

# Programación Funcional en Haskell

## Demostraciones

Paradigmas de Lenguajes de Programación

Departamento de Ciencias de la Computación  
Universidad de Buenos Aires

8 de abril de 2025

- Tipos algebraicos en Haskell
- Esquemas de recursión
- Currificación
- Aplicación parcial
- Alto orden
- Generación infinita
- ...

- Razonamiento ecuacional
- Extensionalidad
- Lemas de generación
- Inducción estructural
- Ejercicios más difíciles
- ...

Básicamente, vamos a ver cómo demostrar propiedades sobre nuestros programas.

# Demostrando propiedades

Sean las siguiente definiciones:

```
doble :: Integer -> Integer
```

```
doble x = 2 * x
```

```
cuadrado :: Integer -> Integer
```

```
cuadrado x = x * x
```

¿Cómo probamos que `doble 2 = cuadrado 2`?

**Solución:**

```
doble 2 =doble 2 * 2 =cuadrado cuadrado 2  $\square$ 
```

Queremos ver que:

$$\text{curry} \ . \ \text{uncurry} = \text{id}$$

Tenemos:

```
curry :: ((a, b) -> c) -> (a -> b -> c)
```

```
curry f = (\x y -> f (x, y))
```

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
```

```
uncurry f = (\(x, y) -> f x y)
```

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

```
(f . g) x = f (g x)
```

```
id :: a -> a
```

```
id x = x
```

¿Cómo hacemos?

Dadas  $f, g :: a \rightarrow b$ , probar  $f = g$  se reduce a probar:

$$\forall x :: a . f\ x = g\ x$$

# Algunas propiedades útiles

Estas son algunas propiedades que podemos usar en nuestras demostraciones:

$$\forall F :: a \rightarrow b . \forall G :: a \rightarrow b . \forall Y :: b . \forall Z :: a .$$

$$F = G \quad \Leftrightarrow \quad \forall x :: a . F \ x = G \ x$$

$$F = \backslash x \rightarrow Y \quad \Leftrightarrow \quad \forall x :: a . F \ x = Y$$

$$(\backslash x \rightarrow Y) \ Z \quad =_{\beta} \quad Y \text{ reemplazando } x \text{ por } Z$$

$$\backslash x \rightarrow F \ x \quad =_{\eta} \quad F$$

F, G, Y y Z pueden ser expresiones complejas, siempre que la variable x no aparezca libre en F, G, ni Z (más detalles cuando veamos Cálculo Lambda).

## Volviendo al ejercicio...

Ahora probemos:

$$\text{curry} . \text{uncurry} = \text{id}$$

Tenemos:

$\text{curry} :: ((a, b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$

$\{C\} \text{ curry } f = (\backslash x \ y \rightarrow f \ (x, y))$

$\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow ((a, b) \rightarrow c)$

$\{U\} \text{ uncurry } f = (\backslash (x, y) \rightarrow f \ x \ y)$

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

$\{COMP\} (f . g) \ x = f \ (g \ x)$

$\text{id} :: a \rightarrow a$

$\{I\} \text{ id } x = x$



## Pares y unión disjunta/tipo suma

Se define la siguiente función, que permite multiplicar pares y enteros entre sí (usando producto escalar entre pares).

```
prod :: Either Int (Int, Int) -> Either Int (Int, Int) -> Either Int (Int, Int)
{P0} prod (Left x) (Left y) = Left (x * y)
{P1} prod (Left x) (Right (y, z)) = Right (x * y, x * z)
{P2} prod (Right (y, z)) (Left x) = Right (y * x, z * x)
{P3} prod (Right (w, x)) (Right (y, z)) = Left (w * y + x * z)
```

¿Podemos probar esto?

$$\forall p :: \text{Either Int (Int, Int)} . \forall q :: \text{Either Int (Int, Int)} . \text{prod } p \text{ } q = \text{prod } q \text{ } p$$

Recordemos los lemas de generación para pares y sumas.

Dado  $p :: (a, b)$ , siempre podemos usar el hecho de que existen  $x :: a$ ,  $y :: b$  tales que  $p = (x, y)$ .

De la misma manera, dado  $e :: \text{Either } a \ b$ , siempre podemos usar el hecho de que:

- $e = \text{Left } x$  con  $x :: a$ , o
- $e = \text{Right } y$  con  $y :: b$ .

# Pares y unión disjunta/tipo suma

Probemos enconces...

$$\forall p :: \text{Either Int (Int, Int)} . \forall q :: \text{Either Int (Int, Int)} . \text{prod } p \ q = \text{prod } q \ p$$

Tenemos:

```
prod :: Either Int (Int, Int) -> Either Int (Int, Int) -> Either Int (Int, Int)
{P0} prod (Left x) (Left y) = Left (x * y)
{P1} prod (Left x) (Right (y, z)) = Right (x * y, x * z)
{P2} prod (Right (y, z)) (Left x) = Right (y * x, z * x)
{P3} prod (Right (w, x)) (Right (y, z)) = Left (w * y + x * z)
```

# Funciones como estructuras de datos

Se cuenta con la siguiente representación de conjuntos:

`type Conj a = (a->Bool)` representados por su función característica. De este modo, si  $c$  es un conjunto y  $e$  un elemento, la expresión  $c\ e$  devuelve `True` si  $e$  pertenece a  $c$  y `False` en caso contrario.

Contamos con las siguientes definiciones:

```
vacío :: Conj a
{V} vacío = \_ -> False
intersección :: Conj a -> Conj a -> Conj a
{I} intersección c d = \e -> c e && d e
```

```
agregar :: Eq a => a -> Conj a -> Conj a
{A} agregar x c = \e -> e == x || c e
diferencia :: Conj a -> Conj a -> Conj a
{D} diferencia c d = \e -> c e && not (d e)
```

Demostrar la siguiente propiedad:

$$\forall c :: \text{Conj } a. \forall d :: \text{Conj } a. \text{intersección } d (\text{diferencia } c\ d) = \text{vacío}$$

- Pruebo  $P(0)$
- Pruebo que **si** vale  $P(n)$  **entonces** vale  $P(n + 1)$ .

- Pruebo  $P([])$
- Pruebo que **si** vale  $P(xs)$  **entonces** para todo elemento  $x$  vale  $P(x:xs)$ .

## En el caso general (inducción estructural)

- Pruebo  $P$  para el o los caso(s) base (para los constructores no recursivos).
- Pruebo que **si** vale  $P(Arg_1), \dots, P(Arg_k)$  **entonces** vale  $P(C\ Arg_1 \ \dots \ Arg_k)$  para cada constructor  $C$  y sus argumentos recursivos  $Arg_1, \dots, Arg_k$ .  
(Los argumentos no recursivos quedan cuantificados universalmente).

# Pasos a seguir

- Leer la propiedad, entenderla y convencerse de que es verdadera.
- Plantear la propiedad como predicado unario.
- Plantear el esquema de inducción.
- Plantear y resolver el o los caso(s) base.
- Plantear y resolver el o los caso(s) inductivo(s).



Veamos que estas dos definiciones de length son equivalentes:

```
length1 :: [a] -> Int
```

```
{L10} length1 [] = 0
```

```
{L11} length1 (_:xs) = 1 + length1 xs
```

```
length2 :: [a] -> Int
```

```
{L2} length2 = foldr (\_ res -> 1 + res) 0
```

Recordemos:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
{F0} foldr f z [] = z
```

```
{F1} foldr f z (x:xs) = f x (foldr f z xs)
```

# Demostrando implicaciones

Queremos probar que:

$$\text{Ord } a \Rightarrow \forall e :: a . \forall ys :: [a] . (\text{elem } e \text{ } ys \Rightarrow e \leq \text{maximum } ys)$$

Antes que nada, ¿quién es  $P$ ?

¿En qué estructura vamos a hacer inducción?

$$P(ys) = \text{Ord } a \Rightarrow \forall e :: a . (\text{elem } e \text{ } ys \Rightarrow e \leq \text{maximum } ys)$$

Ahora bien, si no vale  $\text{Ord } a$ , la implicación de afuera es trivialmente verdadera (recordar que las implicaciones asocian a derecha). Además, si vale  $\text{Ord } a$ , también vale  $\text{Eq } a$  (por la jerarquía de clases de tipos en Haskell).

Suponemos que todo eso vale y vamos a probar lo que nos interesa.

## Demostrando implicaciones (continúa)

Tenemos:

```
elem :: Eq a => a -> [a] -> Bool
```

```
{E0} elem e [] = False
```

```
{E1} elem e (x:xs) = (e == x) || elem e xs
```

```
maximum :: Ord a => [a] -> a
```

```
{M0} maximum [x] = x
```

```
{M1} maximum (x:y:ys) = max x (maximum (y:ys))
```

Sabemos que valen `Eq a` y `Ord a`. Queremos ver que, para toda lista `ys`, vale:

$$\forall e :: a. (\text{elem } e \text{ } ys \Rightarrow e \leq \text{maximum } ys)$$

Seguimos en el pizarrón.

## Otra vuelta de tuerca

Dadas las siguientes definiciones:

```
length :: [a] -> Int
```

```
{L0} length [] = 0
```

```
{L1} length (x:xs) = 1 + (length xs)
```

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

```
{F0} foldl f ac [] = ac
```

```
{F1} foldl f ac (x:xs) = foldl f (f ac x) xs
```

```
reverse :: [a] -> [a]
```

```
{R} reverse = foldl (flip (:)) []
```

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

```
{FL} flip f x y = f y x
```

Queremos probar que:  $\forall ys :: [a] . \text{length } ys = \text{length } (\text{reverse } ys)$

...que, por reverse, es lo mismo que:

$\forall ys :: [a] . \text{length } ys = \text{length } (\text{foldl } (\text{flip } (:)) [] ys)$

Avancemos hasta que nos trabemos.

$$P(ys) = \text{length } ys = \text{length } (\text{foldl } (\text{flip } (:)) [] ys)$$

En el caso inductivo ( $ys = x:xs$ ) nuestra Hipótesis Inductiva es:

$$\text{length } xs = \text{length } (\text{foldl } (\text{flip } (:)) [] xs)$$

Pero lo que necesitamos es:

$$1 + \text{length } xs = \text{length } (\text{foldl } (\text{flip } (:)) (x:[]) xs)$$

¿Qué podemos hacer?

**Respuesta:** demostrar una propiedad más general.

Probemos:  $\forall ys :: [a]. \forall zs :: [a]. \text{length } zs + \text{length } ys =$   
 $\text{length } (\text{foldl } (\text{flip } (:)) zs ys)$

Luego, tomando  $zs = []$  y sabiendo que  $\text{length } [] = 0$ , obtenemos lo que buscábamos.

## ¿Recuerdan esta función?

```
flipTake :: [a] -> Int -> [a]
{FT} flipTake = foldr
    (\x rec n -> if n==0 then [] else x:rec (n-1))
    (const [])
```

Dada esta versión alternativa con recursión explícita:

```
take' :: [a] -> Int -> [a]
{T0} take' [] _ = []
{T1} take' (x:xs) n = if n==0 then [] else x:take' xs (n-1)
```

¿Podemos probar que `take' = flipTake`?

## ¿Está bien lo que hicimos?

Tenemos:

```
take' :: [a] -> Int -> [a]
```

```
{T0} take' [] _ = []
```

```
{T1} take' (x:xs) n = if n==0 then [] else x : take' xs (n-1)
```

```
flipTake :: [a] -> Int -> [a]
```

```
{FT} flipTake = foldr
```

```
  (\x rec n -> if n==0 then [] else x : rec (n-1))
```

```
  (const [])
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
{F0} foldr f z [] = z
```

```
{F1} foldr f z (x:xs) = f x (foldr f z xs)
```

```
const :: (a -> b -> a)
```

```
{C} const = (\x -> \_ -> x)
```

Probemos que `take' = flipTake`.

# Demostrando propiedades sobre árboles

Dadas las siguientes funciones:

```
cantNodos :: AB a -> Int
```

```
{CN0} cantNodos Nil = 0
```

```
{CN1} cantNodos (Bin i r d) = 1 + (cantNodos i) + (cantNodos d)
```

```
inorder :: AB a -> [a]
```

```
{I0} inorder Nil = []
```

```
{I1} inorder (Bin i r d) = (inorder i) ++ (r:inorder d)
```

```
length :: [a] -> Int
```

```
{L0} length [] = 0
```

```
{L1} length (x:xs) = 1 + (length xs)
```

Queremos probar:

$$\forall t :: AB\ a. \text{cantNodos } t = \text{length } (\text{inorder } t)$$



# ¿Y ahora qué hacemos?

¡Necesitamos un lema!

$$\forall xs :: [a] . \forall ys :: [a] . \text{length } (xs ++ ys) = \text{length } xs + \text{length } ys$$

# Una propiedad sobre árboles... y un lema sobre listas

Ahora sí:

```
cantNodos :: AB a -> Int
```

```
{CNO} cantNodos Nil = 0
```

```
{CN1} cantNodos (Bin i r d) = 1 + (cantNodos i) + (cantNodos d)
```

```
inorder :: AB a -> [a]
```

```
{I0} inorder Nil = []
```

```
{I1} inorder (Bin i r d) = (inorder i) ++ (r:inorder d)
```

```
length :: [a] -> Int
```

```
{L0} length [] = 0
```

```
{L1} length (x:xs) = 1 + (length xs)
```

**Lema:**  $\forall xs :: [a] . \forall ys :: [a] . \text{length } (xs ++ ys) = \text{length } xs + \text{length } ys$

Queremos probar:

$$\forall t :: AB\ a . \text{cantNodos } t = \text{length } (\text{inorder } t)$$

¡No nos olvidemos de probar el lema!

# Demostremos el lema

`(++) :: [a] -> [a] -> [a]`

`{C0} [] ++ ys = ys`

`{C1} (x:xs) ++ ys = x : (xs ++ ys)`

`length :: [a] -> Int`

`{L0} length [] = 0`

`{L1} length (x:xs) = 1 + (length xs)`

**Lema:**  $\forall xs :: [a] . \forall ys :: [a] . \text{length } (xs ++ ys) = \text{length } xs + \text{length } ys$

Si quisiéramos demostrar una propiedad sobre el tipo `Árbol23 a b` mediante inducción estructural:

```
data Árbol23 a b =  
  Hoja a  
| Dos b (Árbol23 a b) (Árbol23 a b)  
| Tres b b (Árbol23 a b) (Árbol23 a b) (Árbol23 a b)
```

Para demostrar que vale  $\forall q : \text{Árbol23 } a \ b . P(q)$ :

¿Cuál es o cuáles son los casos base?

¿Cuál es o cuáles son los casos inductivos? ¿Y la(s) hipótesis inductiva(s)?

ι ι ι ι ι ι ι ι ι ι ? ? ? ? ? ? ? ?