $suma\ (doble\ [\,]) \equiv 2 * suma\ [\,]$

✓ **Paso inductivo:** Hay que demostrar

$$\begin{array}{l} \forall\ xs::[Int],\ \forall\ x::Int\ . \\ \quad suma\ (doble\ xs) \equiv 2 * suma\ xs \qquad \text{-- Hipótesis de inducción} \\ \Rightarrow \\ \quad suma\ (doble\ (x : xs)) \equiv 2 * suma\ (x : xs) \end{array}$$

Pruebas por inducción (4)

Axiomas

$$\begin{array}{rlll} & suma\ [\,] & \equiv\ 0 & \text{-- } AxSuma_1 \\ \forall\ x::Int,\ \forall\ xs::[Int]\ .\ & suma\ (x:xs) & \equiv\ x + suma\ xs & \text{-- } AxSuma_2 \end{array}$$

$$\begin{array}{rlll} & doble\ [\,] & \equiv\ [\,] & \text{-- } AxDoble_1 \\ \forall\ x::Int,\ \forall\ xs::[Int]\ .\ & doble\ (x:xs) & \equiv\ 2*x\ :\ doble\ xs & \text{-- } AxDoble_2 \end{array}$$

✓ **Caso base:** Hay que demostrar

$$suma\ (doble\ [\,])\ \equiv\ 2\ *\ suma\ [\,]$$

Demostración

$suma\ (doble\ [\,])$
$\equiv$ {por $AxDoble_1$}
$suma\ [\,]$
$\equiv$ {por $AxSuma_1$}
0

$2\ *\ suma\ [\,]$
$\equiv$ {por $AxSuma_1$}
$2\ *\ 0$
$\equiv$ {aritmética}
0

✓ **Paso inductivo:** Hay que demostrar

$\forall\ xs::[Int],\ \forall\ x::Int\ .$
$\quad suma\ (doble\ xs)\ \equiv\ 2\ *\ suma\ xs$ -- Hipótesis de inducción
$\quad \Rightarrow$
$\quad suma\ (doble\ (x:xs))\ \equiv\ 2\ *\ suma\ (x:xs)$

Demostración

$suma\ (doble\ (x:xs))$
$\equiv$ {por $AxDoble_2$}
$suma\ (2*x\ :\ doble\ xs)$
$\equiv$ {por $AxSuma_2$}
$2*x\ +\ suma\ (doble\ xs)$
$\equiv$ {por hipótesis de inducción}
$2*x\ +\ 2*suma\ xs$

$2\ *\ suma\ (x:xs)$
$\equiv$ {por $AxSuma_2$}
$2\ *\ (x\ +\ suma\ xs)$
$\equiv$ {distributiva de $(*)$ y $(+)$}
$2*x\ +\ 2*suma\ xs$

Pruebas por inducción (5)

✓ Los árboles binarios son un tipo recursivo

$$\textbf{data}\ \ \textit{ÁrbolB}\ a\ =\ \ \textit{VacíoB} \mid \textit{NodoB}\ (\textit{ÁrbolB}\ a)\ a\ (\textit{ÁrbolB}\ a)$$

Principio de inducción para el tipo *ÁrbolB a*

$$\forall t :: \textit{ÁrbolB}\ a\ .\ P(t) \Leftrightarrow \begin{cases} P(\textit{VacíoB}) \\ \wedge \\ \forall i, d :: \textit{ÁrbolB}\ a,\ \forall r :: a\ . \\ \quad P(i) \wedge P(d) \Rightarrow P(\textit{NodoB}\ i\ r\ d) \end{cases}$$

✓ **Caso base:** Hay que demostrar $P(\textit{VacíoB})$

✓ **Paso inductivo:** Hay que demostrar $P(\textit{NodoB}\ i\ r\ d)$ supuestos $P(i)$ y $P(d)$

✓ Los árboles generales son un tipo recursivo

$$\textbf{data}\ \textit{Árbol}\ a\ =\ \ \textit{Vacío} \mid \textit{Nodo}\ a\ [\textit{Árbol}\ a]$$

Principio de inducción para el tipo *Árbol a*

$$\forall t :: \textit{Árbol}\ a\ .\ P(t) \Leftrightarrow \begin{cases} P(\textit{Vacío}) \\ \wedge \\ \forall xs :: [\textit{Árbol}\ a],\ \forall r :: a\ . \\ \quad \forall x \in xs\ .\ P(x) \Rightarrow P(\textit{Nodo}\ r\ xs) \end{cases}$$

✓ **Caso base:** Hay que demostrar $P(\textit{Vacío})$

✓ **Paso inductivo:** Hay que demostrar $P(\textit{Nodo}\ r\ xs)$ supuesto $P(x)$ para todo x perteneciente a xs

Propiedades con varias variables

✓ Si en la propiedad aparecen varias variables, se puede hacer la inducción sobre cualquiera de ellas

✓ Si al intentarlo sobre una concreta la demostración se complica, lo intentamos sobre otra

EJEMPLO

$$\begin{array}{lll} \mathit{length} & :: & [a] \rightarrow \mathit{Int} \\ \mathit{length}\ [\,] & = & 0 \\ \mathit{length}\ (x : xs) & = & 1 + \mathit{length}\ xs \end{array}$$

$$\begin{array}{llll} (+\!\!+) & & :: & [a] \rightarrow [a] \rightarrow [a] \\ [\,] & +\!\!+\ ys & = & ys \\ (x : xs) & +\!\!+\ ys & = & x : (xs +\!\!+ ys) \end{array}$$

Demostrar:

$$\forall\ xs,\ ys :: [a]\ .\ \mathit{length}\ (xs +\!\!+ ys) \equiv \mathit{length}\ xs + \mathit{length}\ ys$$

Axiomas:

$$\begin{array}{lllll} & \mathit{length}\ [\,] & \equiv & 0 & \text{-- } \mathit{AxLength}_1 \\ \forall\ x :: a,\ \forall\ xs :: [a]\ . & \mathit{length}\ (x : xs) & \equiv & 1 + \mathit{length}\ xs & \text{-- } \mathit{AxLength}_2 \end{array}$$

$$\begin{array}{lllll} \forall\ ys :: [a]\ . & [\,]\ +\!\!+\ ys & \equiv & ys & \text{-- } \mathit{AxConcat}_1 \\ \forall\ x :: a,\ \forall\ xs :: [a],\ \forall\ ys :: [a]\ . & (x : xs)\ +\!\!+\ ys & \equiv & x : (xs +\!\!+ ys) & \text{-- } \mathit{AxConcat}_2 \end{array}$$

Propiedades con varias variables (2)

Propiedad

$$\forall\ xs,\ ys :: [a]\ .\ length\ (xs +\!\!+\ ys) \equiv length\ xs + length\ ys$$

- ✓ Por inducción sobre xs
- ✓ **Caso base:** Hay que demostrar

$$\forall\ ys :: [a]\ .\ length\ ([\,] +\!\!+\ ys) \equiv length\ [\,] + length\ ys$$

- ✓ **Paso inductivo:** Hay que demostrar

$$\begin{array}{l} \forall\ xs :: [a],\ \forall\ x :: a\ . \\ \quad \forall\ ys :: [a]\ .\ length\ (xs +\!\!+\ ys) \equiv length\ xs + length\ ys \\ \Rightarrow \\ \quad \forall\ ys :: [a]\ .\ length\ ((x : xs) +\!\!+\ ys) \equiv length\ (x : xs) + length\ ys \end{array}$$

- ✓ Por inducción sobre ys
- ✓ **Caso base:** Hay que demostrar

$$\forall\ xs :: [a]\ .\ length\ (xs +\!\!+\ [\,]) \equiv length\ xs + length\ [\,]$$

- ✓ **Paso inductivo:** Hay que demostrar

$$\begin{array}{l} \forall\ ys :: [a],\ \forall\ y :: a\ . \\ \quad \forall\ xs :: [a]\ .\ length\ (xs +\!\!+\ ys) \equiv length\ xs + length\ ys \\ \Rightarrow \\ \quad \forall\ xs :: [a]\ .\ length\ (xs +\!\!+\ (y : ys)) \equiv length\ xs + length\ (y : ys) \end{array}$$

Objetivos del tema

El alumno debe:

- ✓ Conocer cómo razonar formalmente con programas funcionales
- ✓ Conocer cómo razonar con tipos no recursivos
- ✓ Conocer cómo razonar con tipos recursivos (principios de inducción)
- ✓ Saber demostrar propiedades usando los métodos anteriores