Clase Práctica 3 : Demostración en Programación Funcional (Haskell)

Tomás Felipe Melli

July 11, 2025

Índice

1 Demostrando propiedades **2**

2 Igualdad de funciones **2**

3 Pares y unión disjunta / tipo suma **3**

4 Funciones como estructuras de datos **3**

5 Inducción en los naturales **4**

6 Inducción en listas **4**

7 Inducción estructural (caso general) **4**

8 Desplegando `foldr` **5**

9 Demostrando implicaciones **5**

10 Vuelta de tuerca : `length ys = length (reverse ys)` **7**
10.1 Generalizamos propiedades 8

11 Ejercicio : `take` **8**

12 Demostrando propiedades sobre árboles **10**

13 Últimas preguntas **11**

1 Demostrando propiedades

Sean

```
doble :: Integer -> Integer
doble x = 2 * x

cuadrado :: Integer -> Integer
cuadrado x = x * x
```

Queremos probar que `doble 2 = cuadrado 2`

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

2 Igualdad de funciones

Queremos ver que

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

Cómo encaramos esto ?
Dadas `f, g :: a → b`, probar `f = g` se reduce a probar :

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

Podemos usar ciertas propiedades en nuestras demostraciones :

$$\forall\ \texttt{F :: a → b.}\quad \forall\ \texttt{G :: a → b.}\quad \forall\ \texttt{Y :: b.}\quad \forall\ \texttt{Z :: a}$$

$$\begin{aligned}
&\texttt{F} = \texttt{G} && \iff \forall\ \texttt{x :: a. F x = G x}\\
&\texttt{F} = \texttt{\textbackslash x → Y} && \iff \forall\ \texttt{x :: a. F x = Y}\\
&\texttt{(\textbackslash x → Y Z)} && \overset{\beta}{=} \texttt{Y reemplazando x por Z}\\
&\texttt{\textbackslash x → F x} && \overset{\eta}{=} \texttt{F}
\end{aligned}$$

`F, G, Y` y `Z` pueden ser expresiones complejas, siempre que la variable `x` no aparezca libre en `F, G` ni `Z`
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
```

Por extensionalidad, es lo mismo que probar `(curry . uncurry) f = id f` $\forall f :: a \to b \to c$

$$\begin{aligned}
\texttt{(curry . uncurry) f} &\overset{\{\texttt{COMP}\}}{=} \texttt{curry (uncurry f)}\\
&\overset{\{\texttt{U}\}}{=} \texttt{curry (\textbackslash(x,y) → f x y)}\\
&\overset{\{\texttt{C}\}}{=} \texttt{\textbackslash x' y' → (\textbackslash(x,y) → f x y) (x',y')}\\
&\overset{\beta}{=} \texttt{\textbackslash x' y' → f x' y'}\\
&\overset{\eta}{=} \texttt{\textbackslash x' → f x'}\\
&\overset{\eta}{=} \texttt{f}\\
&\overset{\{I\}}{=} \texttt{id f}
\end{aligned}$$

También se podía aplicar extensionalidad dos veces más y hacer que le entre x e y y se evitaban las lambdas.

3 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)
{P2} prod (Right (w,x)) (Right (y,z)) = Left (w * y + x * z)
```

Queremos probar

$\forall$ `p :: Either Int (Int,Int)`.$\forall$ `q :: Either Int (Int,Int).` `prod p q = prod q p`

Recordamos los **lemas de generación de 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`

Por tanto, procedemos a la demo.
`Por lema de generación de suma, p puede ser Left n con n :: Int, o Right parUno con parUno :: (Int,Int)`
`y que puede ser Left m con m :: Int o Right parDos con parDos :: (Int,Int)`
`Por lema de generación de pares, parUno = (w,x) con w,x :: Int y parDos = (y,z) con y,z :: Int`

`Caso p = Left n, q = Left m`

$$\begin{aligned} \texttt{prod p q} &\overset{\{P0\}}{=} \texttt{Left (n*m)} \\ &\overset{Int}{=} \texttt{Left (m*n)} \\ &\overset{\{P0\}}{=} \texttt{prod q p} \end{aligned}$$

`Caso p = Left n, q = Right (y,z)`

$$\begin{aligned} \texttt{prod p q} &\overset{\{P1\}}{=} \texttt{Right (n*y,n*z)} \\ &\overset{Int}{=} \texttt{Right (y*n,n*z)} \\ &\overset{Int}{=} \texttt{Right (y*n,z*n)} \\ &\overset{\{P2\}}{=} \texttt{prod q p} \end{aligned}$$

`Caso p = Right (w,x), q = Left m`

$$\begin{aligned} \texttt{prod p q} &\overset{\{P2\}}{=} \texttt{Right (w*m,x*m)} \\ &\overset{Int}{=} \texttt{Right (m*w,x*m)} \\ &\overset{Int}{=} \texttt{Right (m*w,m*x)} \\ &\overset{\{P1\}}{=} \texttt{prod q p} \end{aligned}$$

`Caso p = Right (w,x), q = Right (y,z)`

$$\begin{aligned} \texttt{prod p q} &\overset{\{P3\}}{=} \texttt{Left (w*y + x*z)} \\ &\overset{Int}{=} \texttt{Left (y*w + x*z)} \\ &\overset{Int}{=} \texttt{Left (y*w + z*x)} \\ &\overset{\{P3\}}{=} \texttt{prod q p} \end{aligned}$$

4 Funciones como estructuras de datos

Se cuenta con la siguiente representación de conjuntos : `type Conj a = (a→ Bool)` caracterizados por su función de pertenencia. De este modo, si c es un conjunto y e un elemento, la expresión c e devuelve `True` si $e \in c$ y `False` en caso contrario. Contamos con las siguientes definiciones:

```
vacio :: Conj a
{V} vacio = \ -> False

interseccion :: Conj a-> Conj a-> Conj a
{I} interseccion c d = \e -> c e && d e

agregar :: Eq a => a -> Conj a -> Conj a
{A} agregar e c = \e -> e == x || c e

diferencia :: Conj a -> Conj a-> Conj a
{D} diferencia c d = \e -> c e && not (d e)
```

Queremos demostrar la siguiente propiedad:

$\forall$`c :: Conj a.` $\forall$ `d :: Conj a. interseccion d (diferencia c d) = vacio`

Por extensionalidad, basta ver que $\forall x :: a$

$$\begin{aligned}
&\texttt{interseccion d (diferencia c d) x} = \texttt{vacio x} \\
&\overset{\{V\}}{=} \texttt{(_ -> False) x} \\
&\overset{\beta}{=} \texttt{False}
\end{aligned}$$

$$\begin{aligned}
&\texttt{interseccion d (diferencia c d) x} \overset{\{D\}}{=} \texttt{interseccion d (\e -> c e \&\& not (d e)) x} \\
&\overset{\{I\}}{=} \texttt{(\h -> d h \&\& (\e -> c e \&\& not (d e)) h) x} \\
&\overset{\beta}{=} \texttt{(\h -> d h \&\& (c h \&\& not (d h)) x} \\
&\overset{\beta}{=} \texttt{d x \&\& (c x \&\& not (d x))} \\
&\overset{Bool}{=} \texttt{false \&\& c x} \\
&\overset{Bool}{=} \texttt{false}
\end{aligned}$$

5 Inducción en los naturales

Es inducción estructural en la estructura de los naturales

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

6 Inducción en listas

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

7 Inducción estructural (caso general)

- Pruebo para `P` el o los casos base (para los constructores no recursivos)
- Pruebo que si vale `P(arg1), ..., P(argK)` entonces vale `P(C arg1 .. argK)` para cada constructor `C` y sus argumentos recursivos `arg1, ... , argK` (los argumentos no recursivo quedan cuatificados universalmente)

Pasos a seguir

1. Leer la propiedad, entenderla, convencerse de que es verdadera.
2. Plantear las propiedad como **predicado unario** (o sea, que depende de un sólo parámetro que va a ser la estructura sobre la que vamos a hacer inducción. Si hay varios para todo(∀), elegimos uno, lo sacamos y lo que queda es el predicado. Sacar un para todo (∀))

3. Plantear el esquema de inducción.
4. Plantear y resolver el o los casos base.
5. Plantear y resolver el o los casos recursivos.

8 Desplegando `foldr`

Veamos que estas dos definiciones de `length` son equivalentes

```
1  length1 :: [a] -> Int
2  {L10} length1 [] = 0
3  {L11} length1 ( :xs) = 1 + length1 xs
4
5  length2 :: [a] -> Int
6  {L2} length2 = foldr (\ res -> 1 + res) 0
7
8  Recordemos:
9  foldr :: (a -> b -> b) -> b -> [a] -> b
10 {F0} foldr f z [] = z
11 {F1} foldr f z (x:xs) = f x (foldr f z xs)
```

Queremos ver que

`length2 = length1`

`length2 = foldr (\_ res → uno + res) 0`

Por extensionalidad, queremos ver que para todo `xs :: [a], foldr (\_ res → uno + res) 0 xs = length1 xs`.
Hacemos inducción estructural sobre `xs`, (se va a necesitar la HI, sino se podría demostrar por lema de generación de listas).
`P(xs) = foldr (\_ res → uno + res) 0 xs = length1 xs`

- Caso base: `xs = []`

$$\texttt{foldr (\_ res → uno + res) 0 []} \overset{\{F0\}}{=} \texttt{0} \overset{\{L10\}}{=} \texttt{length1 []} = \square$$

- Paso inductivo : `xs = (y:ys)`

`HI : foldr (\_ res → uno + res) 0 ys = length1 ys`
`TI : foldr (\_ res → uno + res) 0 (y:ys) = length1 (y:ys)`

$$\begin{aligned}\texttt{foldr (\_ res → uno + res) 0 (y:ys) = length1 (y:ys)} &\overset{\text{renombre}}{=} \texttt{foldr f 0 (y:ys)} \\ &\overset{\{F1\}}{=} \texttt{foldr f y (foldr f 0 ys)} \\ &\overset{\beta}{=} \texttt{(\res → uno + res) (foldr f 0 ys)} \\ &\overset{\beta}{=} \texttt{uno + (foldr f 0 ys)} \\ &\overset{HI}{=} \texttt{uno + length1 ys} \\ &\overset{\{L11\}}{=} \texttt{length1 (y:ys)}\end{aligned}$$

9 Demostrando implicaciones

Queremos probar que

`Ord a ⇒ ∀e::a. ∀ ys :: [a]. (elem e ys ⇒ e ≤ maximum ys)`

Quién es `P`. Sobre qué estructura vamos a hacer inducción ?

`P(ys) = Ord a ⇒ ∀ e :: a. (elem e ys ⇒ e ≤ maximum ys)`

Ahora bien, si no vale `Ord a`, la implicación de afuera es trivialmente verdadera (recordar que las implicaciones asocian a derecha). Además, si vale `Ord a`, también vale `Eq a` (por la jerarquía de clases de tipos en Haskell). Suponemos que todo eso vale y vamos a probar los que nos interesa. Miremos

```haskell
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) = if x < maximum (y:ys) then maximum (y:ys) else x
```

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

$\forall$`e :: a. (elem e ys` $\Rightarrow$ `e` $\leq$ `maximum ys)`

Queremos ver que $\forall$`ys :: [a]. P(ys)`
Por inducción estructural en `ys`, veamos que vale `P([])` y $\forall$ `x :: a. P(xs)` $\Rightarrow$ `P(x:xs)`

- Caso base : `P([])` = $\forall$ `e :: a. (elem e []` $\Rightarrow$ `e` $\leq$ `maximum []`

  `elem e []` $\overset{E0}{=}$ *False* `por tanto vale la implicación`

- Paso inductivo : $\forall$ `y :: a. P(ys)` $\Rightarrow$ `P(y:ys)`
  Nuestra HI será : `P(ys)` = $\forall$ `e :: a. elem e ys` $\Rightarrow$ `e` $\leq$ `maximum ys` y queremos ver que vale `P(y:ys)` = $\forall$ `e :: a. elem e (y:ys)` $\Rightarrow$ `e` $\leq$ `maximum (y:ys)`
  Sea `e :: a` tal que `True = elem e (y:ys)`. Queremos ver que `e` $\leq$ `maximum (y:ys)`
  Por lema de generación de listas

  a) `ys = []`

  b) $\exists$ `z :: a, zs ::[a] tal que ys = z:zs`

  Veamos el **caso a**

  $$\begin{aligned} \texttt{e} \leq \texttt{maximum (y:ys)} &\equiv \texttt{e} \leq \texttt{maximum [y]} \\ &\overset{M0}{\equiv} \texttt{e} \leq \texttt{y} \end{aligned}$$

  `como sabemos que True = elem e (y:ys) en este caso True = elem e (y:[])`

  $$\begin{aligned} \texttt{True = elem e (y:[])} &\overset{E1}{\equiv} \texttt{(e == y) || elem e []} \\ &\overset{E0}{\equiv} \texttt{(e == y) || False} \\ &\overset{Bool}{\equiv} \texttt{(e == y)} \end{aligned}$$

  `Entonces e = y en particular e` $\leq$ `y`

  Veamos el **caso b**

  $$\begin{aligned} \texttt{e} \leq \texttt{maximum (y:ys)} &\equiv \texttt{e} \leq \texttt{maximum (y:z:zs)} \\ &\overset{M1}{\equiv} \texttt{e} \leq \texttt{if y < maximum (z:zs) then maximum (z:zs) else y} \\ &\equiv \texttt{e} \leq \texttt{if y < maximum (ys) then maximum (ys) else y} \\ &\equiv * \end{aligned}$$

  Por lema de generación de bool, separamos en dos casos
  `Caso b1 : y < maximum ys = True`

  $$\begin{aligned} (*) &\equiv \texttt{e} \leq \texttt{if True then maximum ys else y} \\ &\overset{Bool}{\equiv} \texttt{e} \leq \texttt{if True then maximum ys} \end{aligned}$$

  $$\begin{aligned} \texttt{Habíamos asumido} \rightarrow \texttt{True} &= \texttt{elem e (y:ys)} \\ &= \texttt{elem e (y:ys) = (e == y) || elem e ys} \end{aligned}$$

  Por lema de generación de Bool, separamos en casos
  `Caso b11 : (e == y) = True`

  $$\begin{aligned} \texttt{e} \leq \texttt{maximum ys} &\equiv \texttt{y} \leq \texttt{maximum ys} \\ &\overset{B1}{\equiv} \texttt{True} \quad \texttt{Esto vale ya que por Hip B1 y < maximum ys} \end{aligned}$$

  $\square$

Caso b12 : (e == y) = False

True = (e == y) || elem e ys $\equiv$ False || elem e ys

$\overset{Bool}{\equiv}$ elem e ys

Recordamos la HI : P(ys) = $\forall$e:: a elem e ys $\Rightarrow$ e $\leq$ maximum ys

$\overset{HI}{\equiv}$ True

$\square$

Caso b2 : y < maximum ys = False

(*) $\equiv$ e $\leq$ if False then (maximum ys) else y

$\overset{If}{\equiv}$ e $\leq$ y

Habíamos asumido el antecedente : True = elem e (y:ys) = (e == y) || elem e ys

Por lema de generación de bool, separamos en casos

Caso b21 : (e == y) = True

e = y $\Rightarrow$ e $\leq$ y

$\square$

Caso b22 : (e == y) = False

elem e ys = True $\Rightarrow$ e $\leq$ maximum ys

e $\leq$ maximum ys $\leq$ y

e $\leq$ y

$\square$

10 Vuelta de tuerca : length ys = length (reverse ys)

```haskell
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)

P(ys) := $\forall$ lenght ys = length (foldl (flip (:)) [] ys) Hacemos inducción

- Caso base : P([])

  P([]) = length [] $\overset{L0}{=}$ 0

  = length (fold (flip (:)) [] []) $\overset{F0}{=}$ length [] $\overset{L0}{=}$ 0

  = $\square$

- Paso inductivo : Queremos ver que vale P(y:ys) = length (y:ys) = length (foldl (flip (:)) [] (y:ys)). Planteamos la siguiente hipótesis inductiva

  HI : P(ys) = length ys = length (foldl (flip (:)) [] ys)

$$\texttt{length (foldl (flip (:)) [] (y:ys))} \overset{F1}{\equiv} \texttt{length (foldl (flip (:)) (flip (:) [] y) ys)}$$
$$\overset{FL}{\equiv} \texttt{length (foldl (flip (:)) [y] ys)}$$

Por otro lado, $\texttt{length (y:ys)} \overset{L1}{=} \texttt{1 + length ys}$
El problema es que suele suceder que con `foldl` no nos queda algo cómodo para aplicar la HI. Por tanto, con lo que tenemos no alcanza.

10.1 Generalizamos propiedades

Nosotros necesitamos entonces algo como :

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

La solución es, **demostrar una propiedad más general**. Probemos que

$$\texttt{∀ys :: [a].∀zs :: [a]. length zs + length ys = length (foldl (flip (:)) zs ys)}$$

Luego tomamos `zs = []` y sabiendo que `length [] = 0`, obtenemos lo que buscamos. Suponemos que vale el lema :

$$\texttt{∀ys :: [a]. length [] + length ys = length (foldl (flip (:)) [] ys)}$$

Con esto, `P(ys) = ∀ zs :: [a]. length zs + length ys = length (foldl (flip (:)) zs ys)` Hacemos inducción

- Caso base

$$\texttt{P([]) = ∀ zs :: [a]. length zs + length []} = \texttt{length (foldl (flip (:)) zs [])}$$
$$\overset{L0}{\equiv} \texttt{length zs} = \texttt{length (foldl (flip (:)) zs [])}$$
$$\overset{F0}{\equiv} \texttt{length zs} = \texttt{length zs}$$
$$\square$$

- Paso inductivo

  $$\texttt{∀ x :: a. P(xs)⇒ P(x:xs)}$$

  Nuestra hipótesis inductiva es :

  $$\texttt{P(xs) = ∀ zs :: [a]. length zs + length xs = length (foldl (flip (:)) zs xs)}$$

  Y queremos probar

  $$\texttt{P(x:xs) = ∀ zs :: [a]. length zs + length (x:xs) = length (foldl (flip (:)) zs (x:xs))}$$

  Sea `zs' :: [a]`,

  $$\texttt{length (foldl (flip (:)) zs' (x:xs))} \overset{F1}{\equiv} \texttt{length (foldl (flip (:)) (flip (:) zs' x) xs)}$$
  $$\overset{FL}{\equiv} \texttt{length (foldl (flip (:)) (x:zs') xs)} \qquad (*)$$

  $$\texttt{length zs' + length (x:xs))} \overset{F1}{\equiv} \texttt{length zs' + 1 + length xs)} \qquad (**)$$

  Tomando `zs = x : zs'` la Hi dice que `length (x:zs') + length xs = length (foldl (flip (:)) (x:zs') xs) (*)`
  $\texttt{length (x:zs') + length xs} \overset{L1}{=} \texttt{1 + length zs' + length xs (**)}$

11 Ejercicio : `take`

```
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)
```

Queremos probar que `take' = flipTake`.
Por extensionalidad basta ver que $\forall$`xs :: [a]. take' xs = flipTake xs`
Por inducción en listas, `P(xs) = take' xs = flipTake xs`.

- Caso base

`P([]) = take' [] =  flipTake []`

$$\texttt{take' []} \overset{\eta}{=} \texttt{\textbackslash n} \to \texttt{take' [] n}$$
$$\overset{T0}{=} \texttt{\textbackslash n} \to \texttt{[]}$$

$$\texttt{flipTake []} \overset{FT}{=} \texttt{foldr (\textbackslash x rec n} \to \texttt{if n == 0 then [] else x:rec(n-1)) (const []) []}$$
$$\overset{F0}{=} \texttt{const []}$$
$$\overset{C}{=} \texttt{(\textbackslash x} \to \texttt{\textbackslash\_} \to \texttt{x) []}$$
$$\overset{\beta}{=} \texttt{\textbackslash\_} \to \texttt{[]}$$
$$\square$$

- Paso inductivo
  `HI : P(ys) = take' ys = flipTake ys` queremos ver que `P(y:ys) = take' (y:ys) = flipTake (y:ys)`

$$\texttt{take' (y:ys)} \overset{\eta}{\equiv} \texttt{\textbackslash n} \to \texttt{(take' (y:ys) n)}$$
$$\overset{T1}{\equiv} \texttt{\textbackslash n} \to \texttt{(if n == 0 then [] else y:take' ys (n-1))}$$
$$\overset{HI}{\equiv} \texttt{\textbackslash n} \to \texttt{(if n == 0 then [] else y:flipTake ys (n-1))} \qquad (*)$$

$$\texttt{flipTake (y:ys)} \overset{FT}{\equiv} \texttt{foldr (\textbackslash x rec n} \to \texttt{if n == 0 then [] else x:rec(n-1)) (const []) (y:ys)}$$
$$\overset{F1}{\equiv} \texttt{(\textbackslash x rec n} \to \texttt{if n == 0 then [] else x:rec(n-1)) y}$$
$$\texttt{(foldr (\textbackslash x rec n} \to \texttt{if n == 0 then [] else x:rec(n-1)) (const []) (ys))}$$
$$\overset{FT}{\equiv} \texttt{(\textbackslash x rec n} \to \texttt{if n == 0 then [] else x:rec(n-1)) y (flipTake ys)}$$
$$\overset{\beta}{\equiv} \texttt{(\textbackslash rec n} \to \texttt{if n == 0 then [] else y:rec(n-1)) (flipTake ys)}$$
$$\overset{\beta}{\equiv} \texttt{(\textbackslash n} \to \texttt{if n == 0 then [] else y:(flipTake ys)(n-1))} \qquad (*)$$
$$\square$$

12 Demostrando propiedades sobre árboles

```
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 que

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

Por inducción sobre AB a :

$$\texttt{P(t) = cantNodos = length (inorder t)}$$

- Caso base

$$\begin{aligned}\texttt{P(Nil) = cantNodos Nil} &= \texttt{length (inorder Nil)}\\ \texttt{0} &\overset{CN0}{=} \texttt{length (inorder Nil)}\\ \texttt{length []} &\overset{L0}{=} \texttt{length (inorder Nil)}\\ \texttt{length (inorder Nil)} &\overset{I0}{=} \texttt{length (inorder Nil)}\end{aligned}$$

- Paso inductivo

$$\forall x :: \texttt{a}.\ \ (\texttt{P(i)} \wedge \texttt{P(d)}) \Rightarrow \texttt{P(Bin i x d)}$$

Definimos nuestras hipótesis inductivas :

$$\texttt{P(i)} \equiv \texttt{cantNodos i = length (inorder i)}$$
$$\texttt{P(d)} \equiv \texttt{cantNodos d = length (inorder d)}$$

Queremos ver que `P(Bin i x d) = cantNodos (Bin i x d) = length (inorder (Bin i x d))`

$$\begin{aligned}\texttt{cantNodos (Bin i x d)} &\overset{CN1}{=} \texttt{1 + cantNodos i + cantNodos d}\\ &\overset{HI}{=} \texttt{1 + length (inorder i) + length (inorder d)}\end{aligned}$$

$$\texttt{length (inorder (Bin i x d))} \overset{I1}{=} \texttt{length (inorder i ++ (x:inorder d))}$$

y ahora ? Necesitamos un lema !

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

Hay que probar el lema !

Demostramos 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)
```

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

13 Últimas preguntas

Si quisiéramos demostrar una propiedad sobre el tipo `Árbol123 a b` mediante inducción estructural:

```
data    Arbol23   a b =
           Hoja a
         | Dos b (   Arbol23   a b) (   Arbol23   a b)
         | Tres b b (   Arbol23   a b) (   Arbol23   a b) (   Arbol23   a b)
```

Para demostrar que vale :

$$\forall q :: \text{Árbol123 a b}.\ P(q)$$

Caso base

$$\forall\ x ::\ a.\ P(\text{Hoja } x)$$

Casos inductivos

$$\forall t1, t2 ::\ \text{Arbol123 a b}.\ \forall\ y::b.\ (P(t1) \land P(t2) \Rightarrow P(\text{Dos } y\ t1\ t2))$$

$$\forall t1, t2 , t3 ::\ \text{Arbol123 a b}.\ \forall\ y::b.\ \forall\ z ::\ b.\ (P(t1) \land P(t2) \land P(t3) \Rightarrow P(\text{Tres } y\ z\ t1\ t2\ t3))$$