ej rafa reglas de congruencia

Si $M \to M'$:

$$M \bullet N \longrightarrow M' \bullet N$$

$$V \bullet M \longrightarrow V \bullet M'$$

$$\text{prox}(M) \longrightarrow \text{prox}(M')$$

$$\text{desencolar}(M) \longrightarrow \text{desencolar}(M')$$

$$\text{case } M \text{ of } \langle\rangle \rightsquigarrow N_1 \,/\, c \bullet x \rightsquigarrow N_2 \to \text{case } M' \text{ of } \langle\rangle \rightsquigarrow N_1 \,/\, c \bullet x \rightsquigarrow N_2$$

$$\text{Case } \langle\rangle_{Nat} \bullet \underline{1} \bullet \underline{0} \text{ of } \langle\rangle \rightsquigarrow \text{próximo}(\langle\rangle_{Bool}) \,/\, c \bullet x \rightsquigarrow \text{isZero}(x)$$

$$\xrightarrow[\text{case vacío}]{} \text{isZero}(x)\{x := \underline{0}\} = \text{isZero}(\underline{0})$$

$$\xrightarrow[\text{isZero } \underline{0}]{} \text{true}$$

$$\text{último}_\tau \stackrel{\text{def}}{=} \lambda x : Cola_\tau.\ \text{case } x \text{ of } \langle\rangle \rightsquigarrow \text{prox}(\langle\rangle_\tau) \,/\, c \bullet u \rightsquigarrow u$$

Permite tipar correctamente ya que $\vdash \text{prox}(\langle\rangle_\tau) : \tau$ sin construir ningún valor de tipo $\tau$.

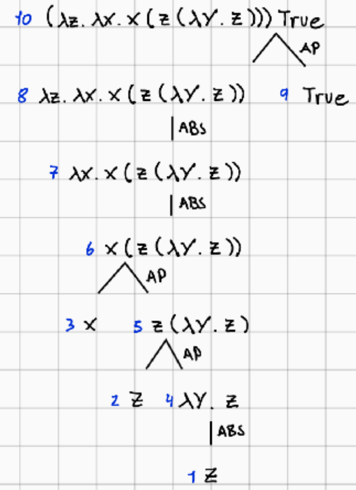
$$\frac{\Gamma \vdash x : Cola_\tau}{\Gamma \vdash \text{último}_\tau\, x : \tau}$$

ej 20 reglas de congruencia

Reglas de congruencia

Si $M \to N$ entonces:

| | | | |
|---|---|---|---|
| $\langle M, O \rangle$ | $\longrightarrow$ | $\langle N, O \rangle$ | $PI_c$ |
| $\langle V, M \rangle$ | $\longrightarrow$ | $\langle V, N \rangle$ | $PD_c$ |
| $\pi_1(M)$ | $\longrightarrow$ | $\pi_1(N)$ | $\pi_{1c}$ |
| $\pi_2(M)$ | $\longrightarrow$ | $\pi_2(N)$ | $\pi_{2c}$ |




1) $z : t_1 \vdash z : t_1$

2) $z : t_2 \vdash z : t_2$

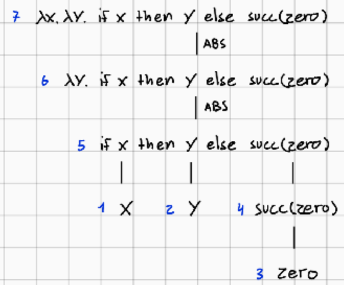
3) $x : t_3 \vdash x : t_3$

4) $z : t_1 \vdash \lambda y : t_4 .\, z : t_4 \to t_1$

5) $S = mgu\{ t_2 \doteq (t_4 \to t_1) \to t_5,\ t_2 \doteq t_1 \}$

$= mgu\{ t_1 \doteq (t_4 \to t_1) \to t_5 \}$ elim $\{ t_2 := t_1 \}$

= Falla por Occurs-check




1) $x : t_1 \vdash x : t_1$

2) $y : t_2 \vdash y : t_2$

3) $\vdash zero : Nat$

4) $S = mgu\{ Nat \doteq Nat \} = \emptyset$

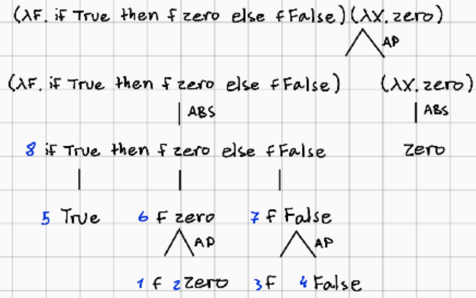
$\vdash succ(zero) : Nat$

5) $S = mgu\{ t_1 \doteq Bool,\ t_2 \doteq Nat \} = \{ t_1 := Bool,\ t_2 := Nat \}$

$x : Bool,\ y : Nat \vdash$ if $x$ then $y$ else $succ(zero) : Nat$

6) $x : Bool \vdash \lambda y : Nat.$ if $x$ then $y$ else $succ(zero) : Nat \to Nat$

7) $\vdash \lambda x : Bool.\ \lambda y : Nat.$ if $x$ then $y$ else $succ(zero) : Bool \to Nat \to Nat$




1) $f : t_1 \vdash f : t_1$

2) $\vdash zero : Nat$

3) $f : t_2 \vdash f : t_2$

4) $\vdash False : Bool$

5) $\vdash True : Bool$

6) $S = mgu\{ t_1 \doteq Nat \to t_3 \} = \{ t_1 := Nat \to t_3 \}$

$f : Nat \to t_3 \vdash f\ zero : t_3$

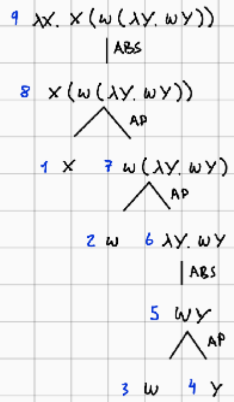
7) $S = mgu\{ t_2 \doteq Bool \to t_4 \} = \{ t_2 := Bool \to t_4 \}$

$f : Bool \to t_4 \vdash f\ False : t_4$

8) $S = mgu\{ Bool \doteq Bool,\ t_3 \doteq t_4,\ Nat \to t_3 \doteq Bool \to t_4 \}$

$= mgu\{ Bool \doteq Bool,\ t_3 \doteq t_4,\ Nat \doteq Bool \}$ decompose

= Falla por Clash




1) $x : t_1 \vdash x : t_1$

2) $w : t_2 \vdash w : t_2$

3) $w : t_3 \vdash w : t_3$

4) $y : t_4 \vdash y : t_4$

5) $S = mgu\{ t_3 \doteq t_4 \to t_5 \} = \{ t_3 := t_4 \to t_5 \}$

$w : t_4 \to t_5,\ y : t_4 \vdash w y : t_5$

6) $w : t_4 \to t_5 \vdash \lambda y : t_4.\ w y : t_4 \to t_5$

7) $S = mgu\{ t_2 \doteq (t_4 \to t_5) \to t_6,\ t_2 \doteq t_4 \to t_5 \}$

$= mgu\{ t_4 \to t_5 \doteq (t_4 \to t_5) \to t_6 \}$ elim $\{ t_2 := t_4 \to t_5 \}$

$= mgu\{ t_4 \doteq t_4 \to t_5,\ t_5 \doteq t_6 \}$ decompose

= Falla por Occurs-Check

2 - LU 26/23 (195)

1 * (Si no se cumple Eq a, la propiedad ya sería verdadera, ya que Eq a => ...)

## EJERCICIO 2 a

Vamos a probar la propiedad para todo árbol usando inducción estructural sobre t :: AEB a. La propiedad es la siguiente:

P(t) = ∀xs :: [a]. esPreRama t xs => length xs ≤ altura t.

(A partir de ahora ignoro el ∀xs :: [a], pero tengo en cuenta que está presente. También asumo Eq a)

Para que la propiedad sea válida por inducción, debe valer tanto para los casos no recursivos (casos base) como los recursivos. Veamos que esto ocurre.

**Casos base.**

"a" variable, no es lo mismo que la a del tipo

- P(Hoja a) = esPreRama (Hoja a) $\overset{\forall xs}{=>}$ length xs ≤ altura (Hoja a)

Desarrollando la izquierda de la implicación:

esPreRama (Hoja a)$^{xs}$ $\overset{E0}{=}$ (\xs → null xs || (xs == [x])) xs $\overset{B}{=}$ null xs || xs == [x].

Desarrollando la parte derecha:

length xs ≤ altura (Hoja a)

altura (Hoja a) $\overset{A0}{=}$ 1 → length xs ≤ 1.

Probemos la implicación asumiendo el antecedente y viendo que el precedente se cumple (técnica clásica).

Además, utilicemos extensionalidad sobre listas para separar en casos.

- Si xs = [],
  null [] || [] == [x] $\overset{N0}{=}$ True || [] == [x] $\overset{\text{PROP MATE}}{=}$ True
  Además, length [] $\overset{L0}{=}$ 0 ≤ 1 ~ T/ndadero por matemática
  Como siempre valen el antecedente y precedente, este caso se cumple.

- Si xs = x : xs,
  null x : xs || x : xs == [x]. $\overset{N1}{=}$ False || x : xs == [x]. *2
  En caso de que x : xs == [x] (haciendo True la expresión),
  veamos que ocurre con el precedente:
  length [x] $\overset{L1}{=}$ 1 + length [] $\overset{L0}{=}$ 1+0 = 1 ≤ 1. Vale también en este caso.

(*) Esto lo podemos hacer por extensionalidad de Bool. *2

Por ende, se cumple el caso base.

**Caso recursivo**

~~P(Bin (AEB a) a (AEB a)) = esPreRama (Bin a (AEB a) a (AEB b)) xs => length xs ≤ altura (Bin a (AEB a) a (AEB b))~~

- P(Bin i r d) = esPreRama (Bin i r d) xs => length xs ≤ altura (Bin i r d)

Además, como HI tenemos que valen P(i) y P(d).
→ Por ej. P(i) = esPreRama i xs => length xs ≤ altura i

Desarrollando la izquierda,

esPreRama (Bin i r d) xs $\overset{E1+B}{=}$
null xs || (r == head xs && (esPreRama i (tail xs) || esPreRama d (tail xs))

*2 Lo que se está haciendo acá es, por extensionalidad, separar por x:xs == [x] = True o False. Solo nos interesa el caso True ya que con False no se cumple el antecedente, haciendo valer la propiedad.

3 - LU 26/23 (145)

Desarrollando a la derecha,

- length xs ≤ altura (Bin i r d),
  $\overset{A1}{=}$ 1 + max (altura i) (altura d)
  → length xs ≤ 1 + max (altura i) (altura d)

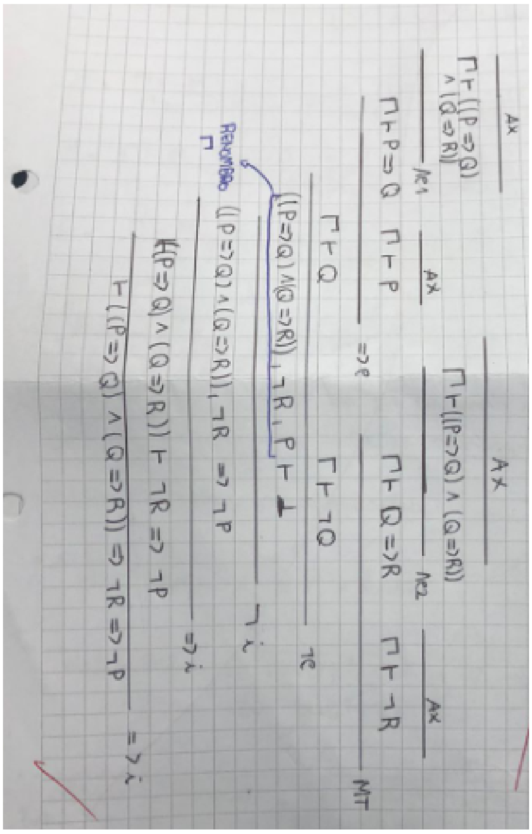
Utilizaremos nuevamente el método de prueba de ver que el precedente es válido cuando el antecedente lo es. Además, dividiré en casos por extensionalidad de listas.

- xs = [] :
  null [] || (r == head xs && (esPreRama i (tail [])) || esPreRama d (tail [])) $\overset{N0}{=}$ True || .... $\overset{\text{PROP MATE}}{=}$ True
  En el precedente, veamos que se cumple siempre que xs = [] para que valga en este caso la implicación.
  length [] ≤ 1 + max (altura i) (altura d)   altura ≥ 0 y L0 y max
  0 ≤ 1 ✓. Vale para este caso.

- xs = x : xs :
  null x:xs || (r == head x:xs && (esPreRama i (tail x:xs) || esPreRama d (tail xs)) $\overset{N1 + \text{PROP MATE } *1}{=}$ r == head xs && ..... (copian lo que falta).
  $\overset{T \text{ y } H\ *2}{=}$ (r == x) && (esPreRama i xs || esPreRama d xs)

Dado que tenemos un &&, podemos usar extensionalidad de bool sobre (r == x) y (esPreRama i xs || esPreRama d xs). En caso de que respectivamente sean True - False, False - False o False - True, el antecedente es falso, por lo que la propiedad vale en estos casos. Veamos el caso True - True para ver que ocurre con el consecuente:

(*1) Para sacar False || .. uso prop. matemática del or.



EJERCICIO 3

a) Reglas de tipado

- $\dfrac{}{\Gamma \vdash Vacío_{\sigma,\tau} : Dicc(\sigma, \tau)}$ Ax-DICC

- $\dfrac{\Gamma \vdash M : Dicc(\sigma, \tau) \quad \Gamma \vdash N : \sigma \quad \Gamma \vdash O : \tau}{\Gamma \vdash definir(M, N, O) : Dicc(\sigma, \tau)}$ T-DEFINIR

- $\dfrac{\Gamma \vdash M : Dicc(\sigma, \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash def?(M, N) : Bool}$ T-DEF?

- $\dfrac{\Gamma \vdash M : Dicc(\sigma, \tau) \quad \Gamma \vdash N : \sigma}{\Gamma \vdash obtener(M, N) : \tau}$ T-OBTENER

b) Valores: $V ::= Vacío_{\sigma,\tau} \mid Definir(V, V, V)$

(Definir tiene valores en el segundo y tercer parámetro ya que es lo que me interesa almacenar).

Reglas de reducción small-step:

- $\dfrac{}{def?(Vacío_{\sigma,\tau}, V) \to False}$ DEF?-VACIO (V valor)

- $\dfrac{}{def?(Definir(V_1, V_2, V_3), V_4) \to \text{if } V_4 == V_2 \text{ then True else } def?(V_1, V_4)}$ DEF? ($V_1, V_2, V_3, V_4$ valores)

- $\dfrac{}{obtener(Vacío_{\sigma,\tau}, V) \to obtener(Vacío_{\sigma,\tau}, V)}$ OBTENER-VACÍO

Podría no ponerse la regla y listo.

- $\dfrac{}{obtener(Definir(V_1, V_2, V_3), V_4) \to \text{if } V_4 == V_2 \text{ then } V_3 \text{ else } obtener(V_1, V_4)}$ OBTENER ($V_1, V_2, V_3, V_4$ valores)

Reducir a sí mismo hace que el programa se cuelgue (al nunca ~~[illegible]~~ terminar de reducirse) dando el comportamiento esperado por el enunciado. Además, cumple el tipado.

La cantidad de reglas de congruencia son 7 (3 para definir, 2 para def? y 2 para obtener)

c) $(\lambda d : Dicc(Nat, Bool).\ \text{if } def?(d, \underline{0}) \text{ then } obtener(d, \underline{0}) \text{ else } False)\ definir(Vacío_{Nat,Bool}, \underline{0}, true) \xrightarrow{\beta}$

$\text{if } def?(definir(Vacío_{Nat,Bool}, \underline{0}, true), \underline{0}) \text{ then } obtener(definir(Vacío_{Nat,Bool}, \underline{0}, True), \underline{0}) \text{ else } False \xrightarrow{\text{ifc + def?}}$

$\text{if } (\text{if } \overbrace{\underline{0} == \underline{0}}^{True} \text{ then True else } def?(Vacío_{Nat,Bool}, \underline{0})) \text{ then } obtener(definir(Vacío_{Nat,Bool}, \underline{0}, true), \underline{0}) \text{ else } False \xrightarrow{\text{ifc + ift}}$

$\text{if True then } obtener(definir(Vacío_{Nat,Bool}, \underline{0}, true), \underline{0}) \text{ else } False \xrightarrow{\text{ift}}$

$obtener(definir(Vacío_{Nat,Bool}, \underline{0}, true), \underline{0}) \xrightarrow{\text{OBTENER}}$

$\text{if } \overbrace{\underline{0} == \underline{0}}^{True} \text{ then true else } obtener(Vacío_{Nat,Bool}, \underline{0}) \xrightarrow{\text{ift}}$

true. (F)

Me salteé los casos en los if donde se debió reducir $0 == 0$ a true. Cuando marqué $\overbrace{0 == 0}^{True}$, en el primer caso, debió haber un paso intermedio aplicando ifc + ifc + Regla $0 == 0$. En el segundo se debió aplicar ifc + Regla $0 == 0$ en un paso intermedio.

Una mejor explicación de Gabi (jtp) en Discord: "Inducción = separación en casos + hipótesis inductiva para los casos inductivos. Extensionalidad = solamente separación en casos."

```
type Tono = Integer
type Duracion = Integer

data Melodia = Silencio Duracion | Nota Tono Duracion | Secuencia Melodia Melodia | Paralelo [Melodia]

foldMelodia:: (Duracion -> m) -> (Tono -> Duracion -> m ) -> (m -> m -> m) -> ([m] -> m) -> Melodia -> m

foldMelodia cSilencio cNota cSecuencia cParalelo m = case m of
        Silencio duracionSil -> cSilencio duracionSil
        Nota tono duracionNota -> cNota tono duracionNota
        Secuencia mel1 mel2 -> cSecuencia ( rec mel1) (rec mel2)
        Paralelo melLista -> cParalelo (map (rec melLista))
        where rec = foldMelodia cSilencio cNota cSecuencia cParalelo

duracionTotal m = foldMelodia id (\tono dur -> dur) (+) maximum


truncar =  foldMelodia (\m -> duracion -> case m of
        Silencio durSil  -> Silencio max(durSil,duracion)
        Nota tono durNota-> Nota tono max(durNota,duracion)
        Secuencia mel1 mel2 ->
        Paralelo melLista -> if duracionTotal m >
        )
```

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaaaaaaaaaazxczxcaAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

semantica denotacional

I) $\lambda x:Nat.\ zero$

$$[\![ \lambda x:Nat.\ zero ]\!]_v = V^{[\![Nat]\!]} \mapsto [\![zero]\!]_{v,x=V}$$
$$= V^{[\![Nat]\!]} \mapsto 0$$
$$= V^{\mathbb{N}} \mapsto 0$$

II) $\lambda x:Nat.\ (\lambda y:Nat.\ y)\ succ(x)$

$$[\![ \lambda x:Nat.\ (\lambda y:Nat.\ y)\ succ(x) ]\!]_v$$
$$= X^{[\![Nat]\!]} \mapsto [\![ (\lambda y:Nat.\ y)\ succ(x) ]\!]_{v,x=X}$$
$$= X^{[\![Nat]\!]} \mapsto [\![ (\lambda y:Nat.\ y) ]\!]_{v,x=X}\ [\![ succ(x) ]\!]_{v,x=X}$$
$$= X^{[\![Nat]\!]} \mapsto \left(Y^{[\![Nat]\!]} \mapsto [\![y]\!]_{v,x=X,y=Y}\right) [\![x]\!]_{v,x=X} + 1$$
$$= X^{\mathbb{N}} \mapsto \left(Y^{\mathbb{N}} \mapsto Y\right)\ X+1$$
$$= X^{\mathbb{N}} \mapsto X+1$$

u

```haskell
-- Ejercicio 1

intercambiar :: (a,b) -> (b,a)
intercambiar (x,y) = (y,x) --{INT0}

espejar :: Either a b -> Either b a
espejar (Left x) = Right x --{E1}
espejar (Right x) = Left x --{E2}

asociarI :: (a,(b,c)) -> ((a,b),c)
asociarI (x,(y,z)) = ((x,y),z) --{AI}

asociarD :: ((a,b),c) -> (a,(b,c))
asociarD ((x,y),z) = (x,(y,z)) --{AD}

-- flip :: (a -> b -> c) -> b -> a -> c
-- flip f x y = f y x {F}

-- curry :: ((a,b) -> c) -> a -> b -> c
-- curry f x y = f (x,y) {CU}

-- uncurry :: (a -> b -> c) -> (a,b) -> c
-- uncurry f (x,y) = f x y {UC}

-- const :: a -> b -> a
-- const x _ = x {C0}

-- id :: a -> a
-- id x = x {ID}


-- Extensionalidad para PARES:
-- Si p :: (a, b), entonces ∃x :: a. ∃y :: b. p = (x, y).

-- Extensionalidad para SUMAS:
-- data Either a b = Left a | Right b
-- Si e :: Either a b, entonces
-- o bien ∃x :: a. e = Left x
-- o bien ∃y :: b. e = Right b


-- i. ∀ p::(a,b) . intercambiar (intercambiar p) = p

-- intercambiar (intercambiar p) = p {extensionalidad}
-- intercambiar (intercambiar (x,y)) = p {INT0}
-- intercambiar (y,x) = p {INT0}
-- (x,y) = p {extensionalidad}
-- p = p {queda demostrada la igualdad}

-- ii. ∀ p::(a,(b,c)) . asociarD (asociarI p) = p

-- asociarD (asociarI p) = p {extensionalidad}
-- asociarD (asociarI (x,(y,z))) = p {AI}
-- asociarD ((x,y),z) = p {AD}
-- (x,(y,z)) = p {extensionalidad}
-- p = p {queda demostrada la igualdad}

-- iii. ∀ p::Either a b . espejar (espejar p) = p

-- Si p::Either a b, entonces o bien ∃x :: a. e = Left x o bien ∃y :: b. e = Right b. Se consideran ambos casos.

-- Caso p = Left x
-- espejar (espejar (p)) = p {extensionalidad}
-- espejar (espejar (Left x)) = p {E1}
-- espejar (Right x) = p {E2}
-- Left x = p {extensionalidad}
-- p = p {queda demostrada la igualdad}

-- Caso p = Right x
-- espejar (espejar (p)) = p {extensionalidad}
-- espejar (espejar (Right x)) = p {E2}
-- espejar (Left x) = p {E1}
-- Right x = p {extensionalidad}
-- p = p {queda demostrada la igualdad}

-- iv. ∀ f::a->b->c . ∀ x::a . ∀ y::b . flip (flip f) x y = f x y

-- flip (flip f) x y = f x y {F}
-- flip f y x = f x y {F}
-- f x y = f x y {queda demostrada la igualdad}

-- v. ∀ f::a->b->c . ∀ x::a . ∀ y::b . curry (uncurry f) x y = f x y

-- curry (uncurry f) x y = f x y {CU}
-- uncurry f (x,y) = f x y {UC}
-- f x y = f x y {queda demostrada la igualdad}

-- Ejercicio 2

-- i. flip . flip = id

-- flip . flip = id {extensionalidad}
-- ∀ f::a->b->c . ∀ x::a . ∀ y::b (flip . flip) f x y = id f x y
-- (flip . flip) f x y = id f x y {.}
-- flip (flip f) x y = id f x y {F}
-- flip f y x = id f x y {F}
-- f x y = f x y {ID}
-- f = f {queda demostrada la igualdad}

-- ii. ∀ f::(a,b)->c . uncurry (curry f) = f

-- uncurry (curry f) = f {extensionalidad}
-- uncurry (curry f) (x,y) = f (x,y) {UC}
-- curry f x y = f (x,y) {CU}
-- f (x,y) = f (x,y) {queda demostrada la igualdad}

-- iii. flip const = const id

-- flip const = const id {extensionalidad}
-- ∀ x::a . ∀ y::b . flip const x y = const id x y
-- flip const x y = const id x y {F}
-- const y x = const id x y {C0}
-- y = const id x y {C0}
-- y = const id y {C0}
-- y = id y {ID}
-- y = y {queda demostrada la igualdad}

-- iv. ∀ f::a->b . ∀ g::b->c . ∀ h::c->d . ((h . g) . f) = (h . (g . f))
-- Sea la definición de la composición: (.) f g x = f (g x)

-- ((h . g) . f) = (h . (g . f)) {extensionalidad}
-- ∀ x::a . ((h . g) . f) x = (h . (g . f)) x
-- ((h . g) . f) x = (h . (g . f)) x {.}
-- (h . g) (f x) = (h . (g . f)) x {.}
-- h (g (f x)) = (h . (g . f)) x {.}
-- h ((g . f) x) = (h . (g . f)) x {.}
-- (h . (g . f)) x = (h . (g . f)) x {queda demostrada la igualdad}

-- Ejercicio 3

-- Considerar las siguientes funciones

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

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

-- append :: [a] -> [a] -> [a]
-- append [] ys = ys {A0}
-- append (x:xs) ys = x : append xs ys {A1}

-- (++) :: [a] -> [a] -> [a]
-- xs ++ ys = foldr (:) ys xs {++}
-- xs ++ [] = xs {++AUX1}
-- [] ++ ys = ys {++AUX2}

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

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

-- Agregadas y utilizadas en demostraciones posteriores:

-- map :: (a -> b) -> [a] -> [b]
-- map f = foldr ((:) . f) [] {M0}

-- map :: (a -> b) -> [a] -> [b]
-- map _ [] = [] {MA0}
-- map f (x:xs) = f x : map f xs {MA1}
```

-- filter :: (a -> Bool) -> [a] -> [a]
-- filter p = foldr (\x xs -> if p x then x : xs else xs) [] {F0}

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

-- head :: [a] -> a
-- head (x:_) = x {H0}

-- foldr :: (a -> b -> b) -> b -> [a] -> b
-- foldr f z [] = z {FR0}
-- foldr f z (x:xs) = f x (foldr f z xs) {FR1}

-- foldl :: (b -> a -> b) -> b -> [a] -> b
-- foldl f z [] = z {FL0}
-- foldl f z (x:xs) = foldl f (f z x) xs {FL1}

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

reverseFR :: [a] -> [a]
reverseFR = foldr (\x xs -> xs ++ [x]) [] --{RFR0}

-- i. ∀ xs::[a] . length (duplicar xs) = 2 * length xs

-- Predicado unario: P(xs) = length (duplicar xs) = 2 * length xs

-- Caso base: P([]) =
-- length (duplicar []) = 2 * length [] {D0}
-- length [] = 2 * length [] {L0}
-- 0 = 2 * 0 {aritmética}
-- 0 = 0 {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = length (duplicar xs) = 2 * length xs
-- Paso inductivo: P(x:xs) = length (duplicar (x:xs)) = 2 * length (x:xs)

-- length (duplicar (x:xs)) = 2 * length (x:xs) {D1}
-- length (x : x : duplicar xs) = 2 * length (x:xs) {L1}
-- 1 + length (x : duplicar xs) = 2 * length (x:xs) {L1}
-- 2 + length (duplicar xs) = 2 * length (x:xs) {HI}
-- 2 + 2 * length xs = 2 * length (x:xs) {L1}
-- 2 + 2 * length xs = 2 * (1 + length xs) {aritmética}
-- 2 + 2 * length xs = 2 + 2 * length xs {queda demostrada la igualdad}

-- ii. ∀ xs::[a] . ∀ ys::[a] . length (append xs ys) = length xs + length ys

-- Predicado unario: P(xs) = ∀ ys::[a] . length (append xs ys) = length xs + length ys

-- Caso base: P([]) =
-- ∀ ys::[a] . length (append [] ys) = length [] + length ys {A0}
-- ∀ ys::[a] . length ys = length [] + length ys {L0}
-- ∀ ys::[a] . length ys = 0 + length ys {aritmética}
-- ∀ ys::[a] . length ys = length ys {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ ys::[a] . length (append xs ys) = length xs + length ys
-- Paso inductivo: P(x:xs) = ∀ ys::[a] . length (append (x:xs) ys) = length (x:xs) + length ys

-- ∀ ys::[a] . length (append (x:xs) ys) = length (x:xs) + length ys {A1}
-- ∀ ys::[a] . length (x : append xs ys) = length (x:xs) + length ys {L1}
-- ∀ ys::[a] . 1 + length (append xs ys) = length (x:xs) + length ys {L1}
-- ∀ ys::[a] . 1 + length (append xs ys) = 1 + length xs + length ys {HI}
-- ∀ ys::[a] . 1 + length xs + length ys = 1 + length xs + length ys {aritmética}
-- ∀ ys::[a] . length xs + length ys = length xs + length ys {queda demostrada la igualdad}

-- iii. ∀ xs::[a] . ∀ f::(a->b) . length (map f xs) = length xs

-- Predicado unario: P(xs) = ∀ f::(a->b) . length (map f xs) = length xs

-- Caso base: P([]) =
-- ∀ f::(a->b) . length (map f []) = length [] {MA0}
-- ∀ f::(a->b) . length [] = length [] {L0}
-- 0 = 0 {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ f::(a->b) . length (map f xs) = length xs
-- Paso inductivo: P(x:xs) = ∀ f::(a->b) . length (map f (x:xs)) = length (x:xs)

-- ∀ f::(a->b) . length (map f (x:xs)) = length (x:xs) {M0}
-- ∀ f::(a->b) . length ((f x) : (map f xs)) = length (x:xs) {L1}
-- ∀ f::(a->b) . 1 + length (map f xs) = length (x:xs) {HI}
-- 1 + length xs = length (x:xs) {L1}
-- 1 + length xs = 1 + length xs {aritmética}
-- length xs = length xs {queda demostrada la igualdad}

-- iv. ∀ xs::[a] . ∀ p::a->Bool . ∀ e::a . (elem e (filter p xs) = True) ⇒ (elem e xs = True) (asumiendo Eq a)

-- Predicado unario: P(xs) = ∀ p::a->Bool . ∀ e::a . (elem e (filter p xs) = True) ⇒ (elem e xs = True)

-- Caso base: P([]) =
-- ∀ p::a->Bool . ∀ e::a . (elem e (filter p []) = True) ⇒ (elem e [] = True) {F0}
-- ∀ p::a->Bool . ∀ e::a . (elem e (foldr (\x xs -> if p x then x : xs else xs) []) = True) ⇒ (elem e [] = True) {FR0}
-- ∀ p::a->Bool . ∀ e::a . (elem e []) = True ⇒ (elem e [] = True) {queda demostrada la implicación}

-- ∀ p::a->Bool . ∀ e::a . False ⇒ (elem e [] = True) {queda demostrada la implicación}

-- Hipótesis inductiva: P(xs) = ∀ p::a->Bool . ∀ e::a . (elem e (filter p xs) = True) ⇒ (elem e xs = True)
-- Paso inductivo: P(x:xs) = ∀ p::a->Bool . ∀ e::a . (elem e (filter p (x:xs)) = True) ⇒ (elem e (x:xs) = True)

-- ∀ p::a->Bool . ∀ e::a . (elem e (filter p (x:xs)) = True) ⇒ (elem e (x:xs) = True) {F0}
-- ∀ p::a->Bool . ∀ e::a . (elem e (if p x then x : filter p xs else filter p xs) = True) ⇒ (elem e (x:xs) = True) {abrimos en casos}

-- La función p aplicada a x puede devolver True o False, por lo que se deben considerar ambos casos:

-- Caso 1: p x = True
-- ∀ p::a->Bool . ∀ e::a . (elem e (x : filter p xs) = True) ⇒ (elem e (x:xs) = True) {E0}
-- ∀ p::a->Bool . ∀ e::a . (foldr (\x b -> b || x == e) False (x : filter p xs) = True) ⇒ (elem e (x:xs) = True) {FR1}
-- ∀ p::a->Bool . ∀ e::a . ((foldr (\x b -> b || x == e) False (filter p xs) || x == e) = True) ⇒ (elem e (x:xs) = True) {E0}
-- ∀ p::a->Bool . ∀ e::a . ((elem e (filter p xs) || x == e) = True) ⇒ (elem e (x:xs) = True) {E0}
-- ∀ p::a->Bool . ∀ e::a . ((elem e (filter p xs) || x == e) = True) ⇒ (foldr (\x b -> b || x == e) False (x:xs) = True) {FR1}
-- ∀ p::a->Bool . ∀ e::a . ((elem e (filter p xs) || x == e) = True) ⇒ ((foldr (\x b -> b || x == e) False (xs) || x == e) = True) {E0}
-- ∀ p::a->Bool . ∀ e::a . ((elem e (filter p xs) || x == e) = True) ⇒ ((elem e xs || x == e) = True) {HI}

-- La hipótesis inductiva demuestra una implicación más fuerte que la que se pide demostrar, por lo que queda demostrada la implicación

-- Caso 2: p x = False
-- ∀ p::a->Bool . ∀ e::a . (elem e (filter p xs) = True) ⇒ (elem e (x:xs) = True) {HI}
-- ∀ p::a->Bool . ∀ e::a . (elem e (filter p xs) = True) ⇒ (elem e xs = True) ⇒ (elem e (x:xs) = True) {HI}

-- Por hipótesis inductiva, si vale elem e xs también vale elem e (x:xs), por lo que queda demostrada la implicación

-- v. ∀ xs::[a] . ∀ x::a . length (ponerAlFinal x xs) = 1 + length xs

-- Predicado unario: P(xs) = ∀ x::a . length (ponerAlFinal x xs) = 1 + length xs

-- Caso base: P([]) =
-- ∀ x::a . length (ponerAlFinal x []) = 1 + length [] {P0}
-- ∀ x::a . length (foldr (:) (x:[]) []) = 1 + length [] {FR0}
-- ∀ x::a . length (x:[]) = 1 + length [] {:}
-- ∀ x::a . length [x] = 1 + length [] {L0}
-- ∀ x::a . 1 = 1 + 0 {aritmética}
-- ∀ x::a . 1 = 1 {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ y::a . length (ponerAlFinal y xs) = 1 + length xs
-- Paso inductivo: P(x:xs) = ∀ y :: a . length (ponerAlFinal y (x:xs)) = 1 + length (x:xs)

-- ∀ y::a . length (ponerAlFinal y (x:xs)) = 1 + length (x:xs) {P0}
-- ∀ y::a . length (foldr (:) (y:[]) (x:xs)) = 1 + length (x:xs) {FR1}
-- ∀ y::a . length (x : foldr (:) (y:[]) xs) = 1 + length (x:xs) {L1}
-- ∀ y::a . 1 + length (foldr (:) (y:[]) xs) = 1 + length (x:xs) {P0}
-- ∀ y::a . 1 + length (ponerAlFinal y xs) = 1 + length (x:xs) {HI}
-- 1 + 1 + length xs = 1 + length (x:xs) {L1}
-- 1 + 1 + length xs = 1 + 1 + length xs {aritmética}
-- 2 + length xs = 2 + length xs {aritmética}
-- length xs = length xs {queda demostrada la igualdad}

-- vi. ∀ xs::[a] . ∀ x::a . head (reverse (ponerAlFinal x xs)) = x

-- Predicado unario: P(xs) = ∀ x::a . head (reverse (ponerAlFinal x xs)) = x

-- Caso base: P([]) =
-- ∀ x::a . head (reverse (ponerAlFinal x [])) = x {P0}
-- ∀ x::a . head (reverse (foldr (:) (x:[]) [])) = x {FR0}
-- ∀ x::a . head (reverse (x:[])) = x {:}
-- ∀ x::a . head (reverse [x]) = x {R0}
-- ∀ x::a . head [x] = x {H0}
-- ∀ x::a . x = x {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ y::a . head (reverse (ponerAlFinal y xs))) = y
-- Paso inductivo: P(x:xs) = ∀ y :: a . head (reverse (ponerAlFinal y (x:xs))) = y

-- ∀ y::a . head (reverse (ponerAlFinal y (x:xs))) = y {P0}
-- ∀ y::a . head (reverse (foldr (:) (y:[]) (x:xs))) = y {FR1}
-- ∀ y::a . head (reverse (x:(foldr (:) (y:[]) xs))) = y {P0}
-- ∀ y::a . head (reverse (x:(ponerAlFinal y xs))) = y {RFR0}
-- ∀ y::a . head (foldr (\x xs -> xs ++ [x]) [] (x:(ponerAlFinal y xs))) = y {FR1}

-- ∀ y::a . head (foldr (\x xs -> xs ++ [x]) [] (ponerAlFinal y xs) ++ [x]) = y {RFR0}
-- ∀ y::a . head (reverse (ponerAlFinal y xs) ++ [x]) = y {LEMA}
-- ∀ y::a . head (reverse (ponerAlFinal y xs)) = y {HI}
-- ∀ y::a . y = y {queda demostrada la igualdad}

-- Lema Auxiliar: ∀ xs::[a] . ∀ x::a . length xs > 0 ⇒ head (xs ++ ys) = head xs

-- Predicado unario: P(xs) = ∀ x::a . length xs > 0 ⇒ head (xs ++ ys) = head xs

-- Caso base: P([]) =

-- ∀ x::a . length [] > 0 ⇒ head ([] ++ ys) = head [] {L0}
-- ∀ x::a . 0 > 0 ⇒ head ([] ++ ys) = head [] {lógica}
-- ∀ x::a . False ⇒ head ([] ++ ys) = head [] {queda demostrada la implicación}

-- Hipótesis inductiva: P(xs) = ∀ x::a . length xs > 0 ⇒ head (xs ++ ys) = head xs
-- Paso inductivo: P(x:xs) = ∀ x::a . length (x:xs) > 0 ⇒ head ((x:xs) ++ ys) = head (x:xs)

-- ∀ x::a . length (x:xs) > 0 ⇒ head ((x:xs) ++ ys) = head (x:xs) {L1}
-- ∀ x::a . 1 + length xs > 0 ⇒ head ((x:xs) ++ ys) = head (x:xs) {H0}
-- ∀ x::a . 1 + length xs > 0 ⇒ head ((x:xs) ++ ys) = x {abrirmos en casos}

-- Caso 1: xs = []

-- ∀ x::a . 1 + length [] > 0 ⇒ head ((x:[]) ++ ys) = x {L0}
-- ∀ x::a . 1 + 0 > 0 ⇒ head ((x:[]) ++ ys) = x {:}
-- ∀ x::a . 1 + 0 > 0 ⇒ head (([x]) ++ ys) = x {aritmética}
-- ∀ x::a . 1 > 0 ⇒ head (([x]) ++ ys) = x {++}
-- ∀ x::a . 1 > 0 ⇒ head (foldr (:) ys [x]) = x {FR1}
-- ∀ x::a . 1 > 0 ⇒ head (x : foldr (:) ys []) = x {++}
-- ∀ x::a . 1 > 0 ⇒ head (x : foldr (:) ys []) = x {H0}
-- ∀ x::a . 1 > 0 ⇒ x = x {lógica}
-- ∀ x::a . True ⇒ x = x {queda demostrada la implicación}

-- Ejercicio 4

-- i. reverse . reverse = id

-- Predicado unario: P(xs) = reverse . reverse xs = id xs

-- Caso base: P([]) =

-- reverse . reverse [] = id [] {.}
-- reverse (reverse []) = id [] {R0}
-- reverse (foldl (flip (:)) [] []) = id [] {FL0}
-- reverse [] = id [] {R0}
-- foldl (flip (:)) [] [] = id [] {FL0}
-- [] = id [] {ID}
-- [] = [] {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = reverse . reverse xs = id xs
-- Paso inductivo: P(x:xs) = reverse . reverse (x:xs) = id (x:xs)

-- reverse . reverse (x:xs) = id (x:xs) {.}
-- reverse (reverse (x:xs)) = id (x:xs) {RFR0}
-- reverse (foldr (\x xs -> xs ++ [x]) [] (x:xs)) = id (x:xs) {FR1}
-- reverse (foldr (\x xs -> xs ++ [x]) [] (xs) ++ [x]) = id (x:xs) {RFR0}
-- reverse (reverse xs ++ [x]) = id (x:xs) {LEMA}
-- reverse [x] ++ reverse (reverse xs) = id (x:xs) {R0}
-- foldl (flip (:)) [] [x] ++ reverse (reverse xs) = id (x:xs) {FL1}
-- (foldl flip (:) (flip (:) [] x) []) ++ reverse (reverse xs) = id (x:xs) {FL1}
-- (flip (:) [] x) ++ reverse (reverse xs) = id (x:xs) {F}
-- (x:[]) ++ reverse (reverse xs) = id (x:xs) {:}
-- [x] ++ reverse (reverse xs) = id (x:xs) {HI}
-- [x] ++ xs = id (x:xs) {++}
-- foldr (:) xs [x] = id (x:xs) {FR1}
-- x : foldr (:) xs [] = id (x:xs) {FR0}
-- x:xs = id (x:xs) {ID}
-- x:xs = x:xs {queda demostrada la igualdad}

-- Lema Auxiliar: ∀ xs::[a] . ∀ ys::[a] . reverse (xs ++ ys) = reverse ys ++ reverse xs

-- Predicado unario: P(xs) = ∀ ys::[a] . reverse (xs ++ ys) = reverse ys ++ reverse xs

-- Caso base: P([]) =

-- ∀ ys::[a] . reverse ([] ++ ys) = reverse ys ++ reverse [] {++AUX2}
-- ∀ ys::[a] . reverse ys = reverse ys ++ [] {++AUX1}
-- ∀ ys::[a] . reverse ys = reverse ys {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ ys::[a] . reverse (xs ++ ys) = reverse ys ++ reverse xs
-- Paso inductivo: P(x:xs) = ∀ ys::[a] . reverse ((x:xs) ++ ys) = reverse ys ++ reverse (x:xs)

-- ∀ ys::[a] . reverse ((x:xs) ++ ys) = reverse ys ++ reverse (x:xs) {++}
-- ∀ ys::[a] . reverse (foldr (:) ys (x:xs)) = reverse ys ++ reverse (x:xs) {FR1}
-- ∀ ys::[a] . reverse ((:) x (foldr (:) ys xs)) = reverse ys ++ reverse (x:xs) {++}
-- ∀ ys::[a] . reverse (x : foldr (:) ys xs) = reverse ys ++ reverse (x:xs) {++}
-- ∀ ys::[a] . reverse (x : (xs ++ ys)) = reverse ys ++ reverse (x:xs) {RFR0}

-- ∀ ys::[a] . foldr (\y xs -> xs ++ [y]) [] (x : (xs ++ ys)) = reverse ys ++ reverse (x:xs) {FR1}
-- ∀ ys::[a] . (foldr (\y xs -> xs ++ [y]) [] (xs ++ ys)) ++ [x] = reverse ys ++ reverse (x:xs) {RFR0}
-- ∀ ys::[a] . (reverse (ys ++ xs)) ++ [x] = reverse ys ++ reverse (x:xs) {HI}
-- ∀ ys::[a] . reverse ys ++ reverse xs ++ [x] = reverse ys ++ reverse (x:xs) {RFR0}
-- ∀ ys::[a] . reverse ys ++ reverse xs ++ [x] = reverse ys ++ (foldr (\y xs -> xs ++ [y]) [] (x:xs)) {FR1}
-- ∀ ys::[a] . reverse ys ++ reverse xs ++ [x] = reverse ys ++ (foldr (\y xs -> xs ++ [y]) [] (xs)) ++ [x] {RFR0}
-- ∀ ys::[a] . reverse ys ++ reverse xs ++ [x] = reverse ys ++ reverse xs ++ [x] {queda demostrada la igualdad}

-- ii. append = (++)

-- Predicado unario: P(xs) = ∀ ys::[a] . append xs ys = xs ++ ys

-- Caso base: P([]) =

-- ∀ ys::[a] . append [] ys = [] ++ ys {A0}
-- ∀ ys::[a] . ys = [] ++ ys {++AUX2}
-- ∀ ys::[a] . ys = ys {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ ys::[a] . append xs ys = xs ++ ys
-- Paso inductivo: P(x:xs) = ∀ ys::[a] . append (x:xs) ys = (x:xs) ++ ys

-- ∀ ys::[a] . append (x:xs) ys = (x:xs) ++ ys {A1}
-- ∀ ys::[a] . x : append xs ys = (x:xs) ++ ys {HI}
-- ∀ ys::[a] . x : (xs ++ ys) = (x:xs) ++ ys {++}
-- ∀ ys::[a] . x : (xs ++ ys) = foldr (:) ys (x:xs) {FR1}
-- ∀ ys::[a] . x : (xs ++ ys) = (:) x (foldr (:) yz xs) {++}
-- ∀ ys::[a] . x : (xs ++ ys) = (:) x (xs ++ ys) {:}
-- ∀ ys::[a] . x : xs ++ ys = x : xs ++ ys {queda demostrada la igualdad}

-- iii. map id = id

-- Predicado unario: P(xs) = map id xs = id xs

-- Caso base: P([]) =

-- map id [] = id [] {M0}
-- [] = id [] {ID}
-- [] = [] {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = map id xs = id xs
-- Paso inductivo: P(x:xs) = map id (x:xs) = id (x:xs)

-- map id (x:xs) = id (x:xs) {M0}
-- foldr ((:) . id) [] (x:xs) = id (x:xs) {FR1}
-- (:) (id x) (foldr ((:) . id) [] xs) = id (x:xs) {ID}
-- (:) x (foldr ((:) . id) [] xs) = id (x:xs) {:}
-- x : (foldr ((:) . id) [] xs) = id (x:xs) {M0}
-- x : (map id xs) = id (x:xs) {HI}
-- x : (id xs) = id (x:xs) {ID}
-- x:xs = x:xs {queda demostrada la igualdad}

-- iv. ∀ f::a->b . ∀ g::b->c . map (g . f) = map g . map f

-- Predicado unario: P(xs) = ∀ f::a->b . ∀ g::b->c . map (g . f) xs = (map g . map f) xs

-- Caso base: P([]) =

-- ∀ f::a->b . ∀ g::b->c . map (g . f) [] = (map g . map f) [] {MA0}
-- ∀ f::a->b . ∀ g::b->c . [] = (map g . map f) [] {.}
-- [] = map g (map f []) {MA0}
-- [] = map g [] {MA0}
-- [] = [] {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ f::a->b . ∀ g::b->c . map (g . f) xs = (map g . map f) xs
-- Paso inductivo: P(x:xs) = ∀ f::a->b . ∀ g::b->c . map (g . f) (x:xs) = (map g . map f) (x:xs)

-- ∀ f::a->b . ∀ g::b->c . map (g . f) (x:xs) = (map g . map f) (x:xs) {MA1}
-- ∀ f::a->b . ∀ g::b->c . (g . f) x : map (g . f) xs = (map g . map f) (x:xs) {HI}
-- ∀ f::a->b . ∀ g::b->c . (g . f) x : ((map g . map f) xs) = (map g . map f) (x:xs) {.}
-- ∀ f::a->b . ∀ g::b->c . (g . f) x : (map g (map f xs)) = (map g . map f) (x:xs) {.}
-- ∀ f::a->b . ∀ g::b->c . (g . f) x : (map g (map f xs)) = (map g (map f x:xs)) {MA1}
-- ∀ f::a->b . ∀ g::b->c . (g . f) x : (map g (map f xs)) = (map g (f x : map f xs)) {MA1}
-- ∀ f::a->b . ∀ g::b->c . (g . f) x : (map g (map f xs)) = g (f x) : (map g (map f xs)) {.}
-- ∀ f::a->b . ∀ g::b->c . (g . f) x : (map g (map f xs)) = (g . f) x : (map g (map f xs)) {queda demostrada la igualdad}

-- v. ∀ f::a->b . ∀ p::b->Bool . filter (p . f) = filter p . map f

-- Predicado unario: P(xs) = ∀ f::a->b . ∀ p::b->Bool . filter (p . f) xs = (filter p . map f) xs

-- Caso base: P([]) =

-- ∀ f::a->b . ∀ p::b->Bool . filter (p . f) [] = (filter p . map f) [] {F0}
-- ∀ f::a->b . ∀ p::b->Bool . foldr (\x xs -> if p (f x) then x : xs else xs) [] [] = (filter p . map f) [] {FR0}
-- ∀ f::a->b . ∀ p::b->Bool . [] = (filter p . map f) [] {.}
-- ∀ f::a->b . ∀ p::b->Bool . [] = filter p (map f []) {MA0}

-- [] = filter p [] {F0}
-- [] = foldr (\x xs -> if p x then x : xs else xs) [] [] {FR0}
-- [] = [] {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ f::a->b . ∀ p::b->Bool . filter (p . f) xs = (filter p . map f) xs
-- Paso inductivo: P(x:xs) = ∀ f::a->b . ∀ p::b->Bool . filter (p . f) (x:xs) = (filter p . map f) (x:xs)

-- ∀ f::a->b . ∀ p::b->Bool . filter (p . f) (x:xs) = (filter p . map f) (x:xs) {F0}
-- ∀ f::a->b . ∀ p::b->Bool . foldr (\y ys -> if (p . f) y then y : ys else ys) [] (x:xs) = (filter p . map f) (x:xs) {FR1}

-- La función p aplicada a f x puede devolver True o False, por lo que se deben considerar ambos casos:

-- Caso 1: (p . f) x = True

-- ∀ f::a->b . ∀ p::b->Bool . f x : (filter (p . f) xs) = (filter p . map f) (x:xs) {HI}
-- ∀ f::a->b . ∀ p::b->Bool . f x : (filter p . map f xs) = (filter p . map f) (x:xs) {.}
-- ∀ f::a->b . ∀ p::b->Bool . f x : (filter p . map f xs) = filter p (map f (x:xs)) {MA1}
-- ∀ f::a->b . ∀ p::b->Bool . f x : (filter p . map f xs) = filter p (f x : map f xs) {F0}
-- ∀ f::a->b . ∀ p::b->Bool . f x : (filter p . map f xs) = foldr (\y ys -> if p y then y : ys else ys) [] (f x : map f xs) {CASO 1}
-- ∀ f::a->b . ∀ p::b->Bool . f x : (filter p . map f xs) = f x : foldr (\y ys -> if p y then y : ys else ys) [] (map f xs) {MA1}
-- ∀ f::a->b . ∀ p::b->Bool . f x : (filter p . map f xs) = f x : (filter p (map f xs)) {.}
-- ∀ f::a->b . ∀ p::b->Bool . f x : (filter p . map f xs) = f x : (filter p . map f xs) {queda demostrada la igualdad}

-- Caso 2: (p . f) x = False

-- ∀ f::a->b . ∀ p::b->Bool . filter (p . f) xs = (filter p . map f) (x:xs) {HI}
-- ∀ f::a->b . ∀ p::b->Bool . (filter p . map f) xs = (filter p . map f) (x:xs) {.}
-- ∀ f::a->b . ∀ p::b->Bool . (filter p . map f) xs = filter p (map f (x:xs)) {MA1}
-- ∀ f::a->b . ∀ p::b->Bool . (filter p . map f) xs = filter p (f x : map f xs) {F0}
-- ∀ f::a->b . ∀ p::b->Bool . (filter p . map f) xs = foldr (\y ys -> if p y then y : ys else ys) [] (f x : map f xs) {CASO 2}
-- ∀ f::a->b . ∀ p::b->Bool . (filter p . map f) xs = foldr (\y ys -> if p y then y : ys else ys) [] (map f xs) {MA1}
-- ∀ f::a->b . ∀ p::b->Bool . (filter p . map f) xs = filter p (map f xs) {.}
-- ∀ f::a->b . ∀ p::b->Bool . (filter p . map f) xs = (filter p . map f) xs {queda demostrada la igualdad}

-- vi. ∀ f::a->b . ∀ e::a . ∀ xs::[xs] . (elem e xs = True) ⇒ (elem (f e) (map f xs) = True) (asumiendo Eq a y Eq b)

-- Predicado unario: P(xs) = ∀ f::a->b . ∀ e::a . (elem e xs = True) ⇒ (elem (f e) (map f xs) = True)

-- Caso base: P([]) =

-- ∀ f::a->b . ∀ e::a . (elem e [] = True) ⇒ (elem (f e) (map f []) = True) {E0}
-- ∀ f::a->b . ∀ e::a . (foldr (\y b -> b || y == e) False [] = True) ⇒ (elem (f e) (map f []) = True) {FR0}
-- ∀ f::a->b . ∀ e::a . (False = True) ⇒ (elem (f e) (map f []) = True) {lógica}
-- ∀ f::a->b . ∀ e::a . False ⇒ (elem (f e) (map f []) = True) {queda demostrada la implicación}

-- Hipótesis inductiva: P(xs) = ∀ f::a->b . ∀ e::a . (elem e xs = True) ⇒ (elem (f e) (map f xs) = True)
-- Paso inductivo: P(x:xs) = ∀ f::a->b . ∀ e::a . (elem e (x:xs) = True) ⇒ (elem (f e) (map f (x:xs)) = True)

-- ∀ f::a->b . ∀ e::a . (elem e (x:xs) = True) ⇒ (elem (f e) (map f (x:xs)) = True) {E0}
-- ∀ f::a->b . ∀ e::a . (foldr (\y b -> b || y == e) False (x:xs) = True) ⇒ (elem (f e) (map f (x:xs)) = True) {FR1}
-- ∀ f::a->b . ∀ e::a . (((foldr (\y b -> b || y == e) False xs) || e == x) = True) ⇒ (elem (f e) (map f (x:xs)) = True) {FR1}
-- ∀ f::a->b . ∀ e::a . ((elem e xs || e == x) = True) ⇒ (elem (f e) (map f (x:xs)) = True) {partimos en casos}

-- Partimos en dos casos: o bien e == x o en caso contrario e /= x

-- Caso 1: e == x

-- ∀ f::a->b . ∀ e::a . ((elem e xs || e == x) = True) ⇒ (elem (f e) (map f (x:xs)) = True) {CASO 1}
-- ∀ f::a->b . ∀ e::a . (True = True) ⇒ (elem (f e) (map f (x:xs)) = True) {lógica}
-- ∀ f::a->b . ∀ e::a . True ⇒ (elem (f e) (map f (x:xs)) = True) {MA1}
-- ∀ f::a->b . ∀ e::a . True ⇒ (elem (f e) (f x : map f xs) = True) {E0}
-- ∀ f::a->b . ∀ e::a . True ⇒ ((foldr (\y b -> b || y == (f e)) False (f x : map f xs) || f x == f e) = True) {FR1}
-- ∀ f::a->b . ∀ e::a . True ⇒ ((foldr (\y b -> b || y == (f e)) False (map f xs) || f x == f e) = True) {MA1}
-- ∀ f::a->b . ∀ e::a . True ⇒ ((elem (f e) (map f xs) || f x == f e) = True) {CASO 1}
-- ∀ f::a->b . ∀ e::a . True ⇒ ((elem (f e) (map f xs) || True) = True) {lógica}
-- ∀ f::a->b . ∀ e::a . True ⇒ (True = True) {lógica}
-- ∀ f::a->b . ∀ e::a . True ⇒ True {queda demostrada la implicación}

-- Caso 2: e /= x

-- ∀ f::a->b . ∀ e::a . ((elem e xs || e == x) = True) ⇒ (elem (f e) (map f (x:xs)) = True) {CASO 2}
-- ∀ f::a->b . ∀ e::a . ((elem e xs || False) = True) ⇒ (elem (f e) (map f (x:xs)) = True) {lógica}
-- ∀ f::a->b . ∀ e::a . (elem e xs = True) ⇒ (elem (f e) (map f (x:xs)) = True) {HI}
-- ∀ f::a->b . ∀ e::a . (elem e xs = True) ⇒ (elem (f e) (map f xs) = True) ⇒ (elem (f e) (map f (x:xs)) = True) {queda demostrada la implicación}

-- Es decir, por hipótesis inductiva queda demostrada la implicación para cualquier e en xs. Luego se puede afirmar
-- que si elem (f e) (map f xs) = True para cualquier e en xs, entonces elem (f e) (map f (x:xs)) = True, que es
-- la misma lista pero con un elemento más

-- Ejercicio 5

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

_zip' :: [a] -> [b] -> [(a,b)]
_zip' [] _ = [] --{Z'0}
_zip' (x:xs) ys = if null ys then [] else (x, head ys) : _zip' xs (tail ys) --{Z'1}

-- _zip = _zip'

-- Predicado unario: P(xs) = ∀ ys::[b] . _zip xs ys = _zip' xs ys

-- Caso base: P([]) =

-- ∀ ys::[b] . _zip [] ys = _zip' [] ys {Z0}
-- ∀ ys::[b] . foldr (\x rec ys -> if null ys then [] else (x, head ys) : rec (tail ys)) (const []) [] ys = _zip' [] ys {FR0}
-- ∀ ys::[b] . const [] ys = _zip' [] ys {Z'0}
-- ∀ ys::[b] . const [] ys = [] {C0}
-- [] = [] {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ ys::[b] . _zip xs ys = _zip' xs ys
-- Paso inductivo: P(x:xs) = ∀ ys::[b] . _zip (x:xs) ys = _zip' (x:xs) ys

-- ∀ ys::[b] . _zip (x:xs) ys = _zip' (x:xs) ys {Z0}
-- ∀ ys::[b] . foldr (\x rec ys -> if null ys then [] else (x, head ys) : rec (tail ys)) (const []) (x:xs) ys = _zip' (x:xs) ys {FR1}
-- ∀ ys::[b] . (if null ys then [] else (x, head ys) : foldr (\x rec ys -> if null ys then [] else (x, head ys) : rec (tail ys)) (const []) xs (tail ys)) = _zip' (x:xs) ys {Z0}
-- ∀ ys::[b] . (if null ys then [] else (x, head ys) : _zip xs (tail ys)) = _zip' (x:xs) ys {HI}
-- ∀ ys::[b] . (if null ys then [] else (x, head ys) : _zip' xs (tail ys)) = _zip' (x:xs) ys {Z'1}
-- ∀ ys::[b] . (if null ys then [] else (x, head ys) : _zip' xs (tail ys)) = (if null ys then [] else (x, head ys) : _zip' xs (tail ys)) {queda demostrada la igualdad}

-- Ejercicio 6

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

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

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

-- i. Eq a => ∀ xs::[a] . ∀ e::a . elem e xs = elem e (nub xs)

-- Predicado unario: P(xs) = ∀ e::a . elem e xs = elem e (nub xs)

-- Caso base: P([]) =

-- ∀ e::a . elem e [] = elem e (nub []) {N0}
-- ∀ e::a . elem e [] = elem e [] {E0}
-- ∀ e::a . foldr (\y b -> b || y == e) False [] = foldr (\y b -> b || y == e) False [] {FR0}
-- ∀ e::a . False = False {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ e::a . elem e xs = elem e (nub xs)
-- Paso inductivo: P(x:xs) = ∀ e::a . elem e (x:xs) = elem e (nub (x:xs))

-- ∀ e::a . elem e (x:xs) = elem e (nub (x:xs)) {E0}
-- ∀ e::a . foldr (\y b -> b || y == e) False (x:xs) = elem e (nub (x:xs)) {FR1}
-- ∀ e::a . (foldr (\y b -> b || y == e) False xs || x == e) = elem e (nub (x:xs)) {E0}
-- ∀ e::a . (elem e xs || x == e) = elem e (nub (x:xs)) {HI}
-- ∀ e::a . (elem e (nub xs) || x == e) = elem e (nub (x:xs)) {partimos en casos}

-- Caso 1: x == e

-- ∀ e::a . (elem e (nub xs) || x == e) = elem e (nub (x:xs)) {CASO 1}
-- ∀ e::a . (elem e (nub xs) || True) = elem e (nub (x:xs)) {lógica}
-- ∀ e::a . True = elem e (nub (x:xs)) {N1}
-- ∀ e::a . True = elem e (x: nub (filter (\y -> x /= y) xs)) {E0}
-- ∀ e::a . True = foldr (\y b -> b || y == e) False (x: nub (filter (\y -> x /= y) xs)) {FR1}
-- ∀ e::a . True = (foldr (\y b -> b || y == e) False (nub (filter (\y -> x /= y) xs) || x == e) {CASO 1}
-- ∀ e::a . True = (foldr (\y b -> b || y == e) False (nub (filter (\y -> x /= y) xs) || True) {lógica}
-- ∀ e::a . True = True {queda demostrada la igualdad}

-- Caso 2: x /= e

-- ∀ e::a . (elem e (nub xs) || x == e) = elem e (nub (x:xs)) {CASO 2}

-- ∀ e::a . (elem e (nub xs) || False) = elem e (nub (x:xs)) {lógica}
-- ∀ e::a . elem e (nub xs) = elem e (nub (x:xs)) {N1}
-- ∀ e::a . elem e (nub xs) = elem e (x: nub (filter (\y -> x /= y) xs)) {E0}
-- ∀ e::a . elem e (nub xs) = foldr (\y b -> b || y == e) False (x: nub (filter (\y -> x /= y) xs)) {FR1}
-- ∀ e::a . elem e (nub xs) = (foldr (\y b -> b || y == e) False (nub (filter (\y -> x /= y) xs)) || x == e) {CASO 2}
-- ∀ e::a . elem e (nub xs) = (foldr (\y b -> b || y == e) False (nub (filter (\y -> x /= y) xs)) || False) {lógica}
-- ∀ e::a . elem e (nub xs) = foldr (\y b -> b || y == e) False (nub (filter (\y -> x /= y) xs)) {E0}
-- ∀ e::a . elem e (nub xs) = elem e (nub (filter (\y -> x /= y) xs) {N1???}
-- ∀ e::a . elem e (nub xs) = elem e (nub xs) {queda demostrada la igualdad}

-- ii. Eq a => ∀ xs::[a] . ∀ ys::[a] . ∀ e::a . elem e (union xs ys) = (elem e xs) || (elem e ys)

-- Predicado unario: P(xs) = ∀ ys::[a] . ∀ e::a . elem e (union xs ys) = (elem e xs) || (elem e ys)

-- Caso base: P([]) =

-- ∀ ys::[a] . ∀ e::a . elem e (union [] ys) = (elem e []) || (elem e ys) {U0}
-- ∀ ys::[a] . ∀ e::a . elem e (nub ([] ++ ys)) = (elem e []) || (elem e ys) {++AUX2}
-- ∀ ys::[a] . ∀ e::a . elem e (nub ys) = (elem e []) || (elem e ys) {E0}
-- ∀ ys::[a] . ∀ e::a . elem e (nub ys) = (foldr (\y b -> b || y == e) False []) || (elem e ys) {FR0}
-- ∀ ys::[a] . ∀ e::a . elem e (nub ys) = False || (elem e ys) {lógica}
-- ∀ ys::[a] . ∀ e::a . elem e (nub ys) = elem e ys {i}
-- ∀ ys::[a] . ∀ e::a . True {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ ys::[a] . ∀ e::a . elem e (union xs ys) = (elem e xs) || (elem e ys)
-- Paso inductivo: P(x:xs) = ∀ ys::[a] . ∀ e::a . elem e (union (x:xs) ys) = (elem e (x:xs)) || (elem e ys)

-- ∀ ys::[a] . ∀ e::a . elem e (union (x:xs) ys) = (elem e (x:xs)) || (elem e ys) {U0}
-- ∀ ys::[a] . ∀ e::a . elem e (nub ((x:xs) ++ ys)) = (elem e (x:xs)) || (elem e ys) {++}
-- ∀ ys::[a] . ∀ e::a . elem e (nub (foldr (:) ys (x:xs))) = (elem e (x:xs)) || (elem e ys) {FR1}
-- ∀ ys::[a] . ∀ e::a . elem e (nub ((:) x (foldr (:) ys xs)) = (elem e (x:xs)) || (elem e ys) {:}
-- ∀ ys::[a] . ∀ e::a . elem e (nub (x : foldr (:) ys xs)) = (elem e (x:xs)) || (elem e ys) {++}
-- ∀ ys::[a] . ∀ e::a . elem e (nub (x:(xs ++ ys))) = (elem e (x:xs)) || (elem e ys) {N1}
-- ∀ ys::[a] . ∀ e::a . elem e (x : nub (filter (\y -> x /= y) (xs ++ ys))) = (elem e (x:xs)) || (elem e ys) {E0}
-- ∀ ys::[a] . ∀ e::a . foldr (\y b -> b || y == e) False (x : nub (filter (\y -> x /= y) (xs ++ ys))) = (elem e (x:xs)) || (elem e ys) {FR1}
-- ∀ ys::[a] . ∀ e::a . (foldr (\y b -> b || y == e) False (nub (filter (\y -> x /= y) (xs ++ ys)) || x == e) = (elem e (x:xs)) || (elem e ys) {E0}
-- ∀ ys::[a] . ∀ e::a . elem e (nub (filter (\y -> x /= y) (xs ++ ys)) || x == e = (elem e (x:xs)) || (elem e ys) {N1???}
-- ∀ ys::[a] . ∀ e::a . elem e (nub (xs ++ ys)) || x == e = (elem e (x:xs)) || (elem e ys) {U0}
-- ∀ ys::[a] . ∀ e::a . elem e (union xs ys) || x == e = (elem e (x:xs)) || (elem e ys) {HI}
-- ∀ ys::[a] . ∀ e::a . elem e xs || elem e ys || x == e = (elem e (x:xs)) || (elem e ys) {E0}
-- ∀ ys::[a] . ∀ e::a . elem e xs || elem e ys || x == e = (foldr (\y b -> b || y == e) False (x:xs)) || elem e ys) {FR1}
-- ∀ ys::[a] . ∀ e::a . elem e xs || elem e ys || x == e = (foldr (\y b -> b || y == e) False xs || x == e) || elem e ys) {E0}
-- ∀ ys::[a] . ∀ e::a . elem e xs || elem e ys || x == e = elem e xs || x == e || elem e ys {queda demostrada la igualdad}

-- iii. Eq a => ∀ xs::[a] . ∀ ys::[a] . ∀ e::a . elem e (intersect xs ys) = (elem e xs) && (elem e ys)

-- Predicado unario: P(xs) = ∀ ys::[a] . ∀ e::a . elem e (intersect xs ys) = (elem e xs) && (elem e ys)

-- Caso base: P([]) =

-- ∀ ys::[a] . ∀ e::a . elem e (intersect [] ys) = (elem e []) && (elem e ys) {I0}
-- ∀ ys::[a] . ∀ e::a . elem e (filter (\e -> elem e ys) []) = (elem e []) && (elem e ys) {F0}
-- ∀ ys::[a] . ∀ e::a . elem e (foldr (\e b -> if elem e ys then e : b else b) [] []) = (elem e []) && (elem e ys) {FR0}
-- ∀ ys::[a] . ∀ e::a . elem e [] = (elem e []) && (elem e ys) {E0}
-- ∀ ys::[a] . ∀ e::a . foldr (\y b -> b || y == e) False [] = (elem e []) && (elem e ys) {FR0}
-- ∀ ys::[a] . ∀ e::a . False = (elem e []) && (elem e ys) {E0}
-- ∀ ys::[a] . ∀ e::a . False = (foldr (\y b -> b || y == e) False [] && elem e ys) {FR0}
-- ∀ ys::[a] . ∀ e::a . False = False && elem e ys {lógica}
-- ∀ ys::[a] . ∀ e::a . False = False {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ ys::[a] . ∀ e::a . elem e (intersect xs ys) = (elem e xs) && (elem e ys)
-- Paso inductivo: P(x:xs) = ∀ ys::[a] . ∀ e::a . elem e (intersect (x:xs) ys) = (elem e (x:xs)) && (elem e ys)

-- ∀ ys::[a] . ∀ e::a . elem e (intersect (x:xs) ys) = (elem e (x:xs)) && (elem e ys) {I0}
-- ∀ ys::[a] . ∀ e::a . elem e (filter (\y -> elem y ys) (x:xs)) = (elem e (x:xs)) && (elem e ys) {F0}
-- ∀ ys::[a] . ∀ e::a . elem e (foldr (\y b -> if elem y ys then y : b else b) [] (x:xs)) = (elem e (x:xs)) && (elem e ys) {partimos en casos}

-- Partimos en dos casos: o bien elem x ys = True o en caso contrario elem x ys = False

-- Caso 1: elem x ys = True

-- ∀ ys::[a] . ∀ e::a . elem e (foldr (\y b -> if elem y ys then y : b else b) [] (x:xs)) = (elem e (x:xs)) && (elem e ys) {CASO 1}
-- ∀ ys::[a] . ∀ e::a . elem e (x : foldr (\y b -> if elem y ys then y : b else b) [] (xs)) = (elem e (x:xs)) && (elem e ys) {F0}
-- ∀ ys::[a] . ∀ e::a . elem e (x : filter (\y -> elem y ys) xs) = (elem e (x:xs)) && (elem e ys) {I0}
-- ∀ ys::[a] . ∀ e::a . elem e (x : intersect xs ys) = (elem e (x:xs)) && (elem e ys) {E0}
-- ∀ ys::[a] . ∀ e::a . foldr (\y b -> b || y == e) False (x : intersect xs ys) = (elem e (x:xs)) && (elem e ys) {FR1}
-- ∀ ys::[a] . ∀ e::a . (foldr (\y b -> b || y == e) False (intersect xs ys)) || x == e = (elem e (x:xs)) && (elem e ys) {E0}
-- ∀ ys::[a] . ∀ e::a . elem e (intersect xs ys) || x == e = (elem e (x:xs)) && (elem e ys) {HI}
-- ∀ ys::[a] . ∀ e::a . elem e xs && elem e ys || x == e = (elem e (x:xs)) && (elem e ys) {lógica}
-- ∀ ys::[a] . ∀ e::a . elem e ys && elem e xs || x == e = (elem e (x:xs)) && (elem e ys) {E0}
-- ∀ ys::[a] . ∀ e::a . elem e ys && elem e xs || x == e = (foldr (\y b -> b || y == e) False (x:xs)) && elem e ys) {FR1}
-- ∀ ys::[a] . ∀ e::a . elem e ys && elem e xs || x == e = ((foldr (\y b -> b || y == e) False xs) || x == e) && elem e ys) {E0}
-- ∀ ys::[a] . ∀ e::a . elem e ys && elem e xs || x == e = (elem e xs || x == e) && elem e ys {???}

-- iv. Eq a => ∀ xs::[a] . ∀ ys::[a] . length (union xs ys) = length xs + length ys

-- Es falso, ya que la longitud de la unión de dos listas no necesariamente es la suma de las longitudes de las listas.
-- Por ejemplo, si xs = [1,2,3] e ys = [3,4,5], entonces la longitud de la unión de xs e ys es 5, mientras que
-- la suma de las longitudes de xs e ys es 6.

-- v. Eq a => ∀ xs::[a] . ∀ ys::[a] . length (union xs ys) ≤ length xs + length ys

-- Predicado unario: P(xs) = ∀ ys::[a] . length (union xs ys) ≤ length xs + length ys

-- Caso base: P([]) =

-- ∀ ys::[a] . length (union [] ys) ≤ length [] + length ys {U0}
-- ∀ ys::[a] . length (nub ([] ++ ys)) ≤ length [] + length ys {++AUX2}
-- ∀ ys::[a] . length (nub ys) ≤ length [] + length ys {L0}
-- ∀ ys::[a] . length (nub ys) ≤ length 0 + length ys {aritmética}
-- ∀ ys::[a] . length (nub ys) ≤ length ys {queda demostrada la desigualdad por LEMA}

-- Hipótesis inductiva: P(xs) = ∀ ys::[a] . length (union xs ys) ≤ length xs + length ys
-- Paso inductivo: P(x:xs) = ∀ ys::[a] . length (union (x:xs) ys) ≤ length (x:xs) + length ys

-- ∀ ys::[a] . length (union (x:xs) ys) ≤ length (x:xs) + length ys {U0}
-- ∀ ys::[a] . length (nub ((x:xs) ++ ys)) ≤ length (x:xs) + length ys {++}
-- ∀ ys::[a] . length (nub (foldr (:) ys (x:xs))) ≤ length (x:xs) + length ys {FR1}
-- ∀ ys::[a] . length (nub ((:) x (foldr (:) ys xs)) ≤ length (x:xs) + length ys {:}
-- ∀ ys::[a] . length (nub (x : foldr (:) ys xs)) ≤ length (x:xs) + length ys {++}
-- ∀ ys::[a] . length (nub (x:(xs ++ ys))) ≤ length (x:xs) + length ys {N1}
-- ∀ ys::[a] . length (x : nub (filter (\y -> x /= y) (xs ++ ys))) ≤ length (x:xs) + length ys {L0}
-- ∀ ys::[a] . 1 + length (nub (filter (\y -> x /= y) (xs ++ ys))) ≤ 1 + length xs + length ys {aritmética}
-- ∀ ys::[a] . length (nub (filter (\y -> x /= y) (xs ++ ys))) ≤ length xs + length ys {N1}
-- ∀ ys::[a] . length (nub (xs ++ ys)) ≤ length xs + length ys {U0}
-- ∀ ys::[a] . length (union xs ys) ≤ length xs + length ys {queda demostrada la desigualdad por HI}

-- Lema auxiliar: ∀ xs::[a] . length (nub xs) ≤ length xs

-- Predicado unario: P(xs) = length (nub xs) ≤ length xs

-- Caso base: P([]) =

-- length (nub []) ≤ length [] {N0}
-- length [] ≤ length [] {L0}
-- 0 ≤ 0 {queda demostrada la desigualdad}

-- Hipótesis inductiva: P(xs) = length (nub xs) ≤ length xs
-- Paso inductivo: P(x:xs) = length (nub (x:xs)) ≤ length (x:xs)

-- length (nub (x:xs)) ≤ length (x:xs) {N1}
-- length (x : nub (filter (\y -> x /= y) xs)) ≤ length (x:xs) {L1}
-- 1 + length (nub (filter (\y -> x /= y) xs) ≤ length (x:xs) {L1}
-- 1 + length (nub xs) ≤ 1 + length xs {aritmética}
-- length (nub xs) ≤ length xs {queda demostrada la desigualdad por HI}

-- Ejercicio 7

-- i. ∀ f::a->b->b . ∀ z::b . ∀ xs, ys::[a] . foldr f z (xs ++ ys) = foldr f (foldr f z ys) xs

-- Predicado unario: P(xs) = ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . foldr f z (xs ++ ys) = foldr f (foldr f z ys) xs

-- Caso base: P([]) =

-- ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . foldr f z ([] ++ ys) = foldr f (foldr f z ys) [] {++AUX2}
-- ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . foldr f z ys = foldr f (foldr f z ys) [] {FR0}
-- ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . foldr f z ys = foldr f z ys {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . foldr f z (xs ++ ys) = foldr f (foldr f z ys) xs
-- Paso inductivo: P(x:xs) = ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . foldr f z ((x:xs) ++ ys) = foldr f (foldr f z ys) (x:xs)

-- ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . foldr f z ((x:xs) ++ ys) = foldr f (foldr f z ys) (x:xs) {++}
-- ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . foldr f z (foldr (:) ys x:xs) = foldr f (foldr f z ys) (x:xs) {FR1}
-- ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . foldr f z ((:) x foldr (:) ys xs) = foldr f (foldr f z ys) (x:xs) {:}
-- ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . foldr f z (x : foldr (:) ys xs) = foldr f (foldr f z ys) (x:xs) {++}

-- ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . foldr f z (x : xs ++ ys) = foldr f (foldr f z ys) (x:xs) {FR1}
-- ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . f x (foldr f z (xs ++ ys)) = foldr f (foldr f z ys) (x:xs) {HI}
-- ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . f x (foldr f (foldr f z ys) xs) = foldr f (foldr f z ys) (x:xs) {FR1}
-- ∀ f::a->b->b . ∀ z::b . ∀ ys::[a] . f x (foldr f (foldr f z ys) xs) = f x (foldr f (foldr f z ys) xs) {queda demostrada la igualdad}

-- ii. ∀ f::b->a->b . ∀ z::b . ∀ xs, ys::[a] . foldl f z (xs ++ ys) = foldl f (foldl f z xs) ys

-- Predicado unario: P(xs) = ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f z (xs ++ ys) = foldl f (foldl f z xs) ys

-- Caso base: P([]) =

-- ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f z ([] ++ ys) = foldl f (foldl f z []) ys {++AUX2}
-- ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f z ys = foldl f (foldl f z []) ys {FL0}
-- ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f z ys = foldl f z ys {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f z (xs ++ ys) = foldl f (foldl f z xs) ys
-- Paso inductivo: P(x:xs) = ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f z ((x:xs) ++ ys) = foldl f (foldl f z (x:xs)) ys

-- ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f z ((x:xs) ++ ys) = foldl f (foldl f z (x:xs)) ys {++}
-- ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f z (foldr (:) ys x:xs) = foldl f (foldl f z (x:xs)) ys {FR1}
-- ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f z ((:) x foldr (:) ys xs) = foldl f (foldl f z (x:xs)) ys {:}
-- ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f z (x : foldr (:) ys xs) = foldl f (foldl f z (x:xs)) ys {++}
-- ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f z (x : xs ++ ys) = foldl f (foldl f z (x:xs)) ys {FL1}
-- ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f (f z x) (xs ++ ys) = foldl f (foldl f z (x:xs)) ys {HI}
-- ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f (foldl f (f z x) xs) ys = foldl f (foldl f z (x:xs)) ys {FL1}
-- ∀ f::b->a->b . ∀ z::b . ∀ ys::[a] . foldl f (foldl f (f z x) xs) ys = foldl f (foldl f (f z x) xs) ys {queda demostrada la igualdad}

-- Ejercicio 8

-- Demostrar que la función potencia funciona correctamente mediante inducción en el exponente

-- potencia :: Integer -> Integer -> Integer
-- potencia x y = foldNat (_ accu -> x * accu) 1 y {POT0}

-- foldNat :: (Integer -> b -> b) -> b -> Integer -> b
-- foldNat _ z 0 = z {FN0}
-- foldNat f z n = f n (foldNat f z (n - 1)) {FN1}

-- potencia xy = x^y

-- Predicado unario: P(y) = ∀ x::Int . potencia x y = x^y

-- Caso base: P(0) =

-- ∀ x::Int . potencia x 0 = x^0 {POT0}
-- ∀ x::Int . foldNat (_ accu -> x * accu) 1 0 = x^0 {FN0}
-- ∀ x::Int . 1 = x^0 {aritmética}
-- ∀ x::Int . 1 = 1 {queda demostrada la igualdad}

-- Hipótesis inductiva: P(y) = ∀ x::Int . potencia x y = x^y
-- Paso inductivo: P(y+1) = ∀ x::Int . potencia x (y+1) = x^(y+1)

-- ∀ x::Int . potencia x (y+1) = x^(y+1) {POT0}
-- ∀ x::Int . foldNat (_ accu -> x * accu) 1 (y+1) = x^(y+1) {FN1}
-- ∀ x::Int . x * foldNat (_ accu -> x * accu) 1 y = x^(y+1) {HI}
-- ∀ x::Int . x * x^y = x^(y+1) {aritmética}
-- ∀ x::Int . x^(y+1) = x^(y+1) {queda demostrada la igualdad}

-- Ejercicio 9

-- data AB a = Empty | Bin (AB a) a (AB a)

-- foldAB :: (b -> a -> b -> b) -> b -> AB a -> b
-- foldAB _ z Empty = z {FAB0}
-- foldAB f z (Bin izq root der) = f (foldAB f z izq) root (foldAB f z der) {FAB1}

-- altura :: AB a -> Int
-- altura = foldAB (\izq _ der -> 1 + max izq der) 0 {AAB0}

-- cantNodos :: AB a -> Int
-- cantNodos = foldAB (\izq _ der -> 1 + izq + der) 0 {CNAB0}

-- ∀ x::AB a . altura x ≤ cantNodos x

-- Predicado unario: P(x) = altura x ≤ cantNodos x

-- Caso base: P(Empty) =

-- altura Empty ≤ cantNodos Empty {AAB0}
-- foldAB (\izq _ der -> 1 + max izq der) 0 Empty ≤ cantNodos Empty {FAB0}
-- 0 ≤ cantNodos Empty {CNAB0}
-- 0 ≤ foldAB (\izq _ der -> 1 + izq + der) 0 Empty {FAB0}
-- 0 ≤ 0 {queda demostrada la desigualdad}

-- Hipótesis inductiva: P(izq) = altura izq ≤ cantNodos izq y P(der) = altura der ≤ cantNodos der
-- Paso inductivo: P(Bin izq root der) = altura (Bin izq root der) ≤ cantNodos (Bin izq root der)

-- altura (Bin izq root der) ≤ cantNodos (Bin izq root der) {AAB0}
-- foldAB (\izq _ der -> 1 + max izq der) 0 (Bin izq root der) ≤ cantNodos (Bin izq root der) {FAB1}
-- 1 + max (foldAB (\izq _ der -> 1 + max izq der) 0 izq) (foldAB (\izq _ der -> 1 + max izq der) 0 der) ≤ cantNodos (Bin izq root der) {AAB0}
-- 1 + max (altura izq) (altura der) ≤ cantNodos (Bin izq root der) {CNAB0}
-- 1 + max (altura izq) (altura der) ≤ foldAB (\izq _ der -> 1 + izq + der) 0 (Bin izq root der) {FAB1}
-- 1 + max (altura izq) (altura der) ≤ 1 + foldAB (\izq _ der -> 1 + izq + der) 0 izq + foldAB (\izq _ der -> 1 + izq + der) 0 der {CNAB0}
-- 1 + max (altura izq) (altura der) ≤ 1 + cantNodos izq + cantNodos der {aritmética}
-- max (altura izq) (altura der) ≤ cantNodos izq + cantNodos der {HI}
-- max (altura izq) (altura der) ≤ altura izq + altura der ≤ cantNodos izq + cantNodos der {queda demostrada la desigualdad}

-- Ejercicio 10

-- data AB a = Nil | Bin (AB a) a (AB a)

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

-- foldAB :: b -> (b -> a -> b -> b) -> AB a -> b
-- foldAB cNil cBin Nil = cNil {F0}
-- foldAB cNil cBin (Bin i r d) = cBin (rec i) r (rec d) where rec = foldAB cNil cBin {F1}

-- altura :: AB a -> Int
-- altura = foldAB 0 (\ri x rd -> 1 + max ri rd) {A0}

-- ∀ x::Int . ∀ y::Int . ∀ z::Int . max (min x y) (min x z) = min x (max y z) {LEMA_1}
-- ∀ x::Int . ∀ y::Int . ∀ z::Int . z + min x y = min (z+x) (z+y) {LEMA_2}

-- Demostrar las siguientes propiedades sobre árboles AB:

-- i) ∀ t::AB a . altura t ≥ 0

-- Predicado unario: P(t) = altura t ≥ 0

-- Caso base: P(Nil) = altura Nil ≥ 0

-- altura Nil ≥ 0 {A0}
-- foldAB 0 (\ri x rd -> 1 + max ri rd) Nil ≥ 0 {F0}
-- 0 ≥ 0 {queda demostrada la desigualdad}

-- Hipótesis inductiva: P(i) = altura i ≥ 0 y P(d) = altura d ≥ 0
-- Paso inductivo: P(Bin i r d) = altura (Bin i r d) ≥ 0

-- altura (Bin i r d) ≥ 0 {A0}
-- foldAB 0 (\ri x rd -> 1 + max ri rd) (Bin i r d) ≥ 0 {F1}
-- 1 + max (foldAB 0 (\ri x rd -> 1 + max ri rd) i) (foldAB 0 (\ri x rd -> 1 + max ri rd) d) ≥ 0 {A0}
-- 1 + max (altura i) (altura d) ≥ 0 {aritmética}

-- ii) ∀ t::AB a . ∀ n::Int . n ≥ 0 ⇒ (altura (truncar t n) = min n (altura t))

-- Predicado unario: P(t) = ∀ n::Int . n ≥ 0 ⇒ (altura (truncar t n) = min n (altura t))

-- Caso base: P(Nil) = ∀ n::Int . n ≥ 0 ⇒ (altura (truncar Nil n) = min n (altura Nil))

-- ∀ n::Int . n ≥ 0 ⇒ (altura (truncar Nil n) = min n (altura Nil)) {T0}
-- ∀ n::Int . n ≥ 0 ⇒ (altura (Nil) = min n (altura Nil)) {A0}
-- ∀ n::Int . n ≥ 0 ⇒ (0 = min n 0) {aritmética}
-- ∀ n::Int . n ≥ 0 ⇒ (0 = 0) {queda demostrada la implicación}

-- Hipótesis inductivas:
--    P(i) = ∀ n::Int . n ≥ 0 ⇒ (altura (truncar i n) = min n (altura i))
--    P(d) = ∀ n::Int . n ≥ 0 ⇒ (altura (truncar d n) = min n (altura d))

-- Paso inductivo: P(Bin i r d) = ∀ n::Int . n ≥ 0 ⇒ (altura (truncar (Bin i r d) n) = min n (altura (Bin i r d)))

-- Vamos a separar la demostración en tres partes: n = 0, n > 0 y n < 0

-- Caso n < 0:

-- Si n < 0 entonces n ≥ 0 es falso y la implicación es verdadera.

-- Caso n = 0:

-- altura (truncar (Bin i r d) 0) = min 0 (altura (Bin i r d)) {T1}
-- altura (if n == 0 then Nil else Bin (truncar i (n-1)) r (truncar d (n-1))) = min 0 (altura (Bin i r d)) {CASO n = 0}
-- altura (Nil) = min 0 (altura (Bin i r d)) {A0}
-- foldAB 0 (\ri x rd -> 1 + max ri rd) Nil = min 0 (altura (Bin i r d)) {F0}
-- 0 = min 0 (altura (Bin i r d)) {aritmética}

-- 0 = 0 {queda demostrada la igualdad}

-- Caso n > 0:

-- altura (truncar (Bin i r d) n) = min n (altura (Bin i r d)) {A0}
-- foldAB 0 (\ri x rd -> 1 + max ri rd) (truncar (Bin i r d) n) = min n (altura (Bin i r d)) {T1}
-- foldAB 0 (\ri x rd -> 1 + max ri rd) (if n == 0 then Nil else Bin (truncar i (n-1)) r (truncar d (n-1))) = min n (altura (Bin i r d)) {CASO n > 0}
-- foldAB 0 (\ri x rd -> 1 + max ri rd) (Bin (truncar i (n-1)) r (truncar d (n-1))) = min n (altura (Bin i r d)) {F1}
-- 1 + max (foldAB 0 (\ri x rd -> 1 + max ri rd) (truncar i (n-1))) (foldAB 0 (\ri x rd -> 1 + max ri rd) (truncar d (n-1))) = min n (altura (Bin i r d)) {A0}
-- 1 + max (altura (truncar i (n-1))) (altura (truncar d (n-1))) = min n (altura (Bin i r d)) {A0}
-- 1 + max (altura (truncar i (n-1))) (altura (truncar d (n-1))) = min n (foldAB 0 (\ri x rd -> 1 + max ri rd) (Bin i r d)) {F1}
-- 1 + max (altura (truncar i (n-1))) (altura (truncar d (n-1))) = min n (1 + max (foldAB 0 (\ri x rd -> 1 + max ri rd) i) (foldAB 0 (\ri x rd -> 1 + max ri rd) d)) {A0}
-- 1 + max (altura (truncar i (n-1))) (altura (truncar d (n-1))) = min n (1 + max (altura i) (altura d)) {LEMA_2}
-- 1 + max (altura (truncar i (n-1))) (altura (truncar d (n-1))) = 1 + min (n-1) (max (altura i) (altura d)) {aritmética}
-- max (altura (truncar i (n-1))) (altura (truncar d (n-1))) = min (n-1) (max (altura i) (altura d))

-- Como estamos en el caso n > 0 entonces vale n-1 ≥ 0 y podemos reescribir de la siguiente forma sin perder generalidad:

-- max (altura (truncar i n)) (altura (truncar d n)) = min n (max (altura i) (altura d)) {HI}
-- max (min n (altura i) min n (altura d)) = min n (max (altura i) (altura d)) {LEMA_1}
-- min n (max (altura i) (altura d)) = min n (max (altura i) (altura d)) {queda demostrada la igualdad y la implicación}

-- Ejercicio 11

-- inorder :: AB a -> [a]
-- inorder = foldAB [] (\ri x rd -> ri ++ (x:rd)) {IO0}

-- elemAB :: Eq a => a -> AB a -> Bool
-- elemAB e = foldAB False (\ri x rd -> (e == x) || ri || rd) {EAB0}

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

-- Demostrar Eq a => ∀ e::a . ∀ x::AB a . elemAB e x = elem e (inorder x)

-- Predicado unario: P(x) = ∀ e::a . elemAB e x = elem e (inorder x)

-- Caso base: P(Empty) =

-- ∀ e::a . elemAB e Empty = elem e . inorder Empty {EAB0}
-- ∀ e::a . foldAB False (\ri x rd -> (e == x) || ri || rd) Empty) = elem e. inorder Empty {FAB0}
-- ∀ e::a . False = elem e. inorder Empty {IO0}
-- ∀ e::a . False = elem e (foldAB [] (\ri x rd -> ri ++ (x:rd)) Empty) {FAB0}
-- ∀ e::a . False = elem e [] {E0}
-- ∀ e::a . False = foldr (\x rec -> (e == x) || rec) False [] {FR0}
-- ∀ e::a . False = False {queda demostrada la igualdad}

-- Hipótesis inductiva:
-- ∀ izq::AB a . P(izq) = ∀ e::a . elemAB e izq = elem e (inorder izq)
-- ∀ der::AB a . P(der) = ∀ e::a . elemAB e der = elem e (inorder der)

-- Paso inductivo: P(Bin izq root der) = ∀ e::a . elemAB e (Bin izq root der) = elem e (inorder (Bin izq root der))

-- ∀ e::a . elemAB e (Bin izq root der) = elem e (inorder (Bin izq root der)) {EAB0}
-- ∀ e::a . foldAB False (\ri x rd -> (e == x) || ri || rd) (Bin izq root der) = elem e (inorder (Bin izq root der)) {FAB1}
-- ∀ e::a . (foldAB False (\ri x rd -> (e == x) || ri || rd) izq || e == root || foldAB False (\ri x rd -> (e == x) || ri || rd) der) = elem e (inorder (Bin izq root der)) {EAB0}
-- ∀ e::a . elemAB e izq || e == root || elemAB e der = elem e (inorder (Bin izq root der)) {HI}
-- ∀ e::a . elem e (inorder izq) || e == root || elem e (inorder der) = elem e (inorder (Bin izq root der)) {IO0}
-- ∀ e::a . elem e (inorder izq) || e == root || elem e (inorder der) = elem e (foldAB [] (\ri x rd -> ri ++ (x:rd)) (Bin izq root der)) {FAB1}
-- ∀ e::a . elem e (inorder izq) || e == root || elem e (inorder der) = elem e (foldAB [] (\ri x rd -> ri ++ (x:rd)) izq ++ (root:foldAB [] (\ri x rd -> ri ++ (x:rd)) der) {IO0}
-- ∀ e::a . elem e (inorder izq) || e == root || elem e (inorder der) = elem e (inorder izq ++ (root:inorder der)) {E0}
-- ∀ e::a . elem e (inorder izq) || e == root || elem e (inorder der) = foldr (\x rec -> (e == x) || rec) False (inorder izq ++ (root:inorder der)) {Ejercicio 7}
-- ∀ e::a . elem e (inorder izq) || e == root || elem e (inorder der) = foldr (\x rec -> (e == x) || rec) (foldr (\x rec -> (e == x) || rec) False (root:inorder der)) (inorder izq) {FR1}
-- ∀ e::a . elem e (inorder izq) || e == root || elem e (inorder der) = foldr (\x rec -> (e == x) || rec) (e == root || (foldr (\x rec -> (e == x) || rec) False (inorder der))) (inorder izq) {E0}
-- ∀ e::a . elem e (inorder izq) || e == root || elem e (inorder der) = foldr (\x rec -> (e == x) || rec) (e == root || elem e (inorder der)) (inorder izq) {E0}
-- ∀ e::a . elem e (inorder izq) || e == root || elem e (inorder der) = e == root || elem e (inorder der) || foldr (\x rec -> (e == x) || rec) False (inorder izq) {E0}
-- ∀ e::a . elem e (inorder izq) || e == root || elem e (inorder der) = e == root || elem e (inorder der) || elem e (inorder izq) {queda demostrada la igualdad}

-- Ejercicio 12

-- data Polinomio a = X --CASO BASE
-- | Cte a -- CASO BASE
-- | Suma (Polinomio a) (Polinomio a) -- Caso Recursivo
-- | Prod (Polinomio a) (Polinomio a) -- Caso Recursivo

-- evaluar :: Num a => a -> Polinomio a -> a
-- evaluar x = foldPolinomio id x (+) (*) {EVAL0}

-- foldPolinomio :: (a -> b) -> b -> (b -> b -> b) -> (b -> b -> b) -> Polinomio a -> b
-- foldPolinomio _ z _ _ X = z {FPX0}
-- foldPolinomio f _ _ _ (Cte x) = f x {FPC0}
-- foldPolinomio f z s p (Suma x y) = s (foldPolinomio f z s p x) (foldPolinomio f z s p y) {FPS0}
-- foldPolinomio f z s p (Prod x y) = p (foldPolinomio f z s p x) (foldPolinomio f z s p y) {FPP0}

-- 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) (&&) (&&) {SCN0}

-- esRaiz :: Num a => a -> Polinomio a -> Bool
-- esRaiz n p = evaluar n p == 0 {ER0}

-- i. Num a => ∀ p::Polinomio a . ∀ q::Polinomio a . ∀ r::a . esRaiz r p ⇒ esRaiz r (Prod p q)

-- Predicado unario: P(p) = ∀ r::a . esRaiz r p ⇒ esRaiz r (Prod p q)

-- Caso base: P(X) = ∀ r::a . esRaiz r X ⇒ esRaiz r (Prod X q)

-- ∀ r::a . esRaiz r X ⇒ esRaiz r (Prod X q) {ER0}
-- ∀ r::a . evaluar r X == 0 ⇒ esRaiz r (Prod X q) {EVAL0}
-- ∀ r::a . foldPolinomio id r (+) (*) X == 0 ⇒ esRaiz r (Prod X q) {FPX0}
-- ∀ r::a . r == 0 ⇒ esRaiz r (Prod X q) {ER0}
-- ∀ r::a . r == 0 ⇒ evaluar r (Prod X q) == 0 {EVAL0}
-- ∀ r::a . r == 0 ⇒ (foldPolinomio id r (+) (*) (Prod X q)) == 0 {FPP0}
-- ∀ r::a . r == 0 ⇒ (foldPolinomio id r (+) (*) X) * (foldPolinomio id r (+) (*) q) == 0 {FPX0}
-- ∀ r::a . r == 0 ⇒ r * (foldPolinomio id r (+) (*) q) == 0

-- O bien r == 0 o bien r != 0

-- Caso 1: r == 0

-- r == 0 ⇒ 0 * (foldPolinomio id r (+) (*) q) == 0 {remplazamos r por 0}
-- 0 == 0 ⇒ 0 * (foldPolinomio id 0 (+) (*) q) == 0 {aritmética}
-- True ⇒ True {queda demostrada la implicación}

-- Caso 2: r != 0

-- r == 0 ⇒ 0 * (foldPolinomio id r (+) (*) q) == 0 {CASO 2}
-- False ⇒ 0 * (foldPolinomio id r (+) (*) q) == 0 {queda demostrada la implicación}

-- Caso Base: P(Cte x) = ∀ r::a . esRaiz r (Cte x) ⇒ esRaiz r (Prod (Cte x) q)

-- ∀ r::a . esRaiz r (Cte x) ⇒ esRaiz r (Prod (Cte x) q {ER0}
-- ∀ r::a . evaluar r (Cte x) == 0 ⇒ esRaiz r (Prod (Cte x) q) {EVAL0}
-- ∀ r::a . foldPolinomio id r (+) (*) (Cte x) == 0 ⇒ esRaiz r (Prod (Cte x) q) {FPC0}
-- ∀ r::a . id x == 0 ⇒ esRaiz r (Prod (Cte x) q) {ID0}
-- ∀ r::a . x == 0 ⇒ esRaiz r (Prod (Cte x) q) {ER0}
-- ∀ r::a . x == 0 ⇒ evaluar r (Prod (Cte x) q) == 0 {EVAL0}
-- ∀ r::a . x == 0 ⇒ (foldPolinomio id r (+) (*) (Prod (Cte x) q)) == 0 {FPP0}
-- ∀ r::a . x == 0 ⇒ (foldPolinomio id r (+) (*) (Cte x)) * (foldPolinomio id r (+) (*) q) == 0 {FPC0}
-- ∀ r::a . x == 0 ⇒ id x * (foldPolinomio id r (+) (*) q) == 0 {ID0}
-- ∀ r::a . x == 0 ⇒ x * (foldPolinomio id r (+) (*) q) == 0

-- O bien x == 0 o bien x != 0

-- Caso 1: x == 0

-- ∀ r::a . x == 0 ⇒ x * (foldPolinomio id r (+) (*) q) == 0 {remplazamos x por 0}
-- ∀ r::a . 0 == 0 ⇒ 0 * (foldPolinomio id r (+) (*) q) == 0 {aritmética}
-- True ⇒ True {queda demostrada la implicación}

-- Caso 2: x != 0

-- ∀ r::a . x == 0 ⇒ x * (foldPolinomio id r (+) (*) q) == 0 {CASO 2}
-- False ⇒ x * (foldPolinomio id r (+) (*) q) == 0 {queda demostrada la implicación}

-- Caso Base: P(Suma p q) = ∀ r::a . esRaiz r (Suma p q) ⇒ esRaiz r (Prod (Suma p q) q)

-- Caso Base: P(Prod p q) = ∀ r::a . esRaiz r (Prod p q) ⇒ esRaiz r (Prod (Prod p q) q)

--- ii. Num a => ∀ p::Polinomio a . ∀ k::a . ∀ e::a . evaluar e (derivado (Prod (Cte k) p)) = evaluar e (Prod (Cte k) (derivado p))

-- iii. Num a => ∀ p::Polinomio a . sinConstantesNegativas p ⇒ sinConstantesNegativas (derivado p)

-- Ejercicio Extra

-- elem :: Eq a => a -> [a] -> Bool
-- elem e [] = False {EL0}
-- elem e (x:xs) = e == x || elem e xs {EL1}

-- maximum :: Ord a => [a] -> a
-- maximum [x] = x {MAX0}
-- maximum (x:xs) = if x < maximum xs then maximum xs else x {MAX1}

-- Ord a => ∀ xs::[a] . ∀ e::a . elem e xs ⇒ e <= maximum xs

-- Quiero ver que si a es un tipo ordenado, entonces para toda lista xs de elementos de tipo a y
-- para todo elemento e de tipo a, si e se encuentra en xs, entonces e es menor o igual al máximo de xs

-- (Ord a ⇒ (∀ xs::[a] . ∀ e::a . elem e xs ⇒ (e <= maximum xs)))

-- Empecemos por ver que a puede ser un tipo ordenado o no. Abramos los casos y exploremos las posibilidades.

-- Caso 1: a no es un tipo ordenado

-- Si a no es un tipo ordenado, entonces no podemos comparar elementos de tipo a. Si el antecedente de la implicación
-- es falso, entonces la implicación es verdadera. Por lo tanto, la implicación es verdadera si a no es un tipo ordenado.

-- Caso 2: a es un tipo ordenado

-- Si a es un tipo ordenado, entonces podemos comparar elementos de tipo a.
-- Procedemos a probar que el consecuente sea verdadero.

-- Predicado unario: P(xs) = ∀ e::a . elem e xs ⇒ e <= maximum xs

-- Caso base: P([]) =

-- ∀ e::a . elem e [] ⇒ e <= maximum [] {EL0}
-- ∀ e::a . False ⇒ e <= maximum [] {queda demostrada la implicación}

-- Hipótesis inductiva: P(xs) = ∀ e::a . elem e xs ⇒ e <= maximum xs
-- Paso inductivo: P(x:xs) = ∀ e::a . elem e (x:xs) ⇒ e <= maximum (x:xs)

-- ∀ e::a . elem e (x:xs) ⇒ e <= maximum (x:xs) {EL1}
-- ∀ e::a . e == x || elem e xs ⇒ e <= maximum (x:xs) {abrimos en dos casos}

-- Caso 1: e == x

-- ∀ e::a . e == x || elem e xs ⇒ e <= maximum (x:xs) {CASO 1}
-- ∀ e::a . true || elem e xs ⇒ e <= maximum (x:xs) {lógica}
-- ∀ e::a . true ⇒ e <= maximum (x:xs) {MAX1}
-- ∀ e::a . true ⇒ e <= (if x < maximum xs then maximum xs else x) {abrimos en dos casos}

-- Caso 1.1: x < maximum xs

-- ∀ e::a . true ⇒ e <= (if x < maximum xs then maximum xs else x) {CASO 1.1}
-- ∀ e::a . true ⇒ e <= maximum xs {HI}

-- Caso 1.2: x >= maximum xs

-- ∀ e::a . true ⇒ e <= (if x < maximum xs then maximum xs else x) {CASO 1.2}
-- ∀ e::a . true ⇒ e <= x {CASO 1}
-- ∀ e::a . true ⇒ x == x {lógica}

-- Caso 2: e != x

-- ∀ e::a . e == x || elem e xs ⇒ e <= maximum (x:xs) {CASO 2}
-- ∀ e::a . false || elem e xs ⇒ e <= maximum (x:xs) {lógica}
-- ∀ e::a . elem e xs ⇒ e <= maximum (x:xs) {HI}
-- ∀ e::a . elem e xs ⇒ e <= maximum xs ⇒ e <= maximum (x:xs)

-- Sabemos que e se encuentra en xs y también sabemos que e es menor o igual al máximo de xs y que e es distinto de x.
-- Si x es el máximo de xs, entonces e es menor a x, vale la implicación. Si x no es el máximo de xs, entonces e es
-- menor o igual al máximo de xs, sigue valiendo la implicación. Por lo tanto, la implicación es verdadera.

-- Ejercicio introducción a la recursión

-- sum :: Num a => [a] -> a
-- sum [] = 0 {SUM0}
-- sum (x:xs) = x + sum xs {SUM1}

-- Demostrar que para todas xs, ys listas finitas vale que: sum (xs ++ ys) = sum xs + sum ys
-- Queremos probar que ∀ xs::[a] . ∀ ys::[a] . sum (xs ++ ys) = sum xs + sum ys haciendo inducción en xs.

-- Predicado unario: P(xs) = ∀ ys::[a] . sum (xs ++ ys) = sum xs + sum ys

-- Caso base: P([]) =

-- ∀ ys::[a] . sum ([] ++ ys) = sum [] + sum ys {++AUX2}
-- ∀ ys::[a] . sum ys = sum [] + sum ys {SUM0}
-- ∀ ys::[a] . sum ys = 0 + sum ys {aritmética}
-- ∀ ys::[a] . sum ys = sum ys {queda demostrada la igualdad}

-- Hipótesis inductiva: P(xs) = ∀ ys::[a] . sum (xs ++ ys) = sum xs + sum ys

-- Paso inductivo: P(x:xs) = ∀ ys::[a] . sum ((x:xs) ++ ys) = sum (x:xs) + sum ys

-- ∀ ys::[a] . sum ((x:xs) ++ ys) = sum (x:xs) + sum ys {++}
-- ∀ ys::[a] . sum (foldr (:) ys (x:xs)) = sum (x:xs) + sum ys {foldr}
-- ∀ ys::[a] . sum (x : foldr (:) ys xs) = sum (x:xs) + sum ys {SUM1}
-- ∀ ys::[a] . x + sum (foldr (:) ys xs) = sum (x:xs) + sum ys {++}
-- ∀ ys::[a] . x + sum (xs ++ ys) = sum (x:xs) + sum ys {HI}
-- ∀ ys::[a] . x + sum xs + sum ys = sum (x:xs) + sum ys {SUM1}
-- ∀ ys::[a] . x + sum xs + sum ys = x + sum xs + sum ys {queda demostrada la igualdad}