# Introducción a Haskell

## Y a la programación funcional

Pablo Baeyens
@pbaeyens

Mario Román
@M42

OSL 2015



Introducción a Haskell
Y a la programación funcional
Pablo Baeyens @pbaeyens
Mario Román @M42
OSL 2015

# Índice





Índice

Haskell

Tipos

Funciones

Más

# ¡Contribuye!

El código fuente de estas diapositivas, con varios ejemplos de Haskell literario, está disponible en:

github.com/M42/osl-talk-haskell

Erratas, correcciones y aportaciones son bienvenidas.





¡Contribuye!
El código fuente de estas diapositivas, con varios ejemplos de Haskell literario, está disponible en:
github.com/M42/osl-talk-haskell
Erratas, correcciones y aportaciones son bienvenidas.

**Esta es la versión con notas.**

# Instalando Haskell Platform

`haskell-platform` contiene el compilador, depurador y otras utilidades. También podemos instalar `ghc`:

```
sudo apt-get install haskell-platform
```

Ambos traen un gestor de librerías: `cabal`.



Instalando Haskell Platform

haskell-platform contiene el compilador, depurador y otras utilidades. También podemos instalar ghc:

sudo apt-get install haskell-platform

Ambos traen un gestor de librerías: cabal.

- ghc: (Glorious) Glasgow Haskell Compiler.
- Linter: `hlint`.

# El intérprete: GHCi

GHC incluye GHCi como intérprete. Permite los siguientes comandos:

- `:q` Quitar
- `:l` Cargar módulo
- `:r` Recargar módulos
- `:t` Consultar tipos



El intérprete: GHCi
GHC incluye GHCi como intérprete. Permite los siguientes comandos:
- :q Quitar
- :l Cargar módulo
- :r Recargar módulos
- :t Consultar tipos

Haskell permite operaciones aritméticas básicas, y operaciones con cadenas, listas o booleanos.

- `:set +t` para mostrar el tipo por defecto.
- `:set +m` para permitir entrada de varias líneas.

# El intérprete: GHCi

Las funciones se llaman escribiendo su nombre, un espacio y sus parámetros, separados por espacios:

```
ghci> 3 + 4
7
ghci> (+) 2 9
11
ghci> succ 27
28
ghci> max 23 34
34
```



El intérprete: GHCi

Las funciones se llaman escribiendo su nombre, un espacio y sus parámetros, separados por espacios:

ghci> 3 + 4
7
ghci> (+) 2 9
11
ghci> succ 27
28
ghci> max 23 34
34

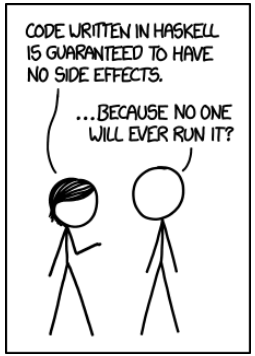
- Estamos usando **notación polaca**.
- Para escribir una función infija de forma prefija se pone entre paréntesis.
- Para escribir una función prefija de forma infija se pone entre acentos graves.

# Puro: sin efectos secundarios

Las funciones no tienen *efectos secundarios*. No alteran el mundo ni cambian el valor de los argumentos.

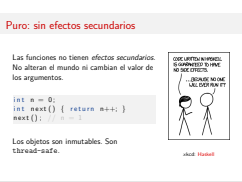
```
int n = 0;
int next() { return n++; }
next(); // n = 1
```

Los objetos son inmutables. Son `thread-safe`.




*x*kcd: Haskell






- Ejemplo de `http://programmers.stackexchange.com/questions/40297`.
- Una función puede cambiar variables o escribir por pantalla. Eso hace que el orden de llamada importe.

# Puro: transparencia referencial

Que las expresiones de Haskell sean referencialmente transparentes quiere decir:

- Todas las variables son **inmutables**.
- Las funciones son **deterministas**.

Y esto nos permite:

- Razonar algebraicamente: `f = g ⇒ f a = g a`.
- Paralelizar fácilmente: `f ‘par‘ g`, sin afectarse.



Puro: transparencia referencial

Que las expresiones de Haskell sean referencialmente transparentes quiere decir:
- Todas las variables son **inmutables**.
- Las funciones son **deterministas**.

Y esto nos permite:
- Razonar algebraicamente: `f = g ⇒ f a = g a`.
- Paralelizar fácilmente: `f ‘par‘ g`, sin afectarse.

- **Inmutables**: no cambian de valor en ejecución.
- **Deterministas**: devuelven lo mismo si se les pasan los mismos argumentos.

# Funcional: evaluación

La programación se centra en **evaluar expresiones** en lugar de **ejecutar instrucciones**.



Funcional: evaluación
La programación se centra en **evaluar expresiones** en lugar de **ejecutar instrucciones**.

# Funcional: las funciones como objetos

Las funciones son objetos de *primera clase*. Pueden ser devueltos por funciones y pueden pasarse como argumentos.

```
duplica lista = map (λ x → 2*x) lista
```

Esto ayuda a reutilizar código.

```
int duplica(int a);
int incrementa(int a);
vector<int> duplica_vector(vector<int> v);
vector<int> incrementa_vector(vector<int> v);
```



Funcional: las funciones como objetos
Las funciones son objetos de *primera clase*. Pueden ser devueltos por funciones y pueden pasarse como argumentos.
duplica lista = map (λ x → 2*x) lista
Esto ayuda a reutilizar código.
int duplica(int a);
int incrementa(int a);
vector<int> duplica_vector(vector<int> v);
vector<int> incrementa_vector(vector<int> v);

Las dos últimas funciones sólo se diferencian en la función que aplican sobre las componentes, eso podría ser un argumento a una función `aplica_vector` más general. Puede notarse la necesidad de separar dos partes:

- La función en sí.
- El aplicarla sobre un vector.

# Funcional: abstracción

El ser funcional facilita factorizar el código. Cada pieza debería aparecer sólo una vez en su forma más general posible. Esto se consigue con:

- **Polimorfismo**, abstraer el tipo.
- **Clases de tipos**, unifican propiedades de varios tipos.
- **Funciones de alto nivel**, abstraen otras funciones.



Funcional: abstracción

El ser funcional facilita factorizar el código. Cada pieza debería aparecer sólo una vez en su forma más general posible. Esto se consigue con:

- **Polimorfismo**, abstraer el tipo.
- **Clases de tipos**, unifican propiedades de varios tipos.
- **Funciones de alto nivel**, abstraen otras funciones.

# Tipos

Usamos `:t` para ver el tipo de una expresión:

```
ghci> :t True
True :: Bool
ghci> :t 'a'
'a' :: Char
ghci> :t "Una string!"
"Una string!" :: [Char]
ghci> :t not
not :: Bool → Bool
```



Tipos

Usamos :t para ver el tipo de una expresión:

ghci> :t True
True :: Bool
ghci> :t 'a'
'a' :: Char
ghci> :t "Una string!"
"Una string!" :: [Char]
ghci> :t not
not :: Bool → Bool

Los tipos se escriben con su primera letra mayúscula. Haskell tiene los tipos básicos ya construidos. Existen `Int`, `Bool`, `Char`, ...

# Tipos

Los tipos de Haskell son **fuertes** y **estáticos**. La mayor parte de los errores se detectan en compilación como errores de tipo.

Además son **inferidos**, por lo que no tenemos por qué especificar el tipo de nuestras funciones:

```
ghci> let nand a b = not (a && b)
ghci> :t nand
nand :: Bool → Bool → Bool
```



Tipos

Los tipos de Haskell son **fuertes** y **estáticos**. La mayor parte de los errores se detectan en compilación como errores de tipo.

Además son **inferidos**, por lo que no tenemos por qué especificar el tipo de nuestras funciones:

```
ghci> let nand a b = not (a && b)
ghci> :t nand
nand :: Bool → Bool → Bool
```

Más errores en compilación y menos en ejecución.
Los tipos nos ayudan a documentar, a estructurar el código y a clarificarlo.

# Clases de tipos

Las clases de tipos agrupan a tipos con la misma interfaz. Por ejemplo, la clase `Eq`, agrupa a los que tienen definida la función `(==)`.

```
2    :: Num a ⇒ a
pi   :: Floating a ⇒ a
(==) :: Eq a ⇒ a → a → Bool
```

Las instancias de `Num` pueden sumarse y multiplicarse, las de `Show` convertirse a `String` y sobre las de `Integral` pueden calcularse restos modulares.



Clases de tipos

Las clases de tipos agrupan a tipos con la misma interfaz. Por ejemplo, la clase Eq, agrupa a los que tienen definida la función (==).

2 :: Num a ⇒ a
pi :: Floating a ⇒ a
(==) :: Eq a ⇒ a → a → Bool

Las instancias de Num pueden sumarse y multiplicarse, las de Show convertirse a String y sobre las de Integral pueden calcularse restos modulares.

# Variables de tipo

Haskell infiere siempre el tipo más general. Para ello usa **variables de tipo**, que pueden ser sustituidas:

```
id  :: a → a
(+) :: Num a ⇒ a → a → a
(<) :: Ord a ⇒ a → a → Bool
```

Las variables pueden restringirse a pertenecer a una clase.



Variables de tipo

Haskell infiere siempre el tipo más general. Para ello usa **variables de tipo**, que pueden ser sustituidas:

```
id  :: a → a
(+) :: Num a ⇒ a → a → a
(<) :: Ord a ⇒ a → a → Bool
```

Las variables pueden restringirse a pertenecer a una clase.

- El espacio de nombres de las variables de tipo es independiente en cada declaración.
- Por convención se utilizan nombres cortos para las variables
- La clase de tipo `Ord` incluye tipos con relaciones de orden definidas.

# Tipos algebraicos

Creamos nuevos tipos definiendo **constructores de datos**: funciones que devuelven valores del tipo que definimos.

```
data () = () -- Tipo None
data Bool = False | True
data Point = Point Float Float
data Triangle = Triangle Point Point Point
```




Los constructores se separan con `|`.
Podemos definir tipos que dependan de otros o tipos con constructores sin argumentos, como `Bool`.

# Constructores de tipos

Los **constructores de tipos** son funciones sobre tipos: toman un tipo y devuelven otro.

```
"Haskell!" :: [Char]
[1,2,3,4] :: Num a ⇒ [a]
[True, False, False] :: [Bool]
[] :: [a]
Just True :: Maybe Bool
Nothing :: Maybe a
```



Constructores de tipos

Los **constructores de tipos** son funciones sobre tipos: toman un tipo y devuelven otro.

```
"Haskell!" :: [Char]
[1,2,3,4] :: Num a ⇒ [a]
[True, False, False] :: [Bool]
[] :: [a]
Just True :: Maybe Bool
Nothing :: Maybe a
```

`[]` construye listas, y `Maybe` construye un tipo que puede tener o no un valor. `->` es el constructor de funciones.

Aunque es posible, no es recomendable aplicar restricciones de clase a los tipos de entrada.

# Constructores de tipos

Sus definiciones son:

```
data [a]      = [] | a:[a]
data Maybe a = Nothing | Just a
```

En las **listas**, el primer constructor es la lista vacía y el segundo antepone un elemento a otra lista.

En el caso de **Maybe** podemos tener algo de tipo a (`Just a`) o nada (`Nothing`).



Constructores de tipos

Sus definiciones son:

```
data [a]      = [] | a:[a]
data Maybe a = Nothing | Just a
```

En las **listas**, el primer constructor es la lista vacía y el segundo antepone un elemento a otra lista.

En el caso de **Maybe** podemos tener algo de tipo a (`Just a`) o nada (`Nothing`).

Las listas son flujos en otros lenguajes. Son puramente funcionales y perezosas.

# Reconocimiento de patrones

Para definir una función sobre un tipo, definimos su comportamiento para cada constructor de datos del tipo:

```
neg :: Bool → Bool
neg False = True
neg True  = False
```

Podemos sustituir argumentos del constructor por variables:

```
factorial :: Integral a ⇒ a → a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```



Reconocimiento de patrones
Para definir una función sobre un tipo, definimos su comportamiento para cada constructor de datos del tipo:
neg :: Bool → Bool
neg False = True
neg True = False
Podemos sustituir argumentos del constructor por variables:
factorial :: Integral a ⇒ a → a
factorial 0 = 1
factorial n = n * factorial (n - 1)

Cada entero es un constructor de tipos de `Int`. Las ecuaciones se evalúan en el orden en que aparecen, cambiando el orden de la segunda definición dejaría de funcionar.

# Recursividad

Para calcular la **longitud de una lista**, definimos la función para sus dos constructores:

```
len :: Num a ⇒ [t] → a
len []     = 0
len (_:xs) = 1 + len xs
```

`len [1,2,3]`:

```
len (1:2:3:[])
1 + len (2:3:[])
1 + 1 + len (3:[])
1 + 1 + 1 + len []
1 + 1 + 1 + 0
1 + 1 + 1
1 + 2
3
```



Recursividad

Para calcular la **longitud de una lista**, definimos la función para sus dos constructores:

```
len :: Num a ⇒ [t] → a
len []     = 0
len (_:xs) = 1 + len xs
```

`len [1,2,3]`:

```
len (1:2:3:[])
1 + len (2:3:[])
1 + 1 + len (3:[])
1 + 1 + 1 + len []
1 + 1 + 1 + 0
1 + 1 + 1
1 + 2
3
```

- El tipo lista y las definiciones recursivas son la base de los programas de Haskell.
- Normalmente `[]` es el caso base y `:` la ecuación recursiva.
- No es la versión más eficiente porque no emplea recursión de cola; necesitaría un parámetro acumulador. Además da lugar a space leak.

- Ponemos el segundo constructor entre paréntesis para distinguir de una función que tomara 3 argumentos: $\_, :, xs$.

# Currificación

¿Por qué `(+)` es de tipo `a -> a -> a` y no `(a,a) -> a`?

Esto nos permite aplicar parcialmente una función. El tipo hay que leerlo realmente como `a -> (a -> a)`, es decir, al darle un número nos devuelve otra función:

```
(+)     :: Num a ⇒ a → a → a
(+3)    :: Num a ⇒ a → a
(+3) 5  :: Num a ⇒ a
```

Es lo mismo decir `(+3) 5` que `(+) 3 5`.



Currificación

¿Por qué (+) es de tipo a -> a -> a y no (a,a) -> a?

Esto nos permite aplicar parcialmente una función. El tipo hay que leerlo realmente como a -> (a -> a), es decir, al darle un número nos devuelve otra función:

(+) :: Num a ⇒ a → a → a
(+3) :: Num a ⇒ a → a
(+3) 5 :: Num a ⇒ a

Es lo mismo decir (+3) 5 que (+) 3 5.

Existen las funciones `curry` y `uncurry`.

# Funciones de orden superior

`map` toma una función y devuelve su versión sobre listas:

```
map :: (a → b) → ([a] → [b])
```

*Ejemplo*: `map not [True, True, False]`

`foldr` toma una función, un acumulador y una lista y aplica los elementos de la lista contra el acumulador.

```
foldr :: (a → b → b) → b → [a] → b
```

*Ejemplo*: `foldr (*) 1 [2,3,5,7]`

▸ Definiciones



Funciones de orden superior
map toma una función y devuelve su versión sobre listas:
map :: (a → b) → ([a] → [b])
Ejemplo: map not [True, True, False]
foldr toma una función, un acumulador y una lista y aplica los elementos de la lista contra el acumulador.
foldr :: (a → b → b) → b → [a] → b
Ejemplo: foldr (*) 1 [2,3,5,7]
Definiciones

- Son funciones que toman funciones como argumento.
- `map` aplica la función a cada elemento de la lista.
- `map f = foldr ((:) . f) []`

# Especialización

A partir de estas podemos crear funciones básicas:

```
negation  = map not
lowerText = map toLower
sum       = foldr (+) 0
product   = foldr (*) 1
concat    = foldr (++) []
and       = foldr (&&) True
```

*Ejemplo*: `lowerText "aBcDEfG"`



Especialización
A partir de estas podemos crear funciones básicas:
negation = map not
lowerText = map toLower
sum = foldr (+) 0
product = foldr (*) 1
concat = foldr (++) []
and = foldr (&&) True
Ejemplo: lowerText "aBcDEfG"

Podemos escribirlas sin todos los argumentos por la currificación.

# Evaluación perezosa

Haskell retrasa la evaluación de una expresión todo lo posible:

```
head [1..10**9]      -- Sólo evalua 1
```

Esto permite la modularización del código:

```
min :: Ord a => [a] -> a
min = head . sort
```

Y el uso de estructuras infinitas:

```
unos      = 1:unos
diezDoses = take 10 (map (+1) unos)
```

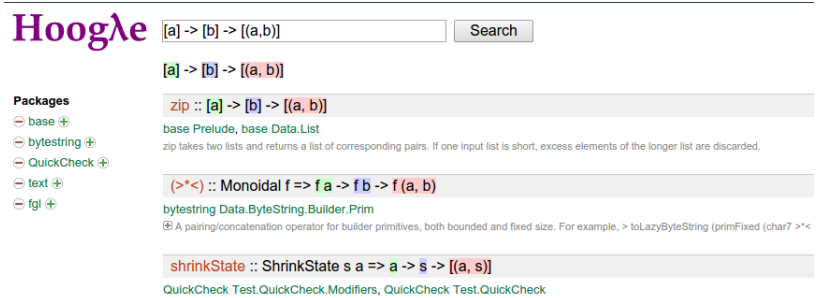


Evaluación perezosa
Haskell retrasa la evaluación de una expresión todo lo posible:
head [1..10**9] -- Sólo evalua 1
Esto permite la modularización del código:
min :: Ord a => [a] -> a
min = head . sort
Y el uso de estructuras infinitas:
unos = 1:unos
diezDoses = take 10 (map (+1) unos)

- Haskell comprueba el tipo de las expresiones (evaluando la WHNF) por lo que length [1, 1/0, 2] es válido pero length [1, .ªdas", 2] no.
- min es O(n) + O(n/2) + O(n/4) + ... = O(n)

# Hoogle

Hoogle permite buscar funciones por tipo entre las librerías estándar de Haskell:






# Demostraciones

Como las funciones no tienen efectos secundarios, podemos razonar la corrección del código por inducción:

```
qsort []     = []
qsort (x:xs) = qsort [y | y<-xs, y<=x]
               ++ [x]
               ++ qsort [y | y<-xs, y>x]
```

**Demostración:** *Quicksort* funciona porque:

- ordena correctamente una lista vacía.
- la lista creada mantiene el orden entre las tres partes



Demostraciones

Como las funciones no tienen efectos secundarios, podemos razonar la corrección del código por inducción:

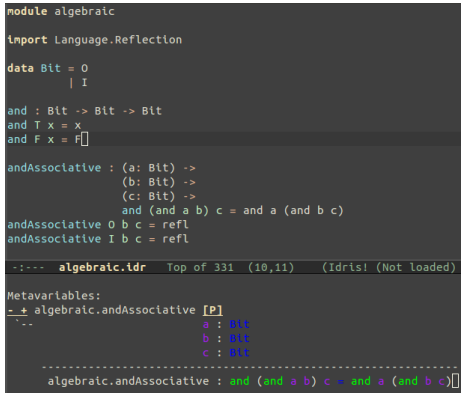
qsort [] = []
qsort (x:xs) = qsort [y | y<-xs, y<=x]
++ [x]
++ qsort [y | y<-xs, y>x]

**Demostración:** Quicksort funciona porque:

- ordena correctamente una lista vacía.
- la lista creada mantiene el orden entre las tres partes

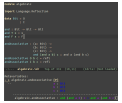
# Idris

**Idris**, construido encima de Haskell, demuestra matemáticamente que los programas son correctos.






Idris
**Idris**, construido encima de Haskell, demuestra matemáticamente que los programas son correctos.



# Curry-Howard

Los siguientes tipos están habitados:

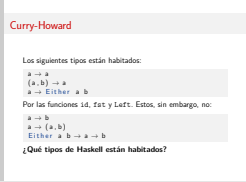
```
a → a
(a,b) → a
a → Either a b
```

Por las funciones `id`, `fst` y `Left`. Estos, sin embargo, no:

```
a → b
a → (a,b)
Either a b → a → b
```

**¿Qué tipos de Haskell están habitados?**





# Curry-Howard

A cada tipo le corresponde una proposición lógica, cambiando:

- `a -> b` por $a \Rightarrow b$
- `(a,b)` por $a \wedge b$
- `Either a b` por $a \vee b$
- `()` por *True*
- `Void` por *False*

**¿Qué tipos de Haskell están habitados?** Aquellos cuya proposición lógica asociada puede demostrarse verdadera.



Curry-Howard

A cada tipo le corresponde una proposición lógica, cambiando:
- a -> b por $a \Rightarrow b$
- (a,b) por $a \wedge b$
- Either a b por $a \vee b$
- () por *True*
- Void por *False*

**¿Qué tipos de Haskell están habitados?** Aquellos cuya proposición lógica asociada puede demostrarse verdadera.

# Mónadas

Las mónadas son constructores de tipos que definen un **contexto computacional**. Se definen con:

```
return :: a → m a
```

`return` introduce un valor de tipo a en el contexto.

```
>>= :: m a → (a → m b) → m b
```

Dado un valor en un contexto y una función que devuelve valores en el contexto, **¿Cómo podemos aplicar la función al valor?**-



Mónadas

Las mónadas son constructores de tipos que definen un **contexto computacional**. Se definen con:

```
return :: a → m a
```

`return` introduce un valor de tipo a en el contexto.

```
>>= :: m a → (a → m b) → m b
```

Dado un valor en un contexto y una función que devuelve valores en el contexto, **¿Cómo podemos aplicar la función al valor?**-

# Mónadas: Maybe

La mónada `Maybe`, nos permite operar con funciones que pueden *fallar*:

```
return x = Just x -- return = Just
```

```
Just x  >>= f = f x
Nothing >>= _ = Nothing
```

Cuando hay un fallo no devolvemos nada, y cuando no aplicamos la función al valor del interior de `Just`.



Mónadas: Maybe
La mónada Maybe, nos permite operar con funciones que pueden fallar:
return x = Just x -- return = Just
Just x >>= f = f x
Nothing >>= _ = Nothing
Cuando hay un fallo no devolvemos nada, y cuando no aplicamos la función al valor del interior de Just.

# Mónadas: IO

**¿Cómo hacemos E/S con funciones puras?**
Utilizamos la mónada `IO`:

```
main = readFile "file1" >>= writeFile "file2"
```

Todo programa de Haskell se ejecuta evaluando la función `main`, de tipo `IO ()`.



Mónadas: IO
¿Cómo hacemos E/S con funciones puras?
Utilizamos la mónada IO:
main = readFile "file1" >>= writeFile "file2"
Todo programa de Haskell se ejecuta evaluando la función main, de tipo IO ().

Su implementación es abstracta, pero nos permite componer acciones.

# Funciones de orden superior

map:

```
map :: (a → b) → ([a] → [b])
map f []     = []
map f (x:xs) = f x : map f xs
```

foldr:

```
foldr :: (a → b → b) → b → [a] → b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

◂ Funciones



Funciones de orden superior

map:

```
map :: (a → b) → ([a] → [b])
map f []     = []
map f (x:xs) = f x : map f xs
```

foldr:

```
foldr :: (a → b → b) → b → [a] → b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

◂ Funciones