

Tema 1: Introducción a la programación funcional

Informática (2013–14)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 1: Introducción a la programación funcional

1. Funciones
2. Programación funcional
3. Rasgos característicos de Haskell
4. Antecedentes históricos
5. Presentación de Haskell

Funciones en Haskell

- ▶ En Haskell, una **función** es una **aplicación** que toma uno o más **argumentos** y devuelve un **valor**.
- ▶ En Haskell, las funciones se definen mediante **ecuaciones** formadas por el **nombre de la función**, los **nombres de los argumentos** y el **cuerpo** que especifica cómo se calcula el valor a partir de los argumentos.
- ▶ Ejemplo de definición de función en Haskell:

```
doble x = x + x
```

- ▶ Ejemplo de evaluación:

```
doble 3
= 3 + 3    [def. de doble]
= 6        [def. de +]
```

Funciones en Haskell

- ▶ En Haskell, una **función** es una **aplicación** que toma uno o más **argumentos** y devuelve un **valor**.
- ▶ En Haskell, las funciones se definen mediante **ecuaciones** formadas por el **nombre de la función**, los **nombres de los argumentos** y el **cuerpo** que especifica cómo se calcula el valor a partir de los argumentos.
- ▶ Ejemplo de definición de función en Haskell:

```
doble x = x + x
```

- ▶ Ejemplo de evaluación:

```
doble 3
= 3 + 3    [def. de doble]
= 6        [def. de +]
```

Evaluaciones de funciones en Haskell

- ▶ Ejemplo de evaluación anidada impaciente:

```
double (double 3)
= double (3 + 3)    [def. de double]
= double 6         [def. de +]
= 6 + 6           [def. de double]
= 12              [def. de +]
```

- ▶ Ejemplo de evaluación anidada perezosa:

```
double (double 3)
= (double 3) + (double 3) [def. de double]
= (3 + 3) + (double 3)   [def. de double]
= 6 + (double 3)        [def. de +]
= 6 + (3 + 3)           [def. de double]
= 6 + 6                 [def. de +]
= 12                    [def. de +]
```

Evaluaciones de funciones en Haskell

- ▶ Ejemplo de evaluación anidada impaciente:

```
doble (doble 3)
= doble (3 + 3)   [def. de doble]
= doble 6         [def. de +]
= 6 + 6          [def. de doble]
= 12             [def. de +]
```

- ▶ Ejemplo de evaluación anidada perezosa:

```
doble (doble 3)
= (doble 3) + (doble 3) [def. de doble]
= (3 +3) + (doble 3)   [def. de doble]
= 6 + (doble 3)        [def. de +]
= 6 + (3 + 3)          [def. de doble]
= 6 + 6                [def. de +]
= 12                   [def. de +]
```

Comprobación de propiedades

- ▶ Propiedad: El doble de x más y es el doble de x más el doble de y
- ▶ Expresión de la propiedad:

```
prop_doble x y = doble (x+y) == (doble x) + (doble y)
```

- ▶ Comprobación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_doble  
+++ OK, passed 100 tests.
```

- ▶ Para usar QuickCheck hay que importarlo, escribiendo al principio del fichero

```
import Test.QuickCheck
```

Comprobación de propiedades

- ▶ Propiedad: El doble de x más y es el doble de x más el doble de y
- ▶ Expresión de la propiedad:

```
prop_doble x y = doble (x+y) == (doble x) + (doble y)
```

- ▶ Comprobación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_doble  
+++ OK, passed 100 tests.
```

- ▶ Para usar QuickCheck hay que importarlo, escribiendo al principio del fichero

```
import Test.QuickCheck
```

Comprobación de propiedades

- ▶ Propiedad: El doble de x más y es el doble de x más el doble de y
- ▶ Expresión de la propiedad:

```
prop_doble x y = doble (x+y) == (doble x) + (doble y)
```

- ▶ Comprobación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_doble  
+++ OK, passed 100 tests.
```

- ▶ Para usar QuickCheck hay que importarlo, escribiendo al principio del fichero

```
import Test.QuickCheck
```

Comprobación de propiedades

- ▶ Propiedad: El doble de x más y es el doble de x más el doble de y
- ▶ Expresión de la propiedad:

```
prop_doble x y = doble (x+y) == (doble x) + (doble y)
```

- ▶ Comprobación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_doble  
+++ OK, passed 100 tests.
```

- ▶ Para usar QuickCheck hay que importarlo, escribiendo al principio del fichero

```
import Test.QuickCheck
```

Refutación de propiedades

- ▶ Propiedad: El producto de dos números cualquiera es distinto de su suma.
- ▶ Expresión de la propiedad:

```
prop_prod_suma x y = x*y /= x+y
```

- ▶ Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma
*** Failed! Falsifiable (after 1 test):
0
0
```

Refutación de propiedades

- ▶ Propiedad: El producto de dos números cualesquiera es distinto de su suma.
- ▶ Expresión de la propiedad:

```
prop_prod_suma x y = x*y /= x+y
```

- ▶ Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma
*** Failed! Falsifiable (after 1 test):
0
0
```

Refutación de propiedades

- ▶ Propiedad: El producto de dos números cualesquiera es distinto de su suma.
- ▶ Expresión de la propiedad:

```
prop_prod_suma x y = x*y /= x+y
```

- ▶ Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma  
*** Failed! Falsifiable (after 1 test):  
0  
0
```

Refutación de propiedades

- ▶ Refinamiento: El producto de dos números no nulos cualesquiera es distinto de su suma.

```
prop_prod_suma' x y =  
  x /= 0 && y /= 0 ==> x*y /= x+y
```

- ▶ Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma'  
+++ OK, passed 100 tests.  
*Main> quickCheck prop_prod_suma'  
*** Failed! Falsifiable (after 5 tests):  
2  
2
```

Refutación de propiedades

- ▶ Refinamiento: El producto de dos números no nulos cualesquiera es distinto de su suma.

```
prop_prod_suma' x y =  
  x /= 0 && y /= 0 ==> x*y /= x+y
```

- ▶ Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma'  
+++ OK, passed 100 tests.  
*Main> quickCheck prop_prod_suma'  
*** Failed! Falsifiable (after 5 tests):  
2  
2
```

Refutación de propiedades

- ▶ Refinamiento: El producto de dos números no nulos cualesquiera es distinto de su suma.

```
prop_prod_suma' x y =  
  x /= 0 && y /= 0 ==> x*y /= x+y
```

- ▶ Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma'  
+++ OK, passed 100 tests.  
*Main> quickCheck prop_prod_suma'  
*** Failed! Falsifiable (after 5 tests):  
2  
2
```

Programación funcional y programación imperativa

- ▶ La **programación funcional** es un estilo de programación cuyo método básico de computación es la aplicación de funciones a sus argumentos.
- ▶ Un **lenguaje de programación funcional** es uno que soporta y potencia el estilo funcional.
- ▶ La **programación imperativa** es un estilo de programación en el que los programas están formados por instrucciones que especifican cómo se ha de calcular el resultado.
- ▶ Ejemplo de problema para diferenciar los estilos de programación: Sumar los n primeros números.

Programación funcional y programación imperativa

- ▶ La **programación funcional** es un estilo de programación cuyo método básico de computación es la aplicación de funciones a sus argumentos.
- ▶ Un **lenguaje de programación funcional** es uno que soporta y potencia el estilo funcional.
- ▶ La **programación imperativa** es un estilo de programación en el que los programas están formados por instrucciones que especifican cómo se ha de calcular el resultado.
- ▶ Ejemplo de problema para diferenciar los estilos de programación: Sumar los n primeros números.

Programación funcional y programación imperativa

- ▶ La **programación funcional** es un estilo de programación cuyo método básico de computación es la aplicación de funciones a sus argumentos.
- ▶ Un **lenguaje de programación funcional** es uno que soporta y potencia el estilo funcional.
- ▶ La **programación imperativa** es un estilo de programación en el que los programas están formados por instrucciones que especifican cómo se ha de calcular el resultado.
- ▶ Ejemplo de problema para diferenciar los estilos de programación: Sumar los n primeros números.

Programación funcional y programación imperativa

- ▶ La **programación funcional** es un estilo de programación cuyo método básico de computación es la aplicación de funciones a sus argumentos.
- ▶ Un **lenguaje de programación funcional** es uno que soporta y potencia el estilo funcional.
- ▶ La **programación imperativa** es un estilo de programación en el que los programas están formados por instrucciones que especifican cómo se ha de calcular el resultado.
- ▶ Ejemplo de problema para diferenciar los estilos de programación: Sumar los n primeros números.

Solución mediante programación imperativa

- ▶ Programa *suma n*:

contador := 0

total := 0

repetir

contador := contador + 1

total := total + contador

hasta que contador = n

- ▶ Evaluación de *suma 4*:

contador	total
0	0
1	1
2	3
3	6
4	10

Solución mediante programación imperativa

- ▶ Programa *suma n*:

contador := 0

total := 0

repetir

contador := contador + 1

total := total + contador

hasta que contador = n

- ▶ Evaluación de *suma 4*:

contador	total
0	0
1	1
2	3
3	6
4	10

Solución mediante programación funcional

- ▶ Programa:

```
suma n = sum [1..n]
```

- ▶ Evaluación de *suma 4*:

```
suma 4
= sum [1..4]      [def. de suma]
= sum [1, 2, 3, 4] [def. de [..]]
= 1 + 2 + 3 + 4   [def. de sum]
= 10              [def. de +]
```

Solución mediante programación funcional

- ▶ Programa:

```
suma n = sum [1..n]
```

- ▶ Evaluación de *suma 4*:

```
suma 4
= sum [1..4]      [def. de suma]
= sum [1, 2, 3, 4] [def. de [..]]
= 1 + 2 + 3 + 4   [def. de sum]
= 10              [def. de +]
```

Rasgos característicos de Haskell

- ▶ Programas concisos.
- ▶ Sistema potente de tipos.
- ▶ Listas por comprensión.
- ▶ Funciones recursivas.
- ▶ Funciones de orden superior.
- ▶ Razonamiento sobre programas.
- ▶ Evaluación perezosa.
- ▶ Efectos monádicos.

Antecedentes históricos

- ▶ 1930s: Alonzo Church desarrolla el lambda cálculo (teoría básica de los lenguajes funcionales).
- ▶ 1950s: John McCarthy desarrolla el Lisp (lenguaje funcional con asignaciones).
- ▶ 1960s: Peter Landin desarrolla ISWIN (lenguaje funcional puro).
- ▶ 1970s: John Backus desarrolla FP (lenguaje funcional con orden superior).
- ▶ 1970s: Robin Milner desarrolla ML (lenguaje funcional con tipos polimórficos e inferencia de tipos).
- ▶ 1980s: David Turner desarrolla Miranda (lenguaje funcional perezoso).
- ▶ 1987: Un comité comienza el desarrollo de Haskell.
- ▶ 2003: El comité publica el “Haskell Report”.

Ejemplo de recursión sobre listas

- ▶ Especificación: $(\text{sum } xs)$ es la suma de los elementos de xs .
- ▶ Ejemplo: $\text{sum } [2,3,7] \rightsquigarrow 12$
- ▶ Definición:

```
sum []      = 0
sum (x:xs) = x + sum xs
```

- ▶ Evaluación:

$\text{sum } [2,3,7]$	
$= 2 + \text{sum } [3,7]$	[def. de sum]
$= 2 + (3 + \text{sum } [7])$	[def. de sum]
$= 2 + (3 + (7 + \text{sum } []))$	[def. de sum]
$= 2 + (3 + (7 + 0))$	[def. de sum]
$= 12$	[def. de +]
- ▶ Tipo de sum: $(\text{Num } a) \Rightarrow [a] \rightarrow a$

Ejemplo de recursión sobre listas

- ▶ Especificación: $(\text{sum } xs)$ es la suma de los elementos de xs .
- ▶ Ejemplo: $\text{sum } [2,3,7] \rightsquigarrow 12$
- ▶ Definición:

```
sum [] = 0
```

```
sum (x:xs) = x + sum xs
```

- ▶ Evaluación:

```
sum [2,3,7]
= 2 + sum [3,7]           [def. de sum]
= 2 + (3 + sum [7])       [def. de sum]
= 2 + (3 + (7 + sum []))  [def. de sum]
= 2 + (3 + (7 + 0))       [def. de sum]
= 12                       [def. de +]
```

- ▶ Tipo de sum : $(\text{Num } a) \Rightarrow [a] \rightarrow a$

Ejemplo de recursión sobre listas

- ▶ Especificación: $(\text{sum } xs)$ es la suma de los elementos de xs .
- ▶ Ejemplo: $\text{sum } [2,3,7] \rightsquigarrow 12$
- ▶ Definición:

```
sum [] = 0
```

```
sum (x:xs) = x + sum xs
```

- ▶ Evaluación:

```
sum [2,3,7]
= 2 + sum [3,7]           [def. de sum]
= 2 + (3 + sum [7])       [def. de sum]
= 2 + (3 + (7 + sum []))  [def. de sum]
= 2 + (3 + (7 + 0))       [def. de sum]
= 12                       [def. de +]
```

- ▶ Tipo de sum : $(\text{Num } a) \Rightarrow [a] \rightarrow a$

Ejemplo de recursión sobre listas

- ▶ Especificación: $(\text{sum } xs)$ es la suma de los elementos de xs .
- ▶ Ejemplo: $\text{sum } [2,3,7] \rightsquigarrow 12$
- ▶ Definición:

```
sum [] = 0
```

```
sum (x:xs) = x + sum xs
```

- ▶ Evaluación:

```
sum [2,3,7]
= 2 + sum [3,7]           [def. de sum]
= 2 + (3 + sum [7])       [def. de sum]
= 2 + (3 + (7 + sum []))  [def. de sum]
= 2 + (3 + (7 + 0))       [def. de sum]
= 12                       [def. de +]
```

- ▶ Tipo de sum : $(\text{Num } a) \Rightarrow [a] \rightarrow a$

Ejemplo con listas de comprensión

- ▶ Especificación: `(ordena xs)` es la lista obtenida ordenando `xs` mediante el algoritmo de ordenación rápida.

- ▶ Ejemplo:

```
ordena [4,6,2,5,3] ~> [2,3,4,5,6]
ordena "deacb"    ~> "abcde"
```

- ▶ Definición:

```
ordena [] = []
ordena (x:xs) =
    (ordena menores) ++ [x] ++ (ordena mayores)
  where menores = [a | a <- xs, a <= x]
        mayores = [b | b <- xs, b > x]
```

- ▶ Tipo de `ordena`: `Ord a => [a] -> [a]`

Ejemplo con listas de comprensión

- Especificación: `(ordena xs)` es la lista obtenida ordenando `xs` mediante el algoritmo de ordenación rápida.

- Ejemplo:

```
ordena [4,6,2,5,3] ~> [2,3,4,5,6]
ordena "deacb"    ~> "abcde"
```

- Definición:

```
ordena [] = []
ordena (x:xs) =
    (ordena menores) ++ [x] ++ (ordena mayores)
  where menores = [a | a <- xs, a <= x]
        mayores = [b | b <- xs, b > x]
```

- Tipo de `ordena`: `Ord a => [a] -> [a]`

Ejemplo con listas de comprensión

- Especificación: `(ordena xs)` es la lista obtenida ordenando `xs` mediante el algoritmo de ordenación rápida.

- Ejemplo:

```
ordena [4,6,2,5,3] ~> [2,3,4,5,6]
ordena "deacb"    ~> "abcde"
```

- Definición:

```
ordena [] = []
ordena (x:xs) =
  (ordena menores) ++ [x] ++ (ordena mayores)
  where menores = [a | a <- xs, a <= x]
        mayores = [b | b <- xs, b > x]
```

- Tipo de `ordena`: `Ord a => [a] -> [a]`

Evaluación del ejemplo con listas de comprensión

ordena [4,6,2,3]	
= (ordena [2,3]) ++ [4] ++ (ordena [6])	[def. ordena]
= ((ordena []) ++ [2] ++ (ordena [3])) ++ [4] ++ (ordena [6])	[def. ordena]
= ([] ++ [2] ++ (ordena [3])) ++ [4] ++ (ordena [6])	[def. ordena]
= ([2] ++ (ordena [3])) ++ [4] ++ (ordena [6,5])	[def. ++]
= ([2] ++ ((ordena []) ++ [3] ++ [])) ++ [4] ++ (ordena [6])	[def. ordena]
= ([2] ++ ([] ++ [3] ++ [])) ++ [4] ++ (ordena [6])	[def. ordena]
= ([2] ++ [3]) ++ [4] ++ (ordena [6])	[def. ++]
= [2,3] ++ [4] ++ (ordena [6])	[def. ++]
= [2,3,4] ++ (ordena [6])	[def. ++]
= [2,3,4] ++ ((ordena []) ++ [6] ++ (ordena []))	[def. ordena]
= [2,3,4] ++ ((ordena []) ++ [6] ++ (ordena []))	[def. ordena]
= [2,3,4] ++ ([] ++ [6] ++ [])	[def. ordena]
= [2,3,4,6]	[def. ++]

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
 - ▶ Cap. 1: Conceptos fundamentales.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
 - ▶ Cap. 1: Introduction.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
 - ▶ Cap. 1: Getting Started.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
 - ▶ Cap. 1: Programación funcional.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
 - ▶ Cap. 1: Introducing functional programming.