Práctica 2: Razonamiento Ecuacional e Inducción Estructural

Tomás Felipe Melli

April 14, 2025

Índice

1 Extensionalidad y Lemas de Generación

1.1 Ejercicio 1

```
intercambiar (x,y) = (y,x)           -{I}
espejar (Left x) = Right x           -{E1}
espejar (Right x) = Left x           -{E2}
asociarI (x,(y,z)) = ((x,y),z)       -{AI}
asociarD ((x,y),z)) = (x,(y,z))      -{AD}
flip f x y = f y x                   -{F}
curry f x y = f (x,y)                -{C}
uncurry f (x,y) = f x y              -{U}
```

Nos piden demostrar :

1. $\forall p :: (a,b).intercambiar\ (intercambiar\ p) = p$. Esta expresión está trabada, por tanto, recurrimos al lema de generación de pares que dice que *si* $\forall x :: a.\forall y :: b.P((x,y))$ *entonces* $\forall p :: (a,b).P(p)$. Con esto en mente, destrabamos y ..

$$intercambiar\ (intercambiar\ p) = p$$

$$\overset{\textit{Lema generación pares}}{\equiv} intercambiar\ (intercambiar\ (x,y)) = (x,y)$$

$$\overset{I}{\equiv} intercambiar\ (y,x) = (x,y)$$

$$\overset{I}{\equiv} (x,y) = (x,y)$$

2. $\forall p :: (a,(b,c)).asociarD\ (asociarI\ p) = p$. Como antes, está trabada. Por el principio de inducción sobre pares podemos expresar a que la propiedad vale sobre la tupla y por tanto...

$$asociarD\ (asociarI\ p) = p$$

$$\overset{\textit{Lema generación pares}}{\equiv} asociarD\ (asociarI\ (x,y)) = (x,y)$$

Sigue trabado el y, por tanto, aplicamos de nuevo el lema.

$$\overset{\textit{Lema generación pares}}{\equiv} asociarD\ (asociarI\ (x,(i,j)) = (x,(i,j))$$

$$\overset{AI}{\equiv} asociarD\ ((x,i),j) = (x,(i,j))$$

$$\overset{AD}{\equiv} (x,(i,j)) = (x,(i,j))$$

3. $\forall p :: Either\ a\ b.\ espejar\ (espejar\ p) = p$. Esta expresión está trabada. Podemos utilizar el *lema de generación para Sumassi* $p :: Either\ a\ b$ *entonces* :

   - o bien $\exists x :: a.p = Left\ x$
   - o bien $\exists y :: b.p = Rigth\ y$

   Con esto en mente, separamos en casos :

   - $$\overset{Lema}{\equiv} espejar\ (espejar\ (Left\ x)) = (Left\ x)$$

     $$\overset{E1}{\equiv} espejar\ (Right\ x) = (Left\ x)$$

     $$\overset{E2}{\equiv} (Left\ x) = (Left\ x)$$

   - $$\overset{Lema}{\equiv} espejar\ (espejar\ (Right\ y)) = (Rigth\ y)$$

     $$\overset{E2}{\equiv} espejar\ (Left\ y) = (Rigth\ y)$$

     $$\overset{E1}{\equiv} (Rigth\ y) = (Rigth\ y)$$

4. $\forall f :: a- > b- > c.\forall x :: a.\forall y :: b.\ flip\ (flip\ f)\ x\ y = f\ x\ y$. En este caso, queremos demostrar algo sobre el comportamiento y no sobre la representación interna, por tal motivo tenemos que pensar en **Extensionalidad**.

$$flip\ (flip\ f)\ x\ y = f\ x\ y$$

$$\stackrel{F}{\equiv} flip\ f\ y\ x = f\ x\ y$$

$$\stackrel{F}{\equiv} f\ x\ y = f\ x\ y$$

5. $\forall f :: a- > b- > c.\forall x :: a.\forall y :: b.\ curry\ (uncurry\ f)\ x\ y = f\ x\ y$. Tenemos que probar cosas sobre el comportamiento de ambas y por ello, utilizamos el principio de extensionalidad para probar esta igualdad.

$$curry\ (uncurry\ f)\ x\ y = f\ x\ y$$

$$\stackrel{C}{\equiv} uncurry\ f\ (x, y) = f\ x\ y$$

$$\stackrel{U}{\equiv} f\ x\ y = f\ x\ y$$

1.2 Ejercicio 2

1. $flip\ .\ flip = id$. Para demostrar esta igualdad, seguimos el principio de Extensionalidad funcional que nos dice que este problema se reduce a resolver algo del tipo $f = g$. Para ello, nos proponemos $\forall x :: (a- > b- > c)- > b- > a- > c$ y

$$flip\ .\ flip\ x = idx$$

$$\stackrel{(.)}{\equiv} (flip\ .\ flip)\ x = id\ x$$

$$\stackrel{(.)}{\equiv} flip\ (flip\ x) = id\ x$$

$$\stackrel{por\ ejercicio\ 1.5}{\equiv} x = id\ x$$

$$\stackrel{id}{\equiv} id\ x = id\ x$$

2. $\forall f :: (a, b)- > c.\ uncurry\ (curry\ f) = f$ Definimos $\forall x :: a.\forall y :: b$

$$\equiv uncurry\ (curry\ f)\ (x, y) = f\ (x, y)$$

$$\stackrel{uncurry}{\equiv} curry\ f\ x\ y = f\ (x, y)$$

$$\stackrel{curry}{\equiv} f\ (x, y) = f\ (x, y)$$

3. $flip\ const = const\ id$ Podemos definir $\forall x :: a.\forall y :: b$ y tomar entonces:

$$flip\ const\ x\ y = const\ idx\ y$$

$$\stackrel{flip}{\equiv} const\ y\ x = const\ id\ x\ y$$

$$\stackrel{const}{\equiv} y = const\ id\ x\ y$$

$$\stackrel{const}{\equiv} y = id\ x\ \ - -chequear$$

4. $\forall f :: a- > b.\forall g :: b- > c.\forall h :: c- > d.$ queremos ver que :

$$((h\ .\ g)\ .\ f) = (h\ .(g\ .\ f))$$

Por principio de extensionalidad funcional bastaría con ver que $\forall x :: a$:

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

$$\stackrel{(.)}{\equiv} (h\ .\ g)\ (f\ x) = h\ ((g\ .f)\ x)$$

$$\stackrel{(.)}{\equiv} h\ (g\ (f\ x)) = h\ ((g\ (f\ x)$$

2 Inducción sobre Listas

2.1 Ejercicio 3

Se tienen las siguientes funciones:

```haskell
length :: [a] -> Int
{L0} length [] = 0
{L1} length (x:xs) = 1 + length xs

duplicar :: [a] -> [a]
{D0} duplicar [] = []
{D1} duplicar (x:xs) = x : x : duplicar xs

(++) :: [a] -> [a] -> [a]
{++0} [] ++ ys = ys
{++1} (x:xs) ++ ys = x : (xs ++ ys)

append :: [a] -> [a] -> [a]
{A0} append xs ys = foldr (:) ys xs

ponerAlFinal :: a -> [a] -> [a]
{P0} ponerAlFinal x = foldr (:) (x:[])

reverse :: [a] -> [a]
{R0} reverse = foldl (flip (:)) []
```

1. $\forall xs :: [a].$

$$length\ (duplicar\ xs) = 2 \times length\ xs$$

Queremos demostrar cierta propiedad sobre listas, por ello, miremos la definición del tipo :

```
data [a] = [] | a : [a]
```

Tenemos que probar que la propiedad P vale para $P[\,]$ y que también vale $\forall x :: a.\forall xs :: [a].(P(xs) \implies P(x : xs))$. Por inducción estructural sobre listas decimos que

- Caso base : $P[\,]$ vale :

$$length\ (duplicar\ [\,]) = 2 \times length\ [\,]$$

$$\overset{D0-L0}{\equiv} length\ [\,] = 2 \times 0$$

$$\overset{L0}{\equiv} 0 = 0$$

- Caso inductivo : tomamos como **H.I**

$$P(xs) \equiv (length\ (duplicar\ xs) = 2 \times length\ xs)$$

Entonces, como tomamos la hipótesis como verdadera, pasamos a probar que vale la propiedad para

$$P(x : xs) \equiv length\ (duplicar\ (x : xs)) = 2 \times length\ (x : xs)$$

$$length\ (duplicar\ (x : xs))$$

$$\overset{D1}{\equiv} length\ (x : x : duplicar\ xs)$$

$$\overset{L1}{\equiv} 1 + length\ (x : duplicar\ xs)$$

$$\overset{L1}{\equiv} 1 + 1 + length\ (duplicar\ xs)$$

$$\overset{HI}{\equiv} 1 + 1 + (2 \times length\ xs)$$

$$\overset{(*)}{\equiv} 1 + 1 + (length\ xs + length\ xs)$$

$$\overset{(+)}{\equiv} (1 + length\ xs) + (1 + length\ xs)$$

$$\overset{L1}{\equiv} (length\ (x : xs)) + (length\ (x : xs))$$

$$\equiv 2 \times length\ (x : xs)$$

2. $\forall xs :: [a].\forall ys :: [a]$. Queremos probar que

$$length\ (xs ++ ys) = length\ xs + length\ ys$$

Para ello, tenemos que utilizar inducción estructural sobre el tipo :

- Vale la propiedad para $P([\,])$

$$length\ ([\,] ++ [\,]) = length\ [\,] + length\ [\,]$$
$$\stackrel{++0}{\equiv} length\ ([\,])$$
$$\stackrel{L0}{\equiv} 0$$
$$\stackrel{L0}{\equiv} length[\,]$$
$$\stackrel{L0}{\equiv} length[\,] + length[\,]$$

- Queremos ver que $\forall x :: a.P(xs) \implies P(x : xs)$ asumiendo como **HI**

$$P(xs) \equiv length\ (xs ++ ys) = length\ xs + length\ ys$$

  Queremos probar que :

$$length\ ((x : xs) ++ ys) = length\ (x : xs) + length\ ys$$
$$\stackrel{++1}{\equiv} length\ (x : (xs ++ ys)$$
$$\stackrel{L1}{\equiv} 1 + length\ (xs ++ ys)$$
$$\stackrel{HI}{\equiv} 1 + length\ xs + length\ ys$$
$$\stackrel{L1}{\equiv} length\ (x : xs) + length\ ys$$

3. $\forall xs :: [a].\forall x :: a$. Queremos ver que valga

$$append\ [x]\ xs = x : xs$$
$$\stackrel{A0}{\equiv} foldr\ (:)\ xs\ [x]$$
$$\stackrel{re-escritura}{\equiv} foldr\ (:)\ xs\ (x : [\,])$$
$$\stackrel{foldr\ caso\ recursivo}{\equiv} (:)\ x\ (foldr\ (:)\ xs\ [\,])$$
$$\stackrel{foldr\ caso\ base}{\equiv} (:)\ x\ xs$$
$$\equiv x\ (:)\ xs$$

Sale directo (**??**)

4. $\forall xs :: [a].\forall f :: (a- > b)$. Queremos probar que

$$length\ (map\ f\ xs) = length\ xs$$

Podemos pensar en demostrarlo con inducción sobre el tipo, entonces postulamos :

(a) Caso Base : la propiedad vale para ... $P([\,])$

$$length\ (map\ f\ [\,]) = length\ [\,]$$
$$\stackrel{map-caso\ base}{\equiv} length\ [\,] = length\ [\,]$$

(b) Caso Recursivo, queremos ver que si la propiedad vale para cierta lista, también vale para esa lista más un elemento. $P(xs) \implies P(x : xs)$

$$length\ (map\ f\ (x : xs)) = length\ (x : xs)$$
$$\stackrel{map}{\equiv} length\ (f\ x :\ map\ f\ xs)$$
$$\stackrel{L1}{\equiv} 1 + length\ (map\ f\ xs)$$
$$\stackrel{H.I}{\equiv} 1 + length\ xs$$
$$\stackrel{L1}{\equiv} length\ (x : xs)$$

5. $\forall xs :: [a].\forall p :: a-> Bool.\forall e :: a$. Queremos ver que vale la siguiente igualdad (asumiendo que Eq =¿ a)

$$((elem\ e\ (filter\ p\ xs)) \implies (elem\ e\ xs))$$

Para demostrarlo, lo vamos a hacer sobre inducción estructural sobre el tipo.

(a) Caso Base : $P([\,])$

$$((elem\ e\ (filter\ p\ [\,])) \implies (elem\ e\ [\,]))$$

$$\overset{filter-caso\ base}{\equiv} ((elem\ e\ [\,]) \implies (elem\ e\ [\,]))$$

$Falso \implies Verdadero$

(b) Caso Inductivo: $P(xs) \implies P(x : xs)$

$$((elem\ e\ (filter\ p\ (x : xs))) \implies (elem\ e\ (x : xs)))$$

$$\overset{filter}{\equiv} (elem\ e\ (if\ p\ x\ then\ x : (filter\ p\ xs)\ else\ (filter\ p\ xs)$$

El tema es que tenemos **dos casos**, si el elemento x cumple el predicado p o si no lo cumple. Para destrabar esto, usamos el **principio de inducción sobre Bool**

i. Tomamos que $p\ x\ es\ True$

$$\overset{filter}{\equiv} (elem\ e\ (x : (filter\ p\ xs)))$$

Pero volvemos a tener problemas, je :). Porque puede ser que x sea e o no, y e se encuentre en xs. Otra vez, separamos los casos

- **Si e == x** esto implica que efectivamente $elem\ e\ (x : xs)$.
- **Si e** $\in xs$ vale por H.I

ii. Tomamos que $p\ x\ es\ False$

$$\overset{filter}{\equiv} (elem\ e\ (filter\ p\ xs))$$

Que por H.I esto vale.

6. $\forall xs :: [a].\forall x :: a$. Queremos probar que

$$ponerAlFinal\ x\ xs = xs ++ (x : [\,])$$

La idea es análoga a lo que venimos haciendo. Inducción sobre el tipo.

(a) $P([\,])$

$$ponerAlFinal\ x\ [\,] = [\,] ++ (x : [\,])$$

$$\overset{P0}{\equiv} foldr\ (:)\ (x : [\,])\ [\,]$$

$$\overset{foldr\ caso\ base}{\equiv} (x : [\,])$$

$$\equiv [x]$$

$$\overset{++0}{\equiv} [\,] ++ (x : [\,])$$

(b) $P(xs) \implies P(x : xs)$

$$ponerAlFinal\ y\ (x : xs) = (x : xs) ++ (y : [\,])$$

$$\equiv ponerAlFinal\ y\ (x : xs)$$

$$\overset{P0}{\equiv} foldr\ (:)\ (y : [\,])\ (x : xs)$$

$$\overset{foldr}{\equiv} (:)\ x\ (foldr\ (:)\ (y : [\,])\ xs)$$

$$\overset{P0}{\equiv} (:)\ x\ (ponerAlFinal\ y\ xs)$$

$$\overset{HI}{\equiv} (:)\ x\ (xs ++ (y : [\,]))$$

$$\overset{++1}{\equiv} (x : xs) ++ (y : [\,]))$$

7.
$$reverse = foldr\ (x\ rec -> rec ++ (x : [\,])\ [\,])$$

Por principio de extensionalidad funcional queremos ver que $\forall xs :: [a]$

$$reverse\ xs = foldr\ (x\ rec -> rec ++ (x : [\,])\ [\,])\ xs$$

Primero queremos ver que valga esta propiedad para el caso base, ya que hacemos inducción estructural sobre el tipo.

(a) $P([\,])$

$$reverse\ [\,] = foldr\ (x\ rec -> rec ++ (x : [\,])\ [\,])\ [\,]$$
$$\overset{foldr\ caso\ base}{\equiv} reverse\ [\,] = [\,]$$
$$\overset{R0}{\equiv} foldl\ (flip\ (:))\ [\,]\ [\,] = [\,]$$
$$\overset{foldl\ caso\ base}{\equiv} [\,] = [\,]$$

(b) $P(xs) \implies P(x : xs)$

$$reverse\ (x : xs) = foldr\ (y\ rec -> rec ++ (y : [\,])\ [\,])\ (x : xs)$$
$$\overset{R0}{\equiv} foldl\ (flip\ (:))[\,](x : xs)$$
$$\overset{foldl}{\equiv} foldl\ (flip\ (:))\ ((flip\ (:))\ [\,]\ x)\ xs$$

(c) $P(xs) \implies P(x : xs)$ Del otro lado porque me trabé.

$$reverse\ (x : xs) = foldr\ (y\ rec -> rec ++ (y : [\,]))\ [\,])\ (x : xs)$$
$$\overset{foldr}{\equiv} (y\ rec -> rec ++ (y : [\,]))\ x\ (foldr\ (y\ rec -> rec ++ (y : [\,])\ [\,])\ xs)$$
$$\overset{HI}{\equiv} (y\ rec -> rec ++ (y : [\,]))\ x\ (reverse\ xs)$$
$$\overset{\beta}{\equiv} (rec -> rec ++ (x : [\,]))\ (reverse\ xs)$$
$$\overset{\beta}{\equiv} reverse\ xs ++ (x : [\,])$$

8. $\forall xs :: [a].\forall x :: a$
$$head\ (reverse\ (ponerAlFinal\ x\ xs)) = x$$

2.2 Ejercicio 4

1. Queremos demostar la suguiente igualdad $reverse\ .\ reverse = id$ Por principio de de extensionalidad funcional, bastaría con probar que $\forall xs :: [a] \quad reverse\ .\ reverse\ xs = id\ xs$ Que podremos demostrar mediante el uso de inducción estructural sobre listas.
$$reverse\ .\ reverse\ xs = id\ xs$$

- Caso Base : $P([\,])$
$$reverse\ .\ reverse\ [\,] = id\ [\,]$$
$$\overset{(.)}{\equiv} reverse\ (reverse\ [\,])$$
$$\overset{R0}{\equiv} reverse\ (foldl\ (flip\ (:))\ [\,]\ [\,])$$
$$\overset{foldl}{\equiv} reverse\ [\,]$$
$$\overset{R0}{\equiv} (foldl\ (flip\ (:))\ [\,]\ [\,])$$
$$\overset{foldl}{\equiv} [\,]$$
$$\overset{id}{\equiv} id\ [\,]$$

- $P(xs) \implies P(x : xs)$
$$reverse\ .\ reverse\ (x : xs) = id\ (x : xs)$$
$$\overset{(.)}{\equiv} reverse\ (reverse\ (x : xs))$$
$$\overset{R0}{\equiv} reverse\ (foldl\ (flip\ (:))\ [\,]\ (x : xs))$$
$$\overset{foldl}{\equiv} reverse\ (foldl\ (flip\ (:))\ ((flip\ (:))\ [\,]\ x)\ xs$$

2.3 Ejercicio 5

```
zip :: [a] -> [b] -> [(a,b)]
{Z0} zip = foldr (\x rec ys -> if null ys then [] else (x, head ys) : rec (tail ys)) (const [])

zip' :: [a] -> [b] -> [(a,b)]
{Z0'} zip' [] ys = []
{Z1'} zip' (x:xs) ys = if null ys then [] else (x, head ys):zip' xs (tail ys)
```

Queremos probar que $zip = zip'$
Para ello, por principio de extensionalidad bastaría probar que $\forall xs, ys :: [a]. \quad zip\ xs\ ys = zip'\ xs\ ys$. Para probar esta igualdad hacemos uso de inducción estructural sobre listas. Y por ello planteamos

1. Caso Base : $P([\,])$

$$zip\ [\,]\ ys = zip'\ [\,]\ ys$$

$$\stackrel{Z0}{\equiv} foldr\ (i\ rec\ is- > if\ null\ is\ then\ [\,]\ else\ (i, head\ is) : rec\ (tail\ is))\ (const[\,])\ [\,]\ ys$$

$$\stackrel{foldr}{\equiv} const\ [\,]\ ys$$

$$\stackrel{const}{\equiv} [\,]$$

$$\stackrel{Z0'}{\equiv} zip'\ [\,]\ ys$$

2. $\forall x :: a \quad P(xs) \implies P(x : xs)$. Queremos probar que vale

$$zip\ (x : xs)\ ys = zip'\ (x : xs)\ ys$$

$$\equiv zip\ (x : xs)\ ys$$

$$\stackrel{Z0}{\equiv} foldr\ (i\ rec\ is- > if\ null\ is\ then\ [\,]\ else\ (i, head\ is) : rec\ (tail\ is))\ (const[\,])\ (x : xs)\ ys$$

$$\stackrel{foldr}{\equiv} (i\ rec\ is- > if\ null\ is\ then\ [\,]\ else\ (i, head\ is) : rec\ (tail\ is))\ x$$

$$(foldr\ (i\ rec\ is- > if\ null\ is\ then\ [\,]\ else\ (i, head\ is) : rec\ (tail\ is))\ (const[\,])\ xs)\ ys$$

$$\stackrel{Z0}{\equiv}$$

2.4 Ejercicio 6

```
nub :: Eq a => [a] -> [a]
{N0} nub [] = []
{N1} nub (x:xs) = x : filter (\y -> x /= y) (nub xs)

union :: Eq a => [a] -> [a] -> [a]
{U0} union xs ys = nub (xs++ys)

intersect :: Eq a => [a] -> [a] -> [a]
{I0} intersect xs ys = filter (\e -> elem e ys) xs
```

Tenemos que indicar si las siguientes propiedades son verdaderas o falsas. En caso de ser verdaderas, probarlas ; caso contrario, contraejemplo.

1. $Eq\ a => \forall xs :: [a].\forall e :: a.\forall p :: a- > Bool. \quad elem\ e\ xs\ \&\&\ p\ e = elem\ e\ (filter\ p\ xs)$ Es verdadero, si el elemento está en la lista y cumple con el predicado, si construimos una lista con filter aplicando ese predicado, efectivamente va a estar en la lista resultante el elemento. La demo es muy similar al ejercicio 3.5.

2. $Eq\ a => \forall xs :: [a].\forall e :: a. \quad elem\ e\ xs = elem\ e\ (nub\ xs)$
Qué hace *nub* ? elimina las apariciones repetidas de cada elemento de la lista que recibe como parámentro. Por tanto, la igualdad se cumple, dado que el elemento está en la lista y por tanto, aparece al menos una vez. Como consecuencia, estará en la lista generada por *nub*. Queremos ver entonces que vale. Y por ello, por inducción estructural sobre listas...

$$P(xs) \equiv elem\ e\ xs = elem\ e\ (nub\ xs)$$

- Caso Base $P([\,])$

$$elem\ e\ [\,] = elem\ e\ (nub\ [\,])$$
$$\equiv elem\ e\ [\,]$$
$$\overset{elem}{\equiv} False$$
$$\overset{elem}{\equiv} elem\ [\,]$$
$$\overset{N0}{\equiv} elem\ (nub\ [\,])$$

- Caso Inductivo $P(xs) \implies P(x:xs) \quad \forall x :: a$

$$elem\ e\ (x:xs) = elem\ e\ (nub\ (x:xs))$$
$$\equiv elem\ e\ (x:xs)$$
$$\overset{elem}{\equiv} e == x\ ||\ elem\ e\ xs$$

Esto nos lleva a **dos casos**. Que por principio de inducción sobre booleanos ...

- Si **e == x**
    - Si **True**

$$\equiv True\ ||\ elem\ e\ xs$$
$$\equiv True$$
$$\overset{elem}{\equiv} elem\ e\ (nub\ (x:xs))$$

    - Si **False**

$$\equiv False\ ||\ elem\ e\ xs$$
$$\equiv elem\ e\ xs$$

y por tanto dependerá de...

$$\equiv elem\ e\ xs$$
$$\overset{HI}{\equiv} elem\ e\ (nub\ xs)$$

3. $Eq\ a => \forall xs :: [a].\forall ys :: [a].\forall e :: a. \quad elem\ e\ (union\ xs\ ys) = (elem\ e\ xs)\ ||\ (elem\ e\ ys)$. En castellano es, el elemento e pertenece a la unión sin repetidos entonces o pertenece a xs o pertenece a ys (no es un $\vee$ exclusivo ya que puede estar en las dos pero no estará en la filtrada). Lo cual es cierto.

4. $Eq\ a => \forall xs :: [a].\forall ys :: [a].\forall e :: a. \quad elem\ e\ (intersect\ xs\ ys) = (elem\ e\ xs)\&\&(elem\ e\ ys)$. Nos dice que si un elemento $e \in xs \wedge e \in ys$ en la lista resultante estará. Queremos demostrar que debe estar en ambas.

5. $Eq\ a => \forall xs :: [a].\forall ys :: [a]. \quad length\ (union\ xs\ ys) = length\ xs + length\ ys$. Esta expresión dice que la longitud de la **union(U0)** de dos listas es la misma que la suma de las longitudes de casa lista. Con esta definición de Unión, esto es falso si hay elementos en común.

$$Sea\ xs = [1,2,3]\ y\ ys = [1,2,3]$$
$$length\ (union\ xs\ ys) = length\ xs + length\ ys$$
$$\overset{U0}{\equiv} length\ (nub\ (xs ++ ys)) = length\ xs + length\ ys$$
$$\overset{N1}{\equiv} length\ (nub\ (xs ++ ys)) = length\ xs + length\ ys$$

Que en este ejemplo sabemos que $[1,2,3] ++ [1,2,3] = [1,2,3,1,2,3]$ y que si hacemos $nub\ [1,2,3,1,2,3] = [1,2,3]$...

$$\equiv length\ [1,2,3] = length\ [1,2,3] + length\ [1,2,3]$$
$$\equiv 3 = 6$$

Lo cual es **absurdo**.

6. $Eq\ a => \forall xs :: [a].\forall ys :: [a]. \quad length\ (union\ xs\ ys) \leq length\ xs + length\ ys$. Lo que propone la expresión es que la longitud de la lista resultante de la unión sin repetidos es menor o igual a la suma de la listas que se unirán. Existen dos caso que tal vez tengan relevancia analizar : el anterior, listas iguales, sabemos que vale, que es mayor la suma de las longitudes. Y el caso en que son diferentes, como $xs = [1]\ y\ ys = [2]$ con lo cual no da que es exactamente igual. Esta expresión tiene sentido y es Verdadera.

3 Otras estructuras de datos

3.1 Ejercicio 9

Dadas las funciones **altura** y **cantNodos** definidas en la práctica 1 para árboles binarios, demostrar la siguiente propiedad:

$$\forall x :: AB\ a. \quad altura\ x \leq cantNodos\ x$$

3.2 Ejercicio 12

Dados el tipo Polinomio definido en la práctica 1 y las siguientes funciones:

```haskell
derivado :: Num a => Polinomio a -> Polinomio a
derivado poli = case poli of
    X           -> Cte 1
    Cte _       -> Cte 0
    Suma p q    -> Suma (derivado p) (derivado q)
    Prod p q    -> Suma (Prod (derivado p) q) (Prod (derivado q) p)

sinConstantesNegativas :: Num a => Polinomio a -> Polinomio a
sinConstantesNegativas = foldPoli True (>=0) (&&) (&&)

esRaiz :: Num a => a -> Polinomio a -> Bool
esRaiz n p = evaluar n p == 0
```

Nos piden demostrar :

1. $Num\ a => \forall p :: Polinomio\ a. \forall q :: Polinomio\ a. \forall r :: a. \quad (esRaiz\ r\ p \implies esRaiz\ r\ (Prod\ p_{,}; q))$
2. $Num\ a => \forall p :: Polinomio\ a. \forall k :: a. \forall e :: a. \quad evaluar\ e\ (derivado\ (Prod\ (Cte\ k)\ p)) = evaluar\ e\ (Prod\ (Cte\ k)\ (derivado\ p))$
3. $Num\ a => \forall p :: Polinomio\ a. \quad (sinConstantesNegativas\ p \implies sinConstantesNegativas\ (derivado\ p))$