Programación Funcional en Haskell

Demostraciones

Paradigmas de Lenguajes de Programación

Departamento de Ciencias de la Computación
Universidad de Buenos Aires

9 de septiembre de 2025

Anteriormente en PLP...

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

Hoy

- 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` $=_{\texttt{doble}}$ `2 * 2` $=_{\texttt{cuadrado}}$ `cuadrado 2` $\square$

Igualdad de funciones

Queremos ver que:

```
curry . uncurry = 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?

Extensionalidad

Dadas `f, g :: a -> b`, probar f $=$ g se reduce a probar:

$$\forall \texttt{x::a} \,.\, \texttt{f x} = \texttt{g x}$$

Algunas propiedades útiles

Estas son algunas propiedades que podemos usar en nuestras demostraciones:

$$\forall \texttt{F :: a -> b . } \forall \texttt{G :: a -> b . } \forall \texttt{Y :: b . } \forall \texttt{Z :: a .}$$

`F` $=$ `G`	$\Leftrightarrow$	$\forall$`x::a.F x` $=$ `G x`
`F` $=$ `\x -> Y`	$\Leftrightarrow$	$\forall$`x::a.F x` $=$ `Y`
`(\x -> Y) Z`	$=_\beta$	`Y` reemplazando `x` por `Z`
`\x -> F x`	$=_\eta$	`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:

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

Tenemos:

```
curry :: ((a, b) -> c) -> (a -> b -> c)
{C} curry f = (\x y -> f (x, y))
uncurry :: (a -> b -> c) -> ((a, b) -> c)
{U} uncurry f = (\(x, y) -> f x y)
(.) :: (b -> c) -> (a -> b) -> (a -> c)
{COMP} (f . g) x = f (g x)
id :: a -> a
{I} id x = x
```

Pares y unión disjunta/tipo suma

Dado el tipo de la unión disjunta:
`data Either a b = Left a | Right b`

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?

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

Pares y unión disjunta/tipo suma

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 :: Either a b`, siempre podemos usar el hecho de que:

- `e = Left x` con `x :: a`, o
- `e = Right y` con `y :: b`.

Pares y unión disjunta/tipo suma

Probemos enconces...

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

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:

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

Inducción en los naturales

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

Inducción en listas

- Pruebo $P([\,])$
- Pruebo que si vale $P(\mathtt{xs})$ entonces para todo elemento $\mathtt{x}$ vale $P(\mathtt{x:xs})$.

En el caso general (inducción estructural)

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

Desplegando foldr

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

Dados dos tipos `a` y `b` tales que valen `Eq a` y `Eq b`, queremos probar que:

$$\forall \texttt{f::a->b} \,.\, \forall \texttt{e::a} \,.\, \forall \texttt{xs::[a]} \,.\, \texttt{elem e xs} \Rightarrow \texttt{elem (f e) (map f xs)}$$

Antes que nada, ¿quién es P?
¿En qué estructura vamos a hacer inducción?

$$P(\texttt{ys}) = \forall \texttt{f::a->b} \,.\, \forall \texttt{e::a} \,.\, \texttt{elem e ys} \Rightarrow \texttt{elem (f e) (map f ys)}$$

Demostrando implicaciones (continúa)

Tenemos:

```
elem :: Eq a => a -> [a] -> Bool
{E0} elem e [] = False
{E1} elem e (x:xs) = (e == x) || elem e xs
```

```
map :: (a -> b) => [a] -> [b]
{M0} map f [] = []
{M1} map f (y:ys) = f y : map f ys
```

Podemos usar:

{`Congruencia ==`} $\forall$`f::a->b` . $\forall$`x::a` . $\forall$`y::a` . `x == y` $\Rightarrow$ `f x == f y`

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

$\forall$`f::a->b` . $\forall$`e::a` . `elem e ys` $\Rightarrow$ `elem (f e) (map f 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] . length ys = length (reverse ys)`

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

$\forall$ `ys::[a] . length ys = length (foldl (flip (:)) [] ys)`

Avancemos hasta que nos trabemos.

Generalizando propiedades

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

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

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

Pero lo que necesitamos es:

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

¿Qué podemos hacer?

Respuesta: demostrar una propiedad más general.
Probemos: $\forall \texttt{ys::[a]}.\forall \texttt{zs::[a]}.\texttt{length zs} + \texttt{length ys} =$
$\texttt{length (foldl (flip (:)) zs ys)}$
Luego, tomando `zs` $=$ `[]` y sabiendo que `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:

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

¿Y ahora qué hacemos?

¡Necesitamos un lema!

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

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

Ahora sí:

```
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)
```

Lema: ∀ `xs::[a]` . ∀ `ys::[a]` . `length (xs++ys)` = `length xs + length ys`

Queremos probar:

∀ `t::AB a` . `cantNodos t` = `length (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: ∀xs::[a] . ∀ys::[a] . `length (xs++ys)` = `length xs + length ys`

Últimas preguntas

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::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)?

Apunte recomendado

"Ejemplo de inducción estructural con implicación"
Disponible en la sección **Útil** del campus.

Fin

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ? ? ? ? ? ? ? ? ?