

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

Anteriormente en PLP...

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

Hoy

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

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

Demostrando propiedades

Sean las siguientes definiciones:

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

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

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

Demostrando propiedades

Sean las siguientes 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
```

Demostrando propiedades

Sean las siguientes definiciones:

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

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

¿Cómo probamos que $\text{doble } 2 = \text{cuadrado } 2$?

Solución:

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

Igualdad de funciones

Queremos ver que:

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

Tenemos:

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

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

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

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

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

$(f . g) x = f (g x)$

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

$\text{id } x = x$

¿Cómo hacemos?

Extensionalidad

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:

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

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).

Algunas propiedades útiles

Estas son algunas propiedades que podemos usar en nuestras demostraciones:

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

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

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

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).

Algunas propiedades útiles

Estas son algunas propiedades que podemos usar en nuestras demostraciones:

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

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

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

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

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).

Algunas propiedades útiles

Estas son algunas propiedades que podemos usar en nuestras demostraciones:

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

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

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

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

$$\lambda x \rightarrow F x =_{\eta} F$$

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:

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

Tenemos:

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

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

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

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

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

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

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

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

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

Tenemos:

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

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

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

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

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

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

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

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

Por ppio. de extensiónalidad
veremos que

$$\forall x :: a \rightarrow b \rightarrow c.$$

$$(\text{curry} . \text{uncurry}) x = \text{id } x$$

$$\lambda x \rightarrow f x =_y f$$

sii $x \notin \text{FV}(f)$

$$(\text{curry} . \text{uncurry}) x = \underset{\text{COMP}}{\text{curry}} (\underline{\text{uncurry } x}) = \text{curry } (\lambda (z, y) \rightarrow x z y)$$

$$= \lambda j i \rightarrow (\lambda (z, y) \rightarrow x z y)(j, i) = \beta \lambda j i \rightarrow x j i$$

$$= \lambda j \rightarrow (\lambda i \rightarrow x j i) =_y \lambda j \rightarrow x j =_y x$$

\downarrow
sintaxis
de Haskell

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

✓

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?

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

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 entonces...

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

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)
```

Probemos entonces...

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

Tenemos:

$$\begin{aligned} \text{prod} &:: \text{Either Int (Int, Int)} \rightarrow \text{Either Int (Int, Int)} \rightarrow \text{Either Int (Int, Int)} \\ \{P0\} \text{ prod} (\text{Left } x) (\text{Left } y) &= \text{Left } (x * y) \\ \{P1\} \text{ prod} (\text{Left } x) (\text{Right } (y, z)) &= \text{Right } (x * y, x * z) \\ \{P2\} \text{ prod} (\text{Right } (y, z)) (\text{Left } x) &= \text{Right } (y * x, z * x) \\ \{P3\} \text{ prod} (\text{Right } (w, x)) (\text{Right } (y, z)) &= \text{Left } (w * y + x * z) \end{aligned}$$

$p :: \text{Either Int (Int, Int)}$

por el lema de generación de la suma: hay 2 casos.

(A) • $\exists k :: \text{Int}$ tal que $p = \text{Left } k$

(B) • $\exists l :: (\text{Int}, \text{Int})$ tal que $p = \text{Right } l$

 → por el lema de generación de pares

$\exists x :: \text{Int}, y :: \text{Int}$ tales que $l = (x, y)$

(C) • $\exists k' :: \text{Int} . q = \text{Left } k'$

(D) • $\exists x', y' :: \text{Int} . q = \text{Right } (x', y')$

Probemos entonces...

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

Tenemos:

$$\text{prod} :: \text{Either Int (Int, Int)} \rightarrow \text{Either Int (Int, Int)} \rightarrow \text{Either Int (Int, Int)}$$

$$\{P0\} \text{prod} (\text{Left } x) (\text{Left } y) = \text{Left } (x * y)$$

$$\{P1\} \text{prod} (\text{Left } x) (\text{Right } (y, z)) = \text{Right } (x * y, x * z)$$

$$\{P2\} \text{prod} (\text{Right } (y, z)) (\text{Left } x) = \text{Right } (y * x, z * x)$$

$$\{P3\} \text{prod} (\text{Right } (w, x)) (\text{Right } (y, z)) = \text{Left } (w * y + x * z)$$

caso (A) y (C)

$$\text{prod } p \ q = \text{prod} (\text{Left } k) (\text{Left } k') \stackrel{P0}{=} \text{Left } (k * k')$$

$$= \text{Left } (k' * k) \stackrel{\text{P0}}{=} \text{prod} (\text{Left } k') (\text{Left } k) = \text{prod } q \ p \quad \checkmark$$

* es
comunitativo

(faltan ver los otros 2).
caso

caso (B) y (C)

$$\text{prod } p \ q = \text{prod} (\text{Right } (x, y)) (\text{Left } k') \stackrel{P2}{=} \text{Right } (x * k', y * k')$$

$$= \text{Right } (k' * x, k' * y) \stackrel{P1}{=} \text{prod} (\text{Left } k') (\text{Right } (x, y)) \quad \checkmark$$

* comm.

$$= \text{prod } q \ p$$

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 $c \in e$ devuelve True si e pertenece a c y False en caso contrario.

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 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 e 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:

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

vacio :: Conj a

{V} vacio = _ -> 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 e 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:

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

Por ext. basta ver que $\forall x :: \text{a}$

$$\text{inter d (dif c d)} x = \text{vacío} x$$

$$\text{vacío} x = \begin{cases} (_) \rightarrow \text{False} \\ \vee \end{cases} x = \beta \text{ False}$$

$$\text{inter d } \underline{\text{(dif c d)}} x = \underbrace{\text{inter d } (\e \rightarrow c e \&\& \text{not (d e)})}_D x$$

$$= \underline{\underline{(\e' \rightarrow d e' \&\& (\e \rightarrow c e \&\& \text{not (d e)}) e')}} x$$

$$= \beta (\e' \rightarrow d e' \&\& (c e' \&\& \text{not (d e')})) x$$

$$= \beta d x \&\& (c x \&\& \text{not (d x)}) \stackrel{\text{por prop. Bool}}{=} \text{False} \&\& c x = \text{False}$$

Inducción en los naturales

- Pruebo $P(0)$

Inducción en los naturales

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

Inducción en listas

- Pruebo $P([])$

Inducción en listas

- 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).

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.

Pasos a seguir

- Leer la propiedad, entenderla y convencerse de que es verdadera.
- Plantear la propiedad como predicado unario.

Pasos a seguir

- Leer la propiedad, entenderla y convencerse de que es verdadera.
- Plantear la propiedad como predicado unario.
- Plantear el esquema de inducción.

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.

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)
```

```

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)

```

$$P(xs) = \text{length}_1 \times s = \text{length}_2 \times s$$

Para ver que vale $\forall xs : [a]. P(xs)$

- $P([])$

por inducción estructural, ver que vale:

- $\forall x : a. P(xs) \Rightarrow P(x:xs)$

- caso base

$$P([]) = \text{length}_1 [] = \text{length}_2 []$$

$$\text{length}_1 [] = 0$$

L10

$$\text{length}_2 [] = \text{foldr} (\lambda res \rightarrow 1 + res) 0 []$$

L2

$$= 0 \quad \checkmark$$

F1

Queremos ver $\text{length}_1 = \text{length}_2$

para pplo. de ext, queremos que

$$\forall xs : [a]. \text{length}_1 xs = \text{length}_2 xs$$


• Caso inductivo: $\forall x :: a. P(xs) \Rightarrow P(x:xs)$

$\text{length1} :: [a] \rightarrow \text{Int}$

{L10} $\text{length1} [] = 0$

{L11} $\text{length1} (_:xs) = 1 + \text{length1} xs$

$\text{length2} :: [a] \rightarrow \text{Int}$

{L2} $\text{length2} = \text{foldr } (\lambda _ \text{res} \rightarrow 1 + \text{res}) 0$

hip. ind.

H.i.

$$P(xs) \equiv (\text{length1 } xs = \text{length2 } xs)$$

$$\text{Sup } P(x:xs)$$

Recordemos:

$\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

{F0} $\text{foldr } f z [] = z$

{F1} $\text{foldr } f z (x:xs) = f x (\text{foldr } f z xs)$

$$P(x:xs) \equiv (\text{length1 } (x:xs) = \text{length2 } (x:xs))$$

$$\text{length1 } (x:xs) = 1 + \underbrace{\text{length1 } xs}_{L11}$$

$$\text{length2 } (x:xs) = \underbrace{\text{foldr } (\lambda _ \text{res} \rightarrow 1 + \text{res}) 0}_{L2} (x:xs)$$

$$= (\lambda _ \text{res} \rightarrow 1 + \text{res}) x \times (\underbrace{\text{foldr } (\lambda _ \text{res} \rightarrow 1 + \text{res}) 0}_{\text{length2}} xs)$$

$$= (\lambda _ \text{res} \rightarrow 1 + \text{res}) x (\text{length2 } xs) = \beta (\lambda \text{res} \rightarrow 1 + \text{res}) (\text{length2 } xs)$$

$$= \beta 1 + \text{length2 } xs \stackrel{H1}{=} 1 + \text{length1 } xs$$

$$= \lambda _ \rightarrow \lambda \text{res} \rightarrow 1 + \text{res}$$

✓

Demostrando implicaciones

Queremos probar que:

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

Demostrando implicaciones

Queremos probar que:

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

Antes que nada, ¿quién es P ?

¿En qué estructura vamos a hacer inducción?

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?

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

Demostrando implicaciones

Queremos probar que:

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

Antes que nada, ¿quién es P ?

¿En qué estructura vamos a hacer inducción?

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

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) = 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:

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

Seguimos en el pizarrón.

```

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:

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

Daremos ver que $\forall ys::[a]. P(ys)$.

Por inducción estructural en ys, veremos que

$$P([])$$

$$\underline{\forall x::a. P(xs) \Rightarrow P(x:xs)}$$

- caso base

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

$$\text{elem } e [] = \text{False}$$

E0

la implicación es válida.

- caso inductivo.

$$\forall y::a. \underbrace{P(y)}_{\text{H.i.}} \Rightarrow P(y:ys)$$

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

Queremos ver que vale $P(y:ys)$.

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

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

$\text{elem} :: \text{Eq } a \Rightarrow a \rightarrow [\text{a}] \rightarrow \text{Bool}$

{E0} $\text{elem } e [] = \text{False}$

{E1} $\text{elem } e (x:xs) = (e == x) \vee \text{elem } e xs$

$\text{maximum} :: \text{Ord } a \Rightarrow [\text{a}] \rightarrow a$

{M0} $\text{maximum } [x] = x$

{M1} $\text{maximum } (x:y:ys) = \text{if } x < \text{maximum } (y:ys) \text{ then } \text{maximum } (y:ys) \text{ else } x$

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

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

Sea $e :: a$ tal que $\text{True} = \text{elem } e (\text{y:ys})$.

Queremos ver que $e \leq \text{maximum } (\text{y:ys})$

Por el lema de generación de listas:

(A) $\text{ys} = []$

(B) $\exists z :: a, zs :: [\text{a}]$ tales que $\text{ys} = z:zs$

caso (A)

$$e \leq \text{maximum } (\text{y:ys}) \equiv e \leq \text{maximum } [y] \stackrel{\text{NO}}{\equiv} e \leq y$$

yo sé que $\text{True} = \text{elem } e (\text{y:ys})$, entonces en este caso $\text{True} = \text{elem } e ([])$

caso (A)

QVq: $e \leq \text{maximum } (y:ys) \equiv e \leq \text{maximum } [y] \stackrel{M0}{\equiv} e \leq y$
 $y \in \text{pue } \text{True} = \text{elem } e (y:ys)$, entonces en este caso $\text{True} = \text{elem } e (y:[])$

$\text{True} = \text{elem } e (y:[])$ $\stackrel{E1}{=} (e == y) \text{ || elem } e [] \stackrel{E0}{=} (e == y) \text{ || False}$
 $= (e == y)$ \rightsquigarrow entonces $e = y \Rightarrow$ en particular $e \leq y$

prop.
Bool

caso (B) QVq $e \leq \text{maximum } (y:ys) \equiv e \leq \text{maximum } (y:z:zs)$

$\equiv e \leq \text{if } y < \text{maximum } (z:zs) \text{ then maximum } (z:zs) \text{ else } y$
 $\stackrel{M1}{=}$ $e \leq \text{if } y < (\text{maximum } ys) \text{ then } (\text{maximum } ys) \text{ else } y$ $\stackrel{*}{=}$

Por el lemma de gen. de Bool,

(B1) $y < \text{maximum } ys$
 $\stackrel{*}{=} \text{True}$

(B2) $y < \text{maximum } ys$
 $\stackrel{*}{=} \text{false}$

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:

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

coso (B1)

~~(*)~~ = $e \leq y$ if True then maximum ys else $y \equiv e \leq \text{maximum } ys$
 \downarrow por si de Booleanos

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

Habiendo
anterior $\rightarrow \text{True} = \text{elem } e \text{ } (y:ys)$

$\text{True} = \text{elem } e \text{ } (y:ys) = (e == y) \parallel \text{elem } e \text{ } ys$

Por el lemma de generación de Bool

$$(B11) (e == y) = \text{True}$$

$$(B12) (e == y) = \text{False}$$

coso (B11)

$e = y$ entonces lo que queremos ver es que

~~(*)~~ $y \leq \text{maximum } ys$

por hip. (B1) vale

$y < \text{maximum } ys$

y por lo tanto

$y \leq \text{maximum } ys$

✓

$\text{elem} :: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}$

{E0} $\text{elem } e [] = \text{False}$

{E1} $\text{elem } e (x:xs) = (e == x) \parallel \text{elem } e xs$

$\text{maximum} :: \text{Ord } a \Rightarrow [a] \rightarrow a$

{M0} $\text{maximum } [x] = x$

{M1} $\text{maximum } (x:y:ys) = \text{if } x < \text{maximum } (y:ys) \text{ then } \text{maximum } (y:ys) \text{ else } x$

Sabemos que valen $\text{Eq } a$ y $\text{Ord } a$. Queremos ver que, para toda lista ys , vale:

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

Caso B12 ($e == y$) = False.

entonces $\text{True} = (e == y) \parallel \text{elem } e \text{ ys} = \text{False} \parallel \text{elem } e \text{ ys}$

ya lo sabíamos

$\Rightarrow \text{elem } e \text{ ys}$

Prop.
Bool

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

Por H.I., vale $e \leq \text{maximum } \text{ys}$. \otimes

Caso B2 $y < \text{maximum } \text{ys} = \text{False} \Rightarrow y \geq \text{maximum } \text{ys}$

$\otimes e \leq \text{if } \text{False} \text{ then } (\text{maximum } \text{ys}) \text{ else } y$

$\Rightarrow e \leq y$

por
implm. de if

$\text{True} = \text{elem } e \text{ (y:ys)} \text{ por HIP}$

$= (e == y) \parallel \text{elem } e \text{ ys}$

Lema de gen. de Bool:

- $(e == y) = \text{True} \Leftrightarrow e = y \Rightarrow e \leq y$ H.I.
- $(e == y) = \text{False} \Leftrightarrow \text{elem } e \text{ ys} = \text{True} \Rightarrow e \leq \text{maximum } \text{ys}$ (trans)
 $e \leq \text{maximum } \text{ys} \leq y \therefore e \leq y$

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

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.

```

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

$$\text{length} [] = 0$$

$$\text{length} (\text{foldl} (\text{flip} (:)) [] []) \stackrel{F0}{=} \text{length} [] = 0$$

• caso inductivo H.i. $P(ys) \equiv \text{length } ys = \text{length} (\text{foldl} (\text{flip} (:)) [] ys)$

$$\text{Queremos ver que vale } P(y:ys) \equiv \text{length } (y:ys) = \text{length} (\text{foldl} (\text{flip} (:)) [] (y:ys))$$

$$\begin{aligned} \text{length} (\text{foldl} (\text{flip} (:)) [] (y:ys)) &\stackrel{F1}{=} \text{length} (\text{foldl} (\text{flip} (:)) (\text{flip} (:) [] y) ys) \\ &= \text{length} (\text{foldl} (\text{flip} (:)) [y] ys) \end{aligned}$$

FL

$$\begin{aligned} P(ys) &\equiv \text{length } ys = \\ &\text{length} (\text{foldl} (\text{flip} (:)) [] ys) \end{aligned}$$

• caso base

$$\begin{aligned} P([]) &\equiv \text{length } [] \\ &= \text{length} (\text{foldl} (\text{flip} (:)) [] []) \end{aligned}$$

✓

$$\begin{aligned} [y] &= y : [] \\ &= (:) y [] \end{aligned}$$

por otra parte, $\text{length } (\text{y}: \text{ys}) = \begin{cases} 1 & \text{if } \text{y} \\ 1 + \text{length } \text{ys} & \text{otherwise} \end{cases}$

$$\begin{aligned}\text{length}(\text{foldl } (\text{flip } (:)) [] (\text{y}: \text{ys})) &\stackrel{\text{F1}}{=} \text{length}(\text{foldl } (\text{flip } (:)) (\text{flip } (:) \square \text{y}) \text{ ys}) \\ &= \text{length}(\text{foldl } (\text{flip } (:)) [\text{y}] \text{ ys})\end{aligned}$$

FL

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

No el comienza con esto

```
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: $\forall \text{ys} : [a]. \text{length } \text{ys} = \text{length}(\text{reverse } \text{ys})$
...que, por `reverse`, es lo mismo que:

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

Generalizando propiedades

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

Generalizando propiedades

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

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

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

Pero lo que necesitamos es:

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

Generalizando propiedades

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

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

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

Pero lo que necesitamos es:

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

¿Qué podemos hacer?

Generalizando propiedades

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

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

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

Pero lo que necesitamos es:

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

¿Qué podemos hacer?

Respuesta: demostrar una propiedad más general.

Generalizando propiedades

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

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

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

Pero lo que necesitamos es:

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

¿Qué podemos hacer?

Respuesta: demostrar una propiedad más general.

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

$$\text{length} (\text{foldl} (\text{flip} (\text{:})) \text{zs} \text{ ys})$$

Generalizando propiedades

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

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

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

Pero lo que necesitamos es:

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

¿Qué podemos hacer?

Respuesta: demostrar una propiedad más general.

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

$$\text{length} (\text{foldl} (\text{flip} (:)) \text{zs} \text{ ys})$$

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

Supongamos que vale el lemma.

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

dive es lo que queremos ver.

```

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

$$\text{length } zs + \text{length } [] = \text{length } zs$$

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

- caso base $\forall x :: a. P(xs) \Rightarrow P(x:xs)$

Hipótesis
Inductiva: $P(xs) = \forall zs :: [a]. \text{length } zs + \text{length } xs = \text{length} (\text{foldl} (\text{flip} (:)) zs xs)$

Queremos probar $P(x:xs) = \forall zs :: [a]. \text{length } zs + \text{length} (x:xs) = \text{length} (\text{foldl} (\text{flip} (:)) zs (x:xs))$

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

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

- Caso base

$$\begin{aligned} P([]) &= \forall zs :: [a]. \\ &\quad \text{length } zs + \text{length } [] \\ &= \text{length} (\text{foldl} (\text{flip} (:)) []) \end{aligned}$$

✓

Sea $zs' :: [a]$.

$\text{length} (\text{foldl} (\text{flip} (:)) zs' (x:xs))$,

$$\stackrel{F1}{=} \text{length} (\text{foldl} (\text{flip} (:)) (\text{flip} (:) zs' x) xs)$$

$$\stackrel{FL}{=} \text{length} (\text{foldl} (\text{flip} (:)) (x:zs') xs) \quad \text{(*)}$$

$$\text{length } zs' + \text{length} (x:xs) \stackrel{L1}{=} \text{length } zs' + 1 + \text{length } xs \quad \text{(**)}$$

$$\stackrel{H1}{=} P(xs) = \forall zs :: [a]. \text{length } zs + \text{length } xs = \text{length} (\text{foldl} (\text{flip} (:)) zs xs)$$

✓

Tomando $zs = x:zs'$

la H1 dice que

$\text{length} (x:zs') + \text{length } xs$

$$= \text{length} (\text{foldl} (\text{flip} (:)) (x:zs') xs) \quad \text{(*)}$$

$$\text{length} (x:zs') + \text{length } xs \stackrel{L1}{=}$$

$$1 + \text{length } zs' + \text{length } xs \quad \text{(**)}$$

$\text{length} :: [a] \rightarrow \text{Int}$

{L0} $\text{length} [] = 0$

{L1} $\text{length} (x:xs) = 1 + (\text{length } xs)$

$\text{foldl} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

{F0} $\text{foldl } f ac [] = ac$

{F1} $\text{foldl } f ac (x:xs) = \text{foldl } f (f ac x) xs$

$\text{reverse} :: [a] \rightarrow [a]$

{R} $\text{reverse} = \text{foldl} (\text{flip} (:)) []$

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

{FL} $\text{flip } f x y = f y x$

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

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

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

¿Recuerdan esta función?

~~take¹~~ o take⁴

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

¿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 $\text{take}' = \text{flipTake}$?

¿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 $\text{take}' = \text{flipTake}$.

```

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)

```

$$\text{take}' [] = \underset{n}{\forall} \lambda n \rightarrow \text{take}' [] n$$

$$\stackrel{\text{T0}}{=} \lambda n \rightarrow []$$

$$\begin{aligned} \text{flipTake} [] &= \underset{\text{FT}}{\text{foldr}} (\lambda x \text{rec } n \rightarrow \text{if } n==0 \text{ then } [] \text{ else } x : \text{rec} (n-1)) \\ &= \underset{\text{F0}}{\text{const}} [] = (\lambda x \rightarrow \underline{\lambda _ \rightarrow x}) [] = \underset{\text{C}}{\beta} \lambda _ \rightarrow [] \quad \checkmark \end{aligned}$$

Coinductive H.i.: $P(yS) = \text{take}' yS = \text{flipTake} yS$

Q.v. $P(y:yS) = \text{take}' (y:yS) = \text{flipTake} (y:yS)$

Ovemos ver que $\text{take}' = \text{flipTake}$
 Por extensiónalidad, basta ver que
 $\forall xs :: [a]. \text{take}' xs = \text{flipTake} xs$
 por ind-extnctual en todos.

$$P(xs) \equiv \text{take}' xs = \text{flipTake} xs$$

o caso base:

$$P([]) \equiv \text{take}' [] = \text{flipTake} []$$

$$f = \underset{n}{\forall} \lambda x \rightarrow fx$$

• Co-inductive H.i.: $P(\text{ys}) = \text{take}' \text{ ys} = \text{flipTake} \text{ ys}$

Prov $P(y:\text{ys}) = \text{take}'(y:\text{ys}) = \text{flipTake}(y:\text{ys})$

$\text{take}'(y:\text{ys}) =_m \lambda n \rightarrow (\text{take}'(y:\text{ys}) n)$

$\stackrel{T1}{=} \lambda n \rightarrow (\text{if } n == 0 \text{ then [] else } y : \underline{\text{take}' \text{ ys}}(n-1))$

$\stackrel{H1}{=} \lambda n \rightarrow (\text{if } n == 0 \text{ then [] else } y : \underline{\text{flipTake} \text{ ys}}(n-1)) \quad \otimes$

$\text{flipTake}(\text{ys}) =_{FT} \text{foldr } (\lambda x \text{ rec } n \rightarrow \text{if } n == 0 \text{ then [] else } x : \text{rec}(n-1))$
 $(\text{const} \text{ []}) (\text{ys})$

$F1 = (\lambda x \text{ rec } n \rightarrow \text{if } n == 0 \text{ then [] else } x : \text{rec}(n-1)) \text{ ys}$

$(\text{foldr } (\lambda x \text{ rec } n \rightarrow \text{if } n == 0 \text{ then [] else } x : \text{rec}(n-1))$
 $(\text{const} \text{ []}) \text{ ys})$

$\stackrel{FT}{=} (\lambda x \text{ rec } n \rightarrow \text{if } n == 0 \text{ then []})$
 $\text{else } x : \text{rec}(n-1) \text{ ys} (\text{flipTake} \text{ ys})$

$=_{\beta} (\lambda \text{rec } n \rightarrow \text{if } n == 0 \text{ then []})$
 $\text{else } y : \text{rec}(n-1) (\text{flipTake} \text{ ys})$

$=_{\beta} \lambda n \rightarrow \text{if } n == 0 \text{ then [] else } \underline{y : (\text{flipTake} \text{ ys})(n-1)} \quad \otimes$

$\text{take}' :: [a] \rightarrow \text{Int} \rightarrow [a]$

{T0} $\text{take}' [] _ = []$

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

$\text{flipTake} :: [a] \rightarrow \text{Int} \rightarrow [a]$

{FT} $\text{flipTake} = \text{foldr}$

$(\lambda x \text{ rec } n \rightarrow \text{if } n == 0 \text{ then [] else } x : \text{rec}(n-1))$
 $(\text{const} \text{ []})$

$\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

{F0} $\text{foldr } f z [] = z$

{F1} $\text{foldr } f z (x:\text{xs}) = f x (\text{foldr } f z \text{ xs})$

$\text{const} :: (a \rightarrow b \rightarrow a)$

{C} $\text{const} = (\lambda x \rightarrow \underline{_} \rightarrow x)$

Demostrando propiedades sobre árboles

$$AB \alpha = Nil \mid Bin(AB \alpha) \alpha (AB \alpha)$$

Dadas las siguientes funciones:

```
cantNodos :: AB a -> Int
{C0} cantNodos Nil = 0
{C1} 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:

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

```

cantNodos :: AB a -> Int
{CNO} cantNodos Nil = 0
{CN1} cantNodos (Bin i r d) = 1 + (cantNodos i) + (cantNodos d)

inorder :: AB a -> [a]
{IO} 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:

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

$$\text{cantNodos } Nil = 0 = \text{length } [] = \text{length}(\text{inorder } Nil)$$

CNO L0 IO



- caso base

$$P(Nil) \equiv \text{cantNodos } Nil$$

$$\text{length} \stackrel{=} {(\text{inorder } Nil)}$$

- caso inducción $\forall x :: a . (P(i) \wedge P(d)) \Rightarrow P(\text{Bin } i \times d)$

Hip. ind. $P(i) \equiv \text{cantNodos } i = \text{length}(\text{inorder } i)$

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

Sup. $P(\text{Bin } i \times d) = \text{cantNodos } (\text{Bin } i \times d) = \text{length}(\text{inorder } (\text{Bin } i \times d))$

$$\begin{aligned} \text{cantNodos } (\text{Bin } i \times d) &= 1 + \text{cantNodos } i + \text{cantNodos } d \\ &\stackrel{CNI}{=} 1 + \text{length}(\text{inorder } i) + \text{length}(\text{inorder } d) \end{aligned}$$

$$\text{length}(\text{inorder } (\text{Bin } i \times d)) \stackrel{I1}{=} \text{length}(\text{inorder } i) + \text{length}(\text{inorder } (\text{Bin } i \times d))$$

Por inducción sobre AB a:

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

¿Y ahora qué hacemos?

¡Necesitamos un lema!

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

Queremos probar:

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

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

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

Últimas preguntas

Si quisieramos demostrar una propiedad sobre el tipo $\text{Árbol23} \ a \ b$ 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)?

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

caso de dos $\left\{ \begin{array}{l} \bullet \forall t_1, t_2 :: \text{Árbol23} \ a \ b . \forall y :: b . (P(t_1) \wedge P(t_2)) \Rightarrow P(\text{Dos } y \ t_1 \ t_2) \\ \bullet \forall t_1, t_2, t_3 :: \text{Árbol23} \ a \ b . \forall y :: b . \forall z :: b . \end{array} \right.$

caso de tres $(P(t_1) \wedge P(t_2) \wedge P(t_3)) \Rightarrow P(\text{Tres } y \ z \ t_1 \ t_2 \ t_3)$

Fin

i i i i i i ? ? ? ? ? ?