Práctica 2: Razonamiento Ecuacional e Inducción Estructural

Tomás Felipe Melli

July 8, 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. Queremos ver que $\forall p :: (a, b)$. `intercambiar` (`intercambiar` p) $= p$. Por inducción sobre pares, alcanza ver que $\forall x :: a. \forall y :: b.$ `intercambiar` (`intercambiar` $(x, y)) = (x, y)$

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

2. Queremos ver que $\forall p :: (a,(b,c)).asociarD\ \ (asociarI\ \ p) = p$. Por inducción sobre pares, alcanza ver que $\forall x :: a, y :: (b, c).\ asociarD\ \ (asociarI\ \ (x, y)) = (x, y)$. Por inducción sobre pares, alcanza ver que $\forall x :: a, y :: b, z :: c.\ asociarD\ (asociarI\ (x,(y, z))) = (x,(y, z))$

$$asociarD\ (asociarI\ (x,(y, z))) = (x,(y, z))$$
$$\overset{\{AI\}}{=} asociarD\ ((x, y), z) = (x,(y, z))$$
$$\overset{\{AD\}}{=} (x,(y, z)) = (x,(y, z))$$

3. Queremos ver que $\forall p :: Either\ \ a\ \ b.\ espejar\ \ (espejar\ \ p) = p$. Por el *lema de generación para Sumas si* $p :: Either\ a\ b$ *entonces* :

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

Con esto en mente, separamos en casos :

- Caso $\exists x :: a\ /\ p = Left\ x$

$$espejar\ (espejar\ p) = p$$
$$\overset{Lema}{=} espejar\ (espejar\ (Left\ x)) = (Left\ x)$$
$$\overset{E1}{=} espejar\ (Right\ x) = (Left\ x)$$
$$\overset{E2}{=} (Left\ x) = (Left\ x)$$

- Caso $\exists y :: b\ /\ p = Rigth\ y$

$$espejar\ (espejar\ p) = p$$
$$\overset{Lema}{=} espejar\ (espejar\ (Right\ y)) = (Rigth\ y)$$
$$\overset{E2}{=} espejar\ (Left\ y) = (Rigth\ y)$$
$$\overset{E1}{=} (Rigth\ y) = (Rigth\ y)$$

4. Queremos ver que $\forall f :: a- > b- > c.\forall x :: a.\forall y :: b.\ flip\ (flip\ f)\ x\ y = f\ x\ y$.

$$flip\ (flip\ f)\ x\ y = f\ x\ y$$
$$\overset{F}{=} (flip\ f)\ y\ x = f\ x\ y$$
$$\overset{F}{=} flip\ f\ y\ x = f\ x\ y$$
$$\overset{F}{=} f\ x\ y = f\ x\ y$$

5. Queremos ver que $\forall f :: a- > b- > c.\forall x :: a.\forall y :: b.\ curry\ (uncurry\ f)\ x\ y = f\ x\ y$.

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

1.2 Ejercicio 2

1.

`Tenemos :`

$$g :: a' - > b'$$
$$f :: c' - > a'$$
$$g.f :: c' - > b'$$
$$flipInt :: (a- > b- > c)- > b- > a- > c$$
$$flipExt :: (d- > e- > f)- > e- > d- > f$$

$flip\ .\ flip = id$. Toma algo del tipo que toma `flipInt` que es el interno, o sea, $a- > b- > c$ y devuelve algo del tipo de los que devuelve el flip externo, es decir $e- > d- > f$. Como $b- > a- > c = d- > e- > f$ para que esté bien definida la composición, $e- > d- > f = a- > b- > c$. Queremos ver que $flip\ .\ flip = id :: (a- > b- > c)- > (a- > b- > c)$. Por extensionalidad funcional, basta ver que $\forall$ `f :: a -> b -> c. flip . flip f = id f :: a -> b -> c` o sea, `(flip . flip) f = flip (flip f)` y queremos ver que eso es igual a `id f`. Que llamaremos $\{E1\}$
Por extensionalidad funcional, basta ver que $\forall x :: a$, `(flip . flip) f x = id f x :: b -> c`. Que llamaremos $\{E2\}$
Por extensionalidad funcional, basta ver que $\forall y :: b$, `(flip . flip) f x y = id f x y :: c`. Que llamaremos $\{E3\}$

$$((flip\ .\ flip)fxy$$
$$= ((flip\ .\ flip)f)\ x\ y$$
$$\stackrel{E1}{=} (flip\ (flip\ f))\ x\ y$$
$$= flip\ (flip\ f)\ x\ y$$
$$\stackrel{E2}{=} flip\ f\ y\ x$$
$$\stackrel{E2}{=} f\ x\ y$$
$$\stackrel{E3}{=} (id\ f)\ x\ y$$
$$= id\ f\ x\ y$$

2. Queremos ver que $\forall f :: (a, b)- > c.\ uncurry\ (curry\ f) = f :: (a, b)- > c$. Por extensionalidad funcional, basta ver que $\forall p :: (a, b)$, `uncurry (curry f) p = f p`. Llamaremos $\{E1\}$
Por inducción sobre pares, basta ver que $\forall x :: a, y :: b$, `uncurry (curry f) (x,y) = f (x,y)`

$$uncurry\ (curry\ f)\ (x, y)$$
$$\stackrel{E2}{=} (curry\ f)\ x\ y$$
$$= curry\ f\ x\ y$$
$$\stackrel{E1}{=} f\ (x, y)$$

3.

```
flip :: (a -> b -> c) -> b -> a -> c
const :: d -> e -> d
id :: f -> f
flip const :: e -> d -> d
```

Queremos ver que `flip const = const id :: e -> d -> d`
Por extensionalidad funcional, basta ver que $\forall x :: e$ `flip const x = const id x :: d -> d` que llamaremos $\{E1\}$.

Por extensionalidad funcional, basta ver que $\forall x :: e, y :: d$ `flip const x y = const id x y ::  d` que llamaremos $\{E2\}$.

$$flip\ const\ x\ y$$
$$\overset{E1}{=} const\ y\ x$$
$$\overset{E2}{=} y$$
$$\overset{E3}{=} id\ y$$
$$\overset{E2}{=} (const\ id\ x)\ y$$
$$= const\ id\ x\ y$$

La última línea vale porque asocia a izquierda.

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 $\forall x :: a$:

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

2 Inducción sobre Listas

2.1 Ejercicio 3

Se tienen las siguientes funciones:

```
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. Queremos ver que $\forall ys :: [a].$ se cumple que $length\ (duplicar\ ys) = 2 \times length\ ys$
   Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([ ])` y $\forall x :: a, xs :: [a]$ `P(xs)` $\Rightarrow$ `P(x:xs)`

   - Caso base : **ys = []**

     $$\texttt{length (duplicar [])}$$
     $$\overset{D0}{=} \texttt{length []}$$
     $$\overset{L0}{=} \texttt{0}$$
     $$\overset{Int}{=} \texttt{2 * 0}$$
     $$\overset{L0}{=} \texttt{2 * length []}$$

Entonces vale `P([])`.

- Caso inductivo : `ys = (x:xs)`. Suponemos que vale `P(xs)`

  `length (duplicar (x:xs))`

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

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

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

  $\overset{HI}{=}$ `1 + 1 + 2 * length xs`

  $=$ `2 + 2 * length xs`

  $=$ `2 * (1 + length xs)`

  $\overset{L1}{=}$ `2 * length (x:xs)`

2. Queremos ver que $\forall zs :: [a]$. se cumple que `P(zs) =` $\forall$ `ys :: [a] , length (zs ++ ys) = length zs + length ys`.
   La idea es hacer inducción sobre una de las listas, en general vamos a elegir la más conveniente.
   Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])` y $\forall$`x ::a, xs ::[a], P(xs)` $\Rightarrow$ `P(x:xs)`

   - Caso base : `xs = []`

     `P([]): length ([] ++ ys)`

     $\overset{++0}{=}$ `length ys`

     $\overset{Int}{=}$ `0 + length ys`

     $\overset{L0}{=}$ `length [] + length ys`

   - Paso inductivo : `zs = (x:xs)`. Suponemos que vale `P(xs)`

     `length((x:xs) ++ ys)`

     $\overset{++1}{=}$ `length(x : (xs ++ ys))`

     $\overset{L1}{=}$ `1 + length(xs ++ ys)`

     $\overset{HI}{=}$ `1 + length xs + length ys)`

     $\overset{L1}{=}$ `length (x:xs) + length ys)`

3. Queremos ver que $\forall$ `xs :: [a] vale P(xs) =` $\forall$ `x :: a. append [x] xs = x:xs`
   Por lema de generación de listas,

   (a) `o xs = []`

   (b) `o xs = y:ys con y :: a, ys :: [a]`

   Vamos a separar en casos :

   (a) Caso `xs = []`

     `append [x] []`

     $\overset{A0}{=}$ `foldr (:) [] [x]`

     $\overset{foldr1}{=}$ `(:) x (foldr (:) [] [])`

     $\overset{foldr0}{=}$ `(:) x []`

     $=$ `x : []`

(b) Caso `xs = y:ys`

$\texttt{append [x] (y:ys)}$

$\overset{A0}{=}$ `foldr (:)  (y:ys) [x]`

$\overset{foldr1}{=}$ `(:)  x (foldr (:)  (y:ys) [])`

$\overset{foldr0}{=}$ `(:)  x (y:ys)`

$=$ `x :  (y:ys)`

4. Queremos ver que $\forall$ `ys ::  [a]` vale `P(ys) =` $\forall$ `f ::  (a -> b), length (map f ys) = length ys`.
Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])` y $\forall$ `x ::  a, xs ::  [a],` `P(xs)` $\Rightarrow$ `P(x:xs)`

- Caso `ys = []`

`length (map f [])`

$\overset{map0}{=}$ `length []`

- Caso `ys = (x:xs)`. Suponemos que vale `P(xs)` y queremos ver que vale `P(x:xs)`

`length (map f (x:xs))`

$\overset{map1}{=}$ `length (f x :  (map f xs))`

$\overset{L1}{=}$ `1 + length (map f xs)`

$\overset{HI}{=}$ `1 + length xs`

$\overset{L1}{=}$ `length (x:xs)`

5. Queremos ver que $\forall$ `ys ::  [a]` vale `P(ys) =` $\forall$ `p ::  (a -> Bool),` $\forall$ `e ::  a, elem e (filter p ys)` $\Rightarrow$ `elem e ys` (asumiendo `Eq a`).
Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])` y $\forall$ `x ::  a, xs ::  [a]`, `P(xs)` $\Rightarrow$ `P(x:xs)`.

- Caso `ys = []`

`elem e (filter p [])`

$\overset{\text{filter0}}{=}$ `elem e []` $\Rightarrow$ `elem e []`

$\overset{\text{elem0}}{=}$ `False` $\Rightarrow$ `False`

$\overset{\text{Bool}}{=}$ `True`

- Caso `ys = (x:xs)`. Supongo que vale `P(xs)`, o sea, `elem e (filter p xs)` $\Rightarrow$ `elem e xs`

`elem e (filter p (x:xs))` $\Rightarrow$ `elem e (x:xs)`

$\overset{\text{filter1}}{=}$ `elem e (if p x then x :  filter p xs else filter p xs)` $\Rightarrow$ `elem e (x:xs)`

$= *$

Por lema de generación de Bool, `p x` es `True` o `False`.

- Caso `p x = True`

$* =$ `elem e (x :  filter p xs)` $\Rightarrow$ `elem e (x:xs)`

$\overset{\text{elem1}}{=}$ `e == x || elem e (filter p xs)` $\Rightarrow$ `elem e (x:xs)`

$= **$

Por lema de generación de Bool, `e == x` es `True` o `False`.

* Caso `e == x = True`

$$
\begin{aligned}
** &= \texttt{True || elem e (filter p xs)} \Rightarrow \texttt{elem e (x:xs)} \\
&\overset{||}{=} \texttt{True} \Rightarrow \texttt{elem e (x:xs)} \\
&\overset{\text{Bool}}{=} \texttt{elem e (x:xs)} \\
&\overset{\text{elem1}}{=} \texttt{e == x || elem e xs} \\
&\overset{\text{e == x}}{=} \texttt{True || elem e xs} \\
&\overset{||}{=} \texttt{True}
\end{aligned}
$$

* Caso `e == x = False`

$$
\begin{aligned}
** &= \texttt{False || elem e (filter p xs)} \Rightarrow \texttt{elem e (x:xs)} \\
&\overset{||}{=} \texttt{elem e (filter p xs)} \Rightarrow \texttt{elem e (x:xs)} \\
&= ***
\end{aligned}
$$

  Por lema de generación de Bool, `elem e (filter p xs)` es `True` o `False`.

  · Caso `elem e (filter p xs) = True`

$$
\begin{aligned}
*** &= \texttt{True} \Rightarrow \texttt{elem e (x:xs)} \\
&\overset{\text{Bool}}{=} \texttt{elem e (x:xs)} \\
&\overset{\text{elem1}}{=} \texttt{e == x || elem e xs} \\
&\overset{HI}{=} \texttt{e == x || True} \\
&\overset{||}{=} \texttt{True}
\end{aligned}
$$

  · Caso `elem e (filter p xs) = False`

$$
\begin{aligned}
*** &= \texttt{False} \Rightarrow \texttt{elem e (x:xs)} \\
&\overset{\text{Bool}}{=} \texttt{True}
\end{aligned}
$$

– Caso `p x = False`

$$
\begin{aligned}
* &= \texttt{elem e (filter p xs)} \Rightarrow \texttt{elem e (x:xs)} \\
&= **
\end{aligned}
$$

Por lema de generación de Bool, `elem e (filter p xs)` es `True` o `False`.

* Caso `elem e (filter p xs) = True`

$$
\begin{aligned}
** &= \texttt{True} \Rightarrow \texttt{elem e (x:xs)} \\
&\overset{\text{Bool}}{=} \texttt{elem e (x:xs)} \\
&\overset{\text{elem1}}{=} \texttt{e == x || elem e xs} \\
&= \texttt{e == x || True} \quad (\text{por } \#) \\
&\overset{||}{=} \texttt{True}
\end{aligned}
$$

* Caso `elem e (filter p xs) = False`

$$
\begin{aligned}
** &= \texttt{False} \Rightarrow \texttt{elem e (x:xs)} \\
&\overset{\text{Bool}}{=} \texttt{True}
\end{aligned}
$$

Por HI: `elem e (filter p xs)` $\Rightarrow$ `elem e xs = True`, y por otro lado

$$
\begin{aligned}
&\texttt{elem e (filter p xs)} \Rightarrow \texttt{elem e xs} \\
&= \texttt{True} \Rightarrow \texttt{elem e xs} \\
&\overset{\text{Bool}}{=} \texttt{elem e xs} \\
&= \texttt{True}
\end{aligned}
$$

6. Queremos ver que $\forall$ `xs :: [a]` vale `P(xs) =` $\forall$ `x :: a, ponerAlFinal x xs = xs ++ (x:[])`.
Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])` y que $\forall$ `y :: a, ys :: [a]`, `P(ys)` $\Rightarrow$ `P(y:ys)`.

- Caso base: `xs = []`

  `ponerAlFinal x []`

  $\overset{\text{P0}}{=}$ `foldr (:) (x:[]) []`

  $\overset{\text{foldr0}}{=}$ `x:[]`

  $\overset{\text{++0}}{=}$ `[] ++ (x:[])`

- Paso inductivo: `xs = (y:ys)`. Supongo que vale `P(ys)`, queremos ver que vale `P(y:ys)`.

  `ponerAlFinal x (y:ys)`

  $\overset{\text{P0}}{=}$ `foldr (:) (x:[]) (y:ys)`

  $\overset{\text{foldr1}}{=}$ `(:) y (foldr (:) (x:[]) ys)`

  $\overset{\text{P0}}{=}$ `(:) y (ponerAlFinal x ys)`

  $\overset{\text{HI}}{=}$ `(:) y : (ys ++ (x:[]))`

  $=$ `(y : (ys ++ (x:[]))`

  $\overset{\text{++1}}{=}$ `(y:ys) ++ (x:[])`

7. Queremos ver que `reverse = foldr (\x rec -> rec ++ (x:[])) [] :: [a]` $\rightarrow$ `[a]`.
Por extensionalidad funcional, esto es equivalente a probar que para toda `ys :: [a]`, vale `P(ys) = reverse ys = foldr (\x rec -> rec ++ (x:[])) [] ys`.
Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])` y $\forall$ `x :: a, xs :: [a]`, `P(xs)`$\Rightarrow$ `P(x:xs)`

- Caso base: `ys = []`

  `reverse []`

  $\overset{\text{R0}}{=}$ `foldl (flip (:)) [] []`

  $\overset{\text{foldl0}}{=}$ `[]`

  $\overset{\text{foldr0}}{=}$ `foldr (\x rec -> rec ++ (x:[])) [] []`

- Paso inductivo: `ys = (x:xs)`. Supongo `P(xs)`, queremos probar `P(x:xs)`.

  `reverse (x:xs)`

  $\overset{\text{R0}}{=}$ `foldl (flip (:)) [] (x:xs)`

  $\overset{\text{foldl1}}{=}$ `foldl (flip (:)) ((flip (:)) [] x) xs`

  $\overset{\text{flip}}{=}$ `foldl (flip (:)) (x:[]) xs`

  $= **$ (ver más abajo)

  Queremos ver que `** = foldr (\y rec -> rec ++ (y:[])) [] (x:xs)`.
  Para esto, usamos el siguiente lema:

  Lema: `foldl (flip (:)) ac xs = foldl (flip (:)) [] xs ++ ac`

  $*$ Caso base: `xs = []`

  `foldl (flip (:)) ac []`

  $\overset{\text{foldl0}}{=}$ `ac`

  $\overset{\text{++0}}{=}$ `[] ++ ac`

  $\overset{\text{foldl0}}{=}$ `foldl (flip (:)) [] [] ++ ac`

- * Paso inductivo: `xs = (y:ys)`. HI: `foldl (flip (:)) ac ys = foldl (flip (:)) [] ys ++ ac`

  ```
  foldl (flip (:))  ac (y:ys)
  ```
  $\overset{\text{foldl1}}{=}$ `foldl (flip (:))  (y:ac) ys`

  $\overset{\text{HI}}{=}$ `foldl (flip (:))  [] ys ++ (y:ac)`

  $\overset{++0}{=}$ `foldl (flip (:))  [] ys ++ (y:([] ++ ac))`

  $\overset{++1}{=}$ `foldl (flip (:))  [] ys ++ (y:[] ++ ac)`

  $=$ `foldl (flip (:))  [] ys ++ (y:[]) ++ ac`

  $\overset{\text{HI}}{=}$ `foldl (flip (:))  (y:[]) ys ++ ac`

  $\overset{\text{flip}}{=}$ `foldl (flip (:))  ((flip (:))  [] y) ys ++ ac`

  $\overset{\text{foldl1}}{=}$ `foldl (flip (:))  [] (y:ys) ++ ac`

Por el lema, tenemos:

`**` $=$ `foldl (flip (:))  (x:[]) xs`

$=$ `foldl (flip (:))  [] xs ++ (x:[])`

$\overset{\text{R0}}{=}$ `reverse xs ++ (x:[])`

$\overset{\text{HI}}{=}$ `(foldr (\y rec -> rec ++ (y:[])) [] xs) ++ (x:[])`

$\overset{\text{foldr1}}{=}$ `foldr (\y rec -> rec ++ (y:[])) [] (x:xs)`

8. Queremos ver que ∀ `ys :: [a]`, vale `P(ys) =` ∀ `x :: a, head (reverse (ponerAlFinal x ys)) = x`. Por inducción sobre listas:

   - Caso base: `ys = []`

     ```
     head (reverse (ponerAlFinal x []))
     ```
     $\overset{\text{P0}}{=}$ `head (reverse (x:[]))`

     $\overset{\text{R'1}}{=}$ `head (reverse [] ++ [x])`

     $\overset{\text{R'0}}{=}$ `head ([] ++ [x])`

     $\overset{++1}{=}$ `head [x]`

     $\overset{(:)}{=}$ `head (x:[])`

     $\overset{\text{head}}{=}$ `x`

   - Paso inductivo: `ys = z:zs`. Supongo que vale `P(zs)`, queremos probar `P(z:zs)`.

     ```
     head (reverse (ponerAlFinal x (z:zs)))
     ```
     $\overset{\text{P0}}{=}$ `head (reverse (foldr (:)  (x:[]) (z:zs)))`

     $\overset{\text{foldr1}}{=}$ `head (reverse (z :  (foldr (:)  (x:[]) zs)))`

     $\overset{\text{reverse1}}{=}$ `head (reverse (foldr (:)  (x:[]) zs) ++ [z])`

     $\overset{\text{Lema1}}{=}$ `head (reverse (zs ++ [x]) ++ [z])`

     $\overset{\text{Lema2}}{=}$ `head ((x :  reverse zs) ++ [z])`

     $\overset{++1}{=}$ `head (x :  (reverse zs ++ [z]))`

     $\overset{\text{head}}{=}$ `x`

Lema1: Para todo `zs :: [a]`, vale: `foldr (:)  (x:[]) zs = zs ++ [x]`.

- Caso base: `zs = []`

  `foldr (:) (x:[]) []`

  $\stackrel{\text{foldr0}}{=}$ `x:[]`

  $\stackrel{(:)}{=}$ `[x]`

  $\stackrel{\text{++0}}{=}$ `[] ++ [x]`

- Paso inductivo: `zs = y:ys`. Supongo HI: `foldr (:) (x:[]) ys = ys ++ [x]`

  `foldr (:) (x:[]) (y:ys)`

  $\stackrel{\text{foldr1}}{=}$ `y : (foldr (:) (x:[]) ys)`

  $\stackrel{\text{HI}}{=}$ `y : (ys ++ [x])`

  $\stackrel{\text{++1}}{=}$ `(y:ys) ++ [x]`

Lema2: `reverse (zs ++ [x]) = x : reverse zs`.
Este lema fue demostrado previamente en el ítem anterior.

Observación: Se intentó inicialmente usar un lema directo con `head (ys ++ [z])`, pero esto requiere casos sobre `null ys`. En cambio, este enfoque evita esa complejidad.

(Extra) Lema general: ∀ `ys :: [a]`, ∀ `z :: a`,
`head (ys ++ [z]) = if null ys then z else head ys`

2.2 Ejercicio 4

1. Queremos ver que `reverse . reverse = id :: [a] → [a]`.
   Por extensionalidad funcional, esto es equivalente a probar que para toda `ys :: [a]`, vale `P(ys) = (reverse . reverse) ys = id ys`.
   Por inducción sobre listas, esto es equivalente a ver si se cumple `P([])` y ∀`x :: a, xs :: [a], P(xs) ⇒ P(x:xs)`.

   - Caso base: `ys = []`

     `(reverse . reverse) []`

     $\stackrel{(.)}{=}$ `reverse (reverse [])`

     $\stackrel{\text{reverse0}}{=}$ `reverse []`

     $\stackrel{\text{reverse0}}{=}$ `[]`

     $\stackrel{\text{id0}}{=}$ `id []`

   - Paso inductivo: `ys = x:xs`. Supongo que vale `P(xs)`, queremos ver que vale `P(x:xs)`.

     `(reverse . reverse) (x:xs)`

     $\stackrel{(.)}{=}$ `reverse (reverse (x:xs))`

     $\stackrel{\text{reverse1}}{=}$ `reverse (reverse xs ++ [x])`

     $\stackrel{\text{Lema}}{=}$ `x : (reverse (reverse xs))`

     $\stackrel{(.)}{=}$ `x : ((reverse . reverse) xs)`

     $\stackrel{\text{HI}}{=}$ `x : xs`

     $\stackrel{\text{id}}{=}$ `id (x:xs)`

   Lema auxiliar: Para todo `ys :: [a]`, y `n :: a`, vale:
   `reverse (ys ++ [n]) = n : reverse ys`

- Caso base: `ys = []`

  $\texttt{reverse ([] ++ [n])}$

  $\overset{++0}{=} \texttt{reverse [n]}$

  $\overset{\text{reverse1}}{=} \texttt{reverse [] ++ [n]}$

  $\overset{\text{reverse0}}{=} \texttt{[] ++ [n]}$

  $\overset{++0}{=} \texttt{[n]}$

  $\overset{(:)}{=} \texttt{n : []}$

  $\overset{\text{reverse0}}{=} \texttt{n : reverse []}$

- Paso inductivo: `ys = x:xs`. Supongo que vale `reverse (xs ++ [n]) = n : reverse xs`.

  $\texttt{reverse ((x:xs) ++ [n])}$

  $\overset{++1}{=} \texttt{reverse (x : (xs ++ [n]))}$

  $\overset{\text{reverse1}}{=} \texttt{reverse (xs ++ [n]) ++ [x]}$

  $\overset{\text{HI}}{=} \texttt{(n : reverse xs) ++ [x]}$

  $\overset{++1}{=} \texttt{n : (reverse xs ++ [x])}$

  $\overset{\text{reverse1}}{=} \texttt{n : reverse (x:xs)}$

2. Queremos ver que `append = (++) :: [a]` $\to$ `[a]` $\to$ `[a]`.
   Por extensionalidad funcional, esto es equivalente a probar que para toda `xs :: [a]`, `ys :: [a]`, vale `P(xs) = append xs ys = xs ++ ys`.
   Aplicamos inducción sobre listas.

   - Caso base: `xs = []`

     $\texttt{append [] ys}$

     $\overset{\text{A0}}{=} \texttt{foldr (:) ys []}$

     $\overset{\text{foldr0}}{=} \texttt{ys}$

     $\overset{++0}{=} \texttt{[] ++ ys}$

   - Paso inductivo: `xs = z:zs`. Supongo que vale `P(zs)`, queremos ver que vale `P(z:zs)`.

     $\texttt{append (z:zs) ys}$

     $\overset{\text{A0}}{=} \texttt{foldr (:) ys (z:zs)}$

     $\overset{\text{foldr1}}{=} \texttt{z : (foldr (:) ys zs)}$

     $\overset{\text{A0}}{=} \texttt{z : (append zs ys)}$

     $\overset{\text{HI}}{=} \texttt{z : (zs ++ ys)}$

     $\overset{++1}{=} \texttt{(z:zs) ++ ys}$

3. Queremos ver que `map id = id :: [a]` $\to$ `[a]`.
   Esto es equivalente a probar que para toda `xs :: [a]`, vale `P(xs) = map id xs = id xs`.
   Aplicamos inducción sobre listas.

   - Caso base: `xs = []`

     $\texttt{map id []}$

     $\overset{\text{map0}}{=} \texttt{[]}$

     $\overset{\text{id}}{=} \texttt{id []}$

- Paso inductivo: `xs = y:ys`. Supongo que vale `P(ys)`, queremos ver que vale `P(y:ys)`.

```
map id (y:ys)
```
$\stackrel{\text{map1}}{=}$ `id y : map id ys`

$\stackrel{\text{HI}}{=}$ `id y : id ys`

$\stackrel{\text{id}}{=}$ `y : ys`

$\stackrel{\text{id}}{=}$ `id (y:ys)`

4. Queremos ver que ∀ `f :: a` → `b`, ∀ `g :: b` → `c`, `map (g . f) = map g . map f`.
   Esto es equivalente a probar que para toda `ys :: [a]`, vale `P(ys) = map (g . f) ys = (map g . map f) ys`.
   Aplicamos inducción sobre listas.

   - Caso base: `ys = []`

   `map (g . f) []`

   $\stackrel{\text{map0}}{=}$ `[]`

   $\stackrel{\text{map0}}{=}$ `map g []`

   $\stackrel{\text{map0}}{=}$ `map g (map f [])`

   $\stackrel{(.)}{=}$ `(map g . map f) []`

   - Paso inductivo: `ys = x:xs`. Supongo que vale `P(xs)`, queremos ver que vale `P(x:xs)`.

   `map (g . f) (x:xs)`

   $\stackrel{\text{map1}}{=}$ `(g . f) x : map (g . f) xs`

   $\stackrel{(.)}{=}$ `g (f x) : map (g . f) xs`

   $\stackrel{\text{HI}}{=}$ `g (f x) : (map g . map f) xs`

   $\stackrel{(.)}{=}$ `g (f x) : map g (map f xs)`

   $\stackrel{map1}{=}$ `map g (f x : map f xs)`

   $\stackrel{map1}{=}$ `map g (map f (x:xs))`

   $\stackrel{(.)}{=}$ `(map g . map f) (x:xs)`

5. Queremos ver que ∀ `f :: a` → `b`, ∀ `p :: b` → `Bool`, vale:
   `map f . filter (p . f) = filter p . map f :: [a]` → `[b]`.
   Por extensionalidad, esto es equivalente a probar que para toda `ys :: [a]`, vale `P(ys) = (map f . filter (p . f)) ys = (filter p . map f) ys`.
   Aplicamos inducción sobre listas.

   - Caso base: `ys = []`

   `(map f . filter (p . f)) []`

   $\stackrel{(.)}{=}$ `map f (filter (p . f) [])`

   $\stackrel{\text{filter0}}{=}$ `map f []`

   $\stackrel{\text{map0}}{=}$ `[]`

   $\stackrel{\text{filter0}}{=}$ `filter p []`

   $\stackrel{\text{map0}}{=}$ `filter p (map f [])`

   $\stackrel{(.)}{=}$ `(filter p . map f) []`

- Paso inductivo: `ys = x:xs`. Supongo que vale `P(xs)`, queremos ver que vale `P(x:xs)`.

  $$\begin{aligned}
  &\texttt{(map f . filter (p . f)) (x:xs)}\\
  &\stackrel{(.)}{=} \texttt{map f (filter (p . f) (x:xs))}\\
  &\stackrel{\text{filter1}}{=} \texttt{map f (if (p . f) x then x : filter (p . f) xs else filter (p . f) xs)}\\
  &= *
  \end{aligned}$$

  Por lema de generación de Bool, `(p . f) x` es `True` o `False`.

  - Caso `(p . f) x = True`

    $$\begin{aligned}
    * &= \texttt{map f (x : filter (p . f) xs)}\\
    &\stackrel{\text{map1}}{=} \texttt{f x : map f (filter (p . f) xs)}\\
    &\stackrel{(.)}{=} \texttt{f x : (map f . filter (p . f)) xs}\\
    &\stackrel{\text{HI}}{=} \texttt{f x : (filter p . map f) xs}\\
    &\stackrel{(.)}{=} \texttt{f x : filter p (map f xs)}\\
    &\stackrel{\text{filter1}}{=} \texttt{filter p (f x : map f xs)}\\
    &\stackrel{\text{map1}}{=} \texttt{filter p (map f (x:xs))}\\
    &\stackrel{(.)}{=} \texttt{(filter p . map f) (x:xs)}
    \end{aligned}$$

  - Caso `(p . f) x = False`

    $$\begin{aligned}
    * &= \texttt{map f (filter (p . f) xs)}\\
    &\stackrel{(.)}{=} \texttt{(map f . filter (p . f)) xs}\\
    &\stackrel{\text{HI}}{=} \texttt{(filter p . map f) xs}\\
    &= \texttt{filter p (map f xs)}\\
    &\stackrel{\text{filter1}}{=} \texttt{filter p (f x : map f xs)}\\
    &\stackrel{\text{map1}}{=} \texttt{filter p (map f (x:xs))}\\
    &\stackrel{(.)}{=} \texttt{(filter p . map f) (x:xs)}
    \end{aligned}$$

6. Queremos ver que ∀ `xs :: [a]`, vale `P(xs) =` ∀ `f :: a` → `b`, ∀ `e :: a, elem e xs` ⇒ `elem (f e) (map f xs)`.
   (Asumiendo `Eq a` y `Eq b`). Aplicamos inducción sobre listas.

   - Caso base: `xs = []`

     $$\begin{aligned}
     &\texttt{elem e []} \Rightarrow \texttt{elem (f e) (map f [])}\\
     &\stackrel{\text{elem0}}{=} \texttt{False} \Rightarrow \texttt{elem (f e) []}\\
     &\stackrel{\text{map0}}{=} \texttt{False} \Rightarrow \texttt{False}\\
     &\stackrel{\text{Bool}}{=} \texttt{True}
     \end{aligned}$$

   - Paso inductivo: `xs = x:xs`. Supongo que vale `P(xs)`. Queremos probar:

     $$\begin{aligned}
     &\texttt{elem e (x:xs)} \Rightarrow \texttt{elem (f e) (map f (x:xs))}\\
     &\stackrel{\text{elem1}}{=} \texttt{e == x || elem e xs} \Rightarrow \texttt{elem (f e) (map f (x:xs))}\\
     &= *
     \end{aligned}$$

     Por lema de generación de Bool, `e == x` es `True` o `False`.

- Caso `e == x = True`

  $* =$ `True` $\Rightarrow$ `elem (f e) (map f (x:xs))`

  $\overset{\text{Bool}}{=}$ `elem (f e) (map f (x:xs))`

  $\overset{\text{map1}}{=}$ `elem (f e) (f x : map f xs)`

  $\overset{\text{elem1}}{=}$ `f e == f x || elem (f e) (map f xs)`

  $= **$

  Por LG de Bool, `f e == f x` es `True` o `False`.

  - Caso `f e == f x = True`

    $** =$ `True || elem (f e) (map f xs)`

    $\overset{||}{=}$ `True`

  - Caso `f e == f x = False` (imposible si `e == x`)

- Caso `e == x = False`

  $* =$ `False || elem e xs` $\Rightarrow$ `elem (f e) (map f (x:xs))`

  $=$ `elem e xs` $\Rightarrow$ `elem (f e) (map f (x:xs))`

  $\overset{\text{map1}}{=}$ `elem e xs` $\Rightarrow$ `elem (f e) (f x : map f xs)`

  $\overset{\text{elem1}}{=}$ `elem e xs` $\Rightarrow$ `(f e == f x || elem (f e) (map f xs))`

  $= **$

  Por LG de Bool, `f e == f x` es `True` o `False`.

  - Caso `f e == f x = True`

    $** =$ `elem e xs` $\Rightarrow$ `True`

    $\overset{\text{Bool}}{=}$ `True`

  - Caso `f e == f x = False`

    $** =$ `elem e xs` $\Rightarrow$ `elem (f e) (map f xs)`

    $\overset{\text{HI}}{=}$ `True`

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 ver que `zip = zip' :: [a]` $\to$ `[b]` $\to$ `[(a,b)]`.
Por principio de extensionalidad, esto es equivalente a ver que $\forall$ `xs :: [a]`, vale `zip xs = zip' xs :: [b]` $\to$ `[(a,b)]`.
Aplicamos inducción sobre listas.

- Caso base: `xs = []`

  `zip []`

  $\overset{\text{Z0}}{=}$ `foldr (\x rec ys` $\to$ `if null ys then [] else (x, head ys) : rec (tail ys)) (const []) []`

  $\overset{\text{foldr0}}{=}$ `const []`

  $\overset{\text{const}}{=}$ `\ys` $\to$ `[]`

  $\overset{\text{Z'0}}{=}$ `zip' []`

- Paso inductivo: `xs = z:zs`. Supongamos como hipótesis inductiva que `zip zs = zip' zs`. Queremos ver que `zip (z:zs) = zip' (z:zs)`.

  `zip (z:zs)`

  $\overset{\text{Z0}}{=}$ `foldr (\x rec ys' → if null ys' then [] else (x, head ys') : rec (tail ys')) (const []) (z:zs)`

  $\overset{\text{foldr1}}{=}$ `(\x rec ys' → if null ys' then [] else (x, head ys') : rec (tail ys')) z (foldr ...zs)`

  $\overset{\beta}{=}$ `(\rec ys' → if null ys' then [] else (z, head ys') : rec (tail ys')) (zip zs)`

  $\overset{\text{HI}}{=}$ `(\rec ys' → if null ys' then [] else (z, head ys') : rec (tail ys')) (zip' zs)`

  $\overset{\beta}{=}$ `\ys' → if null ys' then [] else (z, head ys') : zip' zs (tail ys')`

  $\overset{\text{Z'1}}{=}$ `zip' (z:zs)`

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
```

1. Verdadero. Vamos a suponer que vale `Eq a`. Esto lo suponemos ya que si es Falso, vale la implicación por Bool.
   Queremos ver que ∀ `xs :: [a]`, vale `P(xs)`:
   ∀ `e :: a`, ∀ `p :: a -> Bool`, `elem e xs && p e = elem e (filter p xs)`.
   Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])`
   y que ∀ `x :: a`, `xs :: [a]`, si `P(xs)` ⇒ `P(x:xs)`.

   - Caso base: `xs = []`

     `elem e [] && p e`

     $\overset{\text{elem0}}{=}$ `False && p e`

     $\overset{\&\&}{=}$ `False`

     $\overset{\text{elem0}}{=}$ `elem e []`

     $\overset{\text{filter0}}{=}$ `elem e (filter p [])`

   - Paso inductivo: `xs = y:ys`. Supongamos como HI que:

     `elem e ys && p e = elem e (filter p ys)`

     Queremos ver que:

     `elem e (y:ys) && p e = elem e (filter p (y:ys))`

     `elem e (y:ys) && p e`

     $\overset{\text{elem1}}{=}$ `(e == y || elem e ys) && p e`

     Por Lema de generación de Bool, separamos en casos

     - Caso `e == y = True`

       `(True || elem e ys) && p e`

       $\overset{||}{=}$ `True && p e`

       $\overset{\&\&}{=}$ `p e`

       Por lema de generación de Bool subdividimos en más casos :

- * Subcaso `p e = True`

  `p e` $=$ `True`

  $\overset{||}{=}$ `True || elem e (filter p ys)`

  $\overset{\text{e == y}}{=}$ `e == y || elem e (filter p ys)`

  $\overset{\text{elem1}}{=}$ `elem e (y : filter p ys)`

  $\overset{\text{p e = True}}{=}$ `elem e (if p y then y : filter p ys else filter p ys)`

  $\overset{\text{filter1}}{=}$ `elem e (filter p (y:ys))`

- * Subcaso `p e = False`

  `p e` $=$ `False`

  $\equiv$ `elem e ys && p e`

  $\overset{\text{HI}}{=}$ `elem e (filter p ys)`

  $\overset{\text{p e = False}}{=}$ `elem e (if p y then y : filter p ys else filter p ys)`

  $\overset{\text{filter1}}{=}$ `elem e (filter p (y:ys))`

– Caso `e == y = False`

  `(False || elem e ys) && p e`

  $\overset{||}{=}$ `elem e ys && p e`

  $\overset{\text{HI}}{=}$ `elem e (filter p ys)`

  Por lema de generación de Bool subdividimos en casos

- * Subcaso `p y = False`

  `elem e (filter p ys)`

  $\overset{\text{p y = False}}{=}$ `elem e (if p y then y : filter p ys else filter p ys)`

  $\overset{\text{filter1}}{=}$ `elem e (filter p (y:ys))`

- * Subcaso `p y = True`

  `elem e (filter p ys)`

  $\overset{||}{=}$ `False || elem e (filter p ys)`

  $\overset{\text{e == y = False}}{=}$ `e == y || elem e (filter p ys)`

  $\overset{\text{elem1}}{=}$ `elem e (y : filter p ys)`

  $\overset{\text{filter1}}{=}$ `elem e (filter p (y:ys))`

2. Queremos ver que `Eq a =>` $\forall$ `xs :: [a],` $\forall$ `e :: a, elem e xs = elem e (nub xs)`.
   Si no vale `Eq a`, ya vale la implicación por tipo `Bool`, ya que `False` $\rightarrow$ `algo = True`.
   Por lo tanto, suponemos en adelante que vale `Eq a`.
   Queremos ver que para todo `xs :: [a]`, vale `P(xs)`:
   $\forall$ `e :: a`, `elem e xs = elem e (nub xs)`.
   Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])` y para todo `x :: a`, `xs :: [a]`, `P(xs)` $\Rightarrow$ `P(x:xs)`.

   - Caso base: `xs = []`

     `elem e []`

     $\overset{\text{N0}}{=}$ `elem e (nub [])`

- Paso inductivo: `xs = y:ys`. Supongamos como hipótesis inductiva (HI) que $\forall e :: a$ `elem e ys = elem e (nub ys)`. Queremos ver que `elem e (y:ys) = elem e (nub (y:ys))`.

  `elem e (y:ys)`

  $\overset{\text{def}}{=}$ `e == y || elem e ys`

  $\overset{\text{HI}}{=}$ `e == y || elem e (nub ys)`

  $=$ `*`

  Por lema de generación de bool separamos en casos :

  - Caso `e == y = True`:

    `(*) = True || elem e (nub ys)`

    $\overset{||}{=}$ `True`

    $\overset{\text{bool}}{=}$ `True || elem e (filter (\h` $\rightarrow$ `y /= h) (nub ys))`

    $\overset{\text{bool}}{=}$ `e == y || elem e (filter (\h` $\rightarrow$ `y /= h) (nub ys))`

    $\overset{\text{elem1}}{=}$ `elem e (y : filter (\h` $\rightarrow$ `y /= h) (nub ys))`

    $\overset{\text{N1}}{=}$ `elem e (nub (y:ys))`

  - Caso `e == y = False`:

    `(*) = False || elem e (nub ys)`

    $\overset{||}{=}$ `elem e (nub ys)`

    $=$ `elem e (nub ys) && True`

    $\overset{\text{bool}}{=}$ `elem e (nub ys) && e /= y`

    $\overset{\beta}{=}$ `elem e (nub ys) && (\h` $\rightarrow$ `y /= h) e`

    $\overset{\text{item a)}}{=}$ `elem e (filter (\h` $\rightarrow$ `y /= h) (nub ys))`

    $\overset{||}{=}$ `False || elem e (filter (\h` $\rightarrow$ `y /= h) (nub ys))`

    $\overset{\text{bool}}{=}$ `e == y || elem e (filter (\h` $\rightarrow$ `y /= h) (nub ys))`

    $\overset{\text{elem1}}{=}$ `elem e (y : filter (\h` $\rightarrow$ `y /= h) (nub ys))`

    $\overset{\text{N1}}{=}$ `elem e (nub (y:ys))`

3. Queremos ver que `Eq a =>` $\forall$ `xs :: [a]`, vale `P(xs)`:
   $\forall$ `ys :: [a]`, $\forall$ `e :: a`, `elem e (union xs ys) = (elem e xs) || (elem e ys)`.
   Si no vale `Eq a`, ya vale la implicación por `Bool`, porque `False` $\rightarrow$ `algo = True`.
   Así que continuamos suponiendo que vale `Eq a`.
   Queremos ver que para todo `xs :: [a]`, vale `P(xs)`.
   Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])` y para todo `x :: a`, `xs :: [a]`, `P(xs)` $\Rightarrow$ `P(x:xs)`.

   - Caso base: `xs = []`

     `elem e (union [] ys)`

     $\overset{\text{U0}}{=}$ `elem e (nub ([] ++ ys))`

     $\overset{++}{=}$ `elem e (nub ys)`

     $\overset{\text{item b)}}{=}$ `elem e ys`

     $\overset{\text{Bool}}{=}$ `False || elem e ys`

     $\overset{\text{elem0}}{=}$ `(elem e []) || (elem e ys)`

- Paso inductivo: `xs = z:zs`. Supongamos como hipótesis inductiva (HI) que `elem e (union zs ys) = elem e zs || elem e ys`. Queremos ver que vale para `z:zs`.

  `elem e (union (z:zs) ys)`

  $\overset{\text{U0}}{=}$ `elem e (nub ((z:zs) ++ ys))`

  $\overset{\text{++1}}{=}$ `elem e (nub (z : (zs ++ ys)))`

  $\overset{\text{N1}}{=}$ `elem e (z : filter (\y → z /= y) (nub (zs ++ ys)))`

  $\overset{\text{elem1}}{=}$ `e == z || elem e (filter (\y → z /= y) (nub (zs ++ ys)))`

  $=$ *

  Consideramos los dos casos según el lema de generación de `Bool`:

  - Caso `e == z = True`:

    `(*) = True || elem e (filter (\y → z /= y) (nub (zs ++ ys)))`

    $\overset{||}{=}$ `True`

    $=$ `True || (elem e zs || elem e ys)`

    $=$ `e == z || (elem e zs || elem e ys)`

    $=$ `(e == z || elem e zs) || elem e ys`

    $\overset{\text{elem1}}{=}$ `(elem e (z:zs)) || (elem e ys)`

  - Caso `e == z = False`:

    `(*) = False || elem e (filter (\y → z /= y) (nub (zs ++ ys)))`

    $=$ `elem e (filter (\y → z /= y) (nub (zs ++ ys)))`

    $\overset{\text{item a)}}{=}$ `elem e (nub (zs ++ ys)) && (\y → z /= y) e`

    $\overset{\beta}{=}$ `elem e (nub (zs ++ ys)) && z /= e`

    $=$ `elem e (nub (zs ++ ys))`

    $\overset{\text{U0}}{=}$ `elem e (union zs ys)`

    $\overset{\text{HI}}{=}$ `elem e zs || elem e ys`

    $=$ `False || (elem e zs || elem e ys)`

    $=$ `e == z || (elem e zs || elem e ys)`

    $=$ `(e == z || elem e zs) || elem e ys`

    $\overset{\text{elem1}}{=}$ `(elem e (z:zs)) || (elem e ys)`

- Probar lema: `(p1 || p2) || p3 = p1 || (p2 || p3)`
  Aplicamos análisis por casos sobre `p1` (ley de `Bool`):

  - Caso `p1 = True`:

    `(True || p2) || p3` $=$ `True || p3` $=$ `True` $=$ `True || (p2 || p3)` $=$ `p1 || (p2 || p3)`

  - Caso `p1 = False`:

    `(False || p2) || p3` $=$ `p2 || p3`

    Ahora analizamos `p2`:

    * Caso `p2 = True`:

      `p2 || p3` $=$ `True` $=$ `True || p3` $=$ `p2 || p3` $=$ `False || (p2 || p3)` $=$ `p1 || (p2 || p3)`

    * Caso `p2 = False`:

      `p2 || p3` $=$ `p3` $=$ `False || p3` $=$ `p2 || p3` $=$ `False || (p2 || p3)` $=$ `p1 || (p2 || p3)`

4. Queremos ver que `Eq a =>` $\forall$ `xs :: [a]`, vale `P(xs)`:
   $\forall$ `ys :: [a]`, $\forall$ `e :: a`, `elem e (intersect xs ys) = (elem e xs) && (elem e ys)`.
   Si no vale `Eq a`, ya vale la implicación por `Bool`, porque `False` $\rightarrow$ `algo = True`.
   Así que continuamos suponiendo que vale `Eq a`.
   Queremos ver que para todo `xs :: [a]`, vale `P(xs)`.
   Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])` y para todo `x :: a`, `xs :: [a]`, `P(xs)` $\Rightarrow$ `P(x:xs)`.

   - Caso base: `xs = []`

     `elem e (intersect [] ys)`

     $\overset{\text{I0}}{=}$ `elem e (filter (\e → elem e ys) [])`

     $\overset{\text{filter0}}{=}$ `elem e []`

     $\overset{\text{def}}{=}$ `False`

     $\overset{\&\&}{=}$ `False && elem e ys`

     $\overset{\text{elem0}}{=}$ `(elem e []) && (elem e ys)`

   - Paso inductivo: `xs = z:zs`. Supongamos como hipótesis inductiva (HI) que `elem e (intersect zs ys) = elem e zs && elem e ys`. Queremos ver que vale para `z:zs`.

     `elem e (intersect (z:zs) ys)`

     $\overset{\text{I0}}{=}$ `elem e (filter (\e → elem e ys) (z:zs))`

     $\overset{\text{filter1},\beta}{=}$ `elem e (if elem z ys then z :filter (\e → elem e ys) zs else filter (\e → elem e ys) zs)`

     $=$ `*`

     Consideramos los casos según el valor de `elem z ys` por el lema de generación de Bool:

     – Caso `elem z ys = True`:

       `(*) = elem e (z : filter (\e → elem e ys) zs)`

       $\overset{\text{elem1}}{=}$ `e == z || elem e (filter (\e → elem e ys) zs)`

       $=$ `**`

       Tenemos que volver a separar en casos por lema de generación de Bool, `e == z = True` o `e == z = False`

       ∗ Caso `e == z = True`:

         `(**) = True || elem e ...`

         $=$ `True`

         $=$ `elem e ys`

         $=$ `(True || elem e zs) && elem e ys`

         $=$ `(e == z || elem e zs) && elem e ys`

         $\overset{elem1}{=}$ `(elem e (z:zs)) && elem e ys`

       ∗ Caso `e == z = False`:

         `(**) = False || elem e (filter ...)`

         $=$ `elem e (filter (\e → elem e ys) zs)`

         $\overset{\text{I0}}{=}$ `elem e (intersect zs ys)`

         $\overset{\text{HI}}{=}$ `elem e zs && elem e ys`

         $=$ `(False || elem e zs) && elem e ys`

         $=$ `(e == z || elem e zs) && elem e ys`

         $\overset{elem1}{=}$ `(elem e (z:zs)) && elem e ys`

– Caso `elem z ys = False`:

$$\begin{aligned}
&\texttt{(*) = elem e (filter (\textbackslash e} \to \texttt{elem e ys) zs)}\\
&\overset{I0}{=} \texttt{elem e (intersect zs ys)}\\
&\overset{\text{HI}}{=} \texttt{elem e zs \&\& elem e ys}\\
&= \texttt{**}
\end{aligned}$$

Por lema de generación de Bool `e == z = True` o `e == z = False`:

* Caso `e == z = True`:

$$\begin{aligned}
&\texttt{(**) = elem e zs \&\& False} \quad (\text{porque } elemzys = False \text{ y } e == z)\\
&\overset{\&\&}{=} \texttt{False}\\
&\\
&= \texttt{(True || elem e zs) \&\& elem e ys}\\
&= \texttt{(e == z || elem e zs) \&\& elem e ys}\\
&\overset{elem1}{=} \texttt{(elem e (z:zs)) \&\& elem e ys}
\end{aligned}$$

* Caso `e == z = False`:

$$\begin{aligned}
&\texttt{(**) = elem e zs \&\& elem e ys}\\
&= \texttt{(False || elem e zs) \&\& elem e ys}\\
&= \texttt{(e == z || elem e zs) \&\& elem e ys}\\
&\overset{elem1}{=} \texttt{(elem e (z:zs)) \&\& elem e ys}
\end{aligned}$$

5. Queremos mostrar que `length (union xs ys)` $\neq$ `length xs + length ys` en general, dando un contrajemplo.
   Tomamos: `xs = [1]`, `ys = [1]`

$$\begin{aligned}
&\texttt{length [1]} = 1\\
&\texttt{length [1]} = 1\\
&\Rightarrow \texttt{length xs + length ys} = 1 + 1 = 2\\
&\texttt{union [1] [1]}\\
&\overset{\text{U0}}{=} \texttt{nub ([1] ++ [1])}\\
&= \texttt{nub (1 : [1])}\\
&\overset{\text{N1}}{=} \texttt{1 : filter (\textbackslash y} \to \texttt{1 /= y) [1]}\\
&= \texttt{1 : filter (\textbackslash y} \to \texttt{1 /= y) []}\\
&= \texttt{1 : []}\\
&= \texttt{[1]}\\
&\Rightarrow \texttt{length (union [1] [1])} = 1\\
&\Rightarrow \texttt{1} \neq \texttt{2 = length xs + length ys}
\end{aligned}$$

   Por lo tanto, `length (union xs ys) = length xs + length ys` no se cumple en general.

6. Queremos ver que `Eq a =>` $\forall$ `xs :: [a]`, vale `P(xs)`:
   $\forall$ `ys :: [a]`, `length (union xs ys)` $\leq$ `length xs + length ys`.
   Si no vale `Eq a`, ya vale la implicación por `Bool`, porque `False` $\to$ `algo = True`.
   Así que continuamos suponiendo que vale `Eq a`.
   Queremos ver que para todo `xs :: [a]`, vale `P(xs)`.
   Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])` y para todo `x :: a`, `xs :: [a]`, `P(xs)` $\Rightarrow$ `P(x:xs)`.

- Caso base: `xs = []`

  `length (union [] ys)`
  $\overset{U0}{=}$ `length (nub ([] ++ ys))`
  $\overset{++0}{=}$ `length (nub ys)`
  $\overset{Lema1}{\leq}$ `length ys` ()
  $=$ `length ys + 0`
  $\overset{length0}{=}$ `length ys + length []`
  $\overset{asociatividad}{=}$ `length [] + length ys`

- Paso inductivo: `xs = z:zs`. Supongamos como HI `P(zs)` que para todo `ys :: [a]`, `length (union zs ys)` $\leq$ `length zs + length ys`.
  Queremos ver que `length (union (z:zs) ys)` $\leq$ `length (z:zs) + length ys`.

  `length (union (z:zs) ys)`
  $\overset{U0}{=}$ `length (nub ((z:zs) ++ ys))`
  $\overset{++1}{=}$ `length (nub (z : (zs ++ ys)))`
  $\overset{N1}{=}$ `length (z : filter (\y` $\rightarrow$ `z /= y) (nub (zs ++ ys)))`
  $\overset{length1}{=}$ `1 + length (filter (\y` $\rightarrow$ `z /= y) (nub (zs ++ ys)))`
  $\overset{Lema2}{\leq}$ `1 + length (nub (zs ++ ys))`
  $\overset{HI}{\leq}$ `1 + length zs + length ys`
  $\overset{length1}{=}$ `length (z:zs) + length ys`

Lema1: $\forall$ `ys :: [a]`, vale `P(ys) : length (nub ys)` $\leq$ `length ys` Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])` y para todo `y :: a`, `ys :: [a]`, `P(ys)` $\Rightarrow$ `P(y:ys)`.

- Caso base: `ys = []`

  `length (nub [])`
  $\overset{N0}{=}$ `length []`
  $\leq$ `length []` (por $a \leq a$ para todo $a : Int$)

- Paso inductivo: `ys = x:xs`. Supongamos como HI que `length (nub xs)` $\leq$ `length xs`.

  `length (nub (x:xs))`
  $\overset{N1}{=}$ `length (x : filter (\y` $\rightarrow$ `x /= y) (nub xs))`
  $\overset{length1}{=}$ `1 + length (filter (\y` $\rightarrow$ `x /= y) (nub xs))`
  $\overset{Lema2}{\leq}$ `1 + length (nub xs)`
  $\overset{HI}{\leq}$ `1 + length xs`
  $\overset{length1}{=}$ `length (x:xs)`

Lema2: $\forall$ `ys :: [a]`, `length (filter p ys)` $\leq$ `length ys` Por principio de inducción sobre listas, esto es equivalente a ver que se cumple `P([])` y para todo `y :: a`, `ys :: [a]`, `P(ys)` $\Rightarrow$ `P(y:ys)`.

- Caso base: `ys = []`

  `length (filter p [])`
  $\overset{filter0}{=}$ `length []`
  $\leq$ `length []`

- Paso inductivo: `ys = x:xs`. Supongamos como HI que `length (filter p xs)` $\leq$ `length xs`.

  `length (filter p (x:xs))`

  $\overset{\text{filter1}}{=}$ `length (if p x then x : filter p xs else filter p xs)`

  $=$ `*`

  Por lema de generación de Bool separamos en dos casos :

  - Caso `p x = True`:

    `(*) = length (x : filter p xs)`

    $\overset{\text{length1}}{=}$ `1 + length (filter p xs)`

    $\overset{\text{HI}}{\leq}$ `1 + length xs`

    $\overset{\text{length1}}{=}$ `length (x:xs)`

  - Caso `p x = False`:

    `(*) = length (filter p xs)`

    $\overset{\text{HI}}{\leq}$ `length xs`

    $=$ `0 + length xs`

    $\leq$ `1 + length xs`

    $\overset{\text{length1}}{=}$ `length (x:xs)`

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:

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

Primero, nos traemos las funciones :

```haskell
data AB a = Nil | Bin (AB a) r (AB a)

altura :: AB a -> Int
{A} altura = foldAB 0 (\recI r recD -> 1 + max recI recD)

cantNodos :: AB a -> Int
{C} cantNodos = foldAB 0 (\recI r recD -> 1 + recI + recD)
```

Queremos ver que $\forall$ `x :: AB a`, se cumple `P(x)`: `altura x` $\leq$ `cantNodos x`.
Por inducción estructural sobre `AB`, esto es equivalente a ver que se cumple `P(Nil)`
y que para todo `i :: AB a`,`d :: AB a`,`r :: a`, `P(i)` $\wedge$ `P(d)`$\Rightarrow$ `P(Bin i r d)`.

- Caso base: `x = Nil`

  `altura Nil`

  $\overset{\text{A}}{=}$ `foldAB 0 (\recI r recD` $\rightarrow$ `1 + max recI recD) Nil`

  $\overset{\text{F0}}{=}$ `0`

  $\leq$ `0` (porque $a \leq a$ para todo a)

  $\overset{\text{F0}}{=}$ `foldAB 0 (\recI r recD` $\rightarrow$ `1 + recI + recD) Nil`

  $=$ `cantNodos Nil`

- Paso inductivo: `x = Bin i r d`. Suponemos que vale `P(i) y P(d)`, o sea como HI que:

  `altura i` $\leq$ `cantNodos i`

  `altura d` $\leq$ `cantNodos d`

$$\texttt{altura (Bin i r d)}$$
$$\stackrel{A}{=} \texttt{foldAB 0 (\textbackslash recI r recD} \to \texttt{1 + max recI recD) (Bin i r d)}$$
$$\stackrel{F1}{=} (\texttt{\textbackslash recI r1 recD} \to 1 + \texttt{max recI recD})(\texttt{foldAB 0 (\textbackslash recI2 r2 recD2} \to \texttt{1 + max recI2 recD2) i})$$
$$\texttt{r (foldAB 0 (\textbackslash recI3 r4 recD3} \to \texttt{1 + max recI3 recD3) d})$$
$$\stackrel{A}{=} (\texttt{\textbackslash recI r1 recD} \to 1 + \texttt{max recI recD})\ \texttt{altura i r altura d}$$
$$\stackrel{\beta}{=} (\texttt{\textbackslash recI r1} \to 1 + \texttt{max recI (altura d)})\ \texttt{altura i r}$$
$$\stackrel{\beta}{=} (\texttt{\textbackslash recI} \to 1 + \texttt{max recI (altura d)})\ \texttt{altura i}$$
$$\stackrel{\beta}{=} 1 + \texttt{max (altura i) (altura d)}$$
$$= 1 + \texttt{if altura i} \geq \texttt{altura d then altura i else altura d}$$
$$= *$$

$$= 1 + \texttt{cantNodos i} + \texttt{cantNodos d}$$
$$\stackrel{\beta}{=} (\texttt{\textbackslash recI} \to 1 + \texttt{recI} + \texttt{cantNodos d})\ \texttt{cantNodos i}$$
$$\stackrel{\beta}{=} (\texttt{\textbackslash recI r1} \to 1 + \texttt{recI} + \texttt{cantNodos d})\ \texttt{cantNodos i r}$$
$$\stackrel{\beta}{=} (\texttt{\textbackslash recI r1 recD} \to 1 + \texttt{recI} + \texttt{recD})\ \texttt{cantNodos i r cantNodos d}$$
$$\stackrel{C}{=} (\texttt{\textbackslash recI r1 recD} \to 1 + \texttt{recI} + \texttt{recD})(\texttt{foldAB 0 (\textbackslash recI r2 recD} \to \texttt{1 + recI + recD) i})\ \texttt{r}$$
$$(\texttt{foldAB 0 (\textbackslash recI r3 recD} \to \texttt{1 + recI + recD) d})$$
$$\stackrel{F1}{=} \texttt{foldAB 0 (\textbackslash recI r recD} \to \texttt{1 + recI + recD) (Bin i r d)}$$
$$\stackrel{C}{=} \texttt{cantNodos (Bin i r d)}$$

Queremos ver por lema de generación de Bool, $\texttt{altura i} \geq \texttt{altura d}$ es True o False.

- Caso `True`:

$$\texttt{(*) = 1 + altura i}$$
$$= \texttt{1 + altura i + 0}$$
$$\leq \texttt{1 + altura i + altura d} \quad \text{(porque altura d} \geq 0 \text{ por lema)}$$
$$\stackrel{HI}{\leq} \texttt{1 + cantNodos i + altura d}$$
$$\stackrel{HI}{\leq} \texttt{1 + cantNodos i + cantNodos d}$$

- Caso `False`:

$$\texttt{(*) = 1 + altura d}$$
$$= \texttt{1 + 0 + altura d} \quad \text{(asociatividad)}$$
$$\leq \texttt{1 + altura i + altura d} \quad \text{(porque altura i} \geq 0 \text{ por lema)}$$
$$\stackrel{HI}{\leq} \texttt{1 + cantNodos i + altura d}$$
$$\stackrel{HI}{\leq} \texttt{1 + cantNodos i + cantNodos d}$$

Por lo tanto:

$$\texttt{altura (Bin i r d)} \leq \texttt{1 + cantNodos i + cantNodos d}$$
$$\stackrel{def}{=} \texttt{cantNodos (Bin i r d)}$$

Lema: Para todo `ab :: AB a`, se cumple que $\texttt{altura ab} \geq \texttt{0}$.
Este resultado se prueba en el ítem 10 a).

3.2 Ejercicio 10

Dada la siguiente función:

```
truncar :: AB a -> Int -> AB a
{T0} truncar Nil _ = Nil
{T1} truncar (Bin i r d) n =  if n == 0 then Nil  else Bin (truncar i (n-1)) r (truncar d (n-1))
```

Y los siguientes lemas:

1. $\forall x :: Int.\ \forall y :: Int.\ \forall z :: Int.\ \max(\min(x, y), \min(x, z)) = \min(x, \max(y, z))$
2. $\forall x :: Int.\ \forall y :: Int.\ \forall z :: Int.\ z + \min(x, y) = \min(z + x, z + y)$

Demostrar las siguientes propiedades:

i. $\forall t :: AB\ a.\ \text{altura}(t) \geq 0$

ii. $\forall t :: AB\ a.\ \forall n :: Int.\ (n \geq 0 \Rightarrow (\text{altura}(\text{truncar}\ t\ n) = \min n\ (\text{altura}\ t)))$

3.3 Ejercicio 11

Considerar las siguientes funciones:

```
inorder :: AB a -> [a]
{I0} inorder = foldAB [] (\ri x rd -> ri ++ (x:rd))

elemAB :: Eq a => a -> AB a -> Bool
{A0} elemAB e = foldAB False (\ri x rd -> (e == x) || ri || rd)

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

3.4 Ejercicio 12

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

```
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. $Num\ a => \forall p :: Polinomio\ a. \forall q :: Polinomio\ a. \forall r :: a.\quad (esRaiz\ r\ p \implies esRaiz\ r\ (Prod\ p\text{¸}; q))$
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. $Num\ a => \forall p :: Polinomio\ a.\quad (sinConstantesNegativas\ p \implies sinConstantesNegativas\ (derivado\ p))$