Extensionalidad:

```
Dadas f, g :: a -> b, probar f = g se reduce a probar:
                              ∀ x::a . f x = g x
```

Propiedades útiles:

`∀F :: a -> b . ∀G :: a -> b . ∀Y :: b . ∀Z :: a .`

1. `F = G` $\Leftrightarrow$ `∀x::a. F x = G x`
2. `F = \x -> Y` $\Leftrightarrow$ `∀x::a. F x = Y`
3. `(x -> Y) Z` $\underset{\beta}{=}$ `Y` reemplazando `x` por `Z`
4. `\x -> F x` $\underset{\eta}{=}$ `F`

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

Ejemplos

2. `F x = x * 2   y   G = \x -> x * 2   =>   F = G`
3. `(\x -> x + 5) 3 =β (3 + 5)`

Lemas de generación

Para pares: `Si p :: (a, b), entonces ∃x :: a. ∃y :: b. p = (x, y).`

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 y
```

Idea general para una demostración por inducción estructural

- Leer la propiedad, entenderla y convencerse de que es verdadera.
- Plantear la propiedad como predicado unario.
- Plantear el esquema de inducción.
- Plantear y resolver el o los caso(s) base.
- Plantear y resolver el o los caso(s) inductivo(s).

Dato: Los argumentos no recursivos quedan cuantificados universalmente.

Ejemplo: principio de inducción sobre listas

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

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `[a]` tal que:

- $\mathcal{P}(\texttt{[]})$
- $\forall \texttt{x} :: \texttt{a}.\ \forall \texttt{xs} :: \texttt{[a]}.\ (\underbrace{\mathcal{P}(\texttt{xs})}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{x : xs})}_{\text{T.I.}})$

Entonces $\forall \texttt{xs} :: \texttt{[a]}.\ \mathcal{P}(\texttt{xs})$.

Ejemplo: principio de inducción sobre árboles binarios

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

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `AB a` tal que:

- $\mathcal{P}(\texttt{Nil})$
- $\forall \texttt{i} :: \texttt{AB a}.\ \forall \texttt{r} :: \texttt{a}.\ \forall \texttt{d} :: \texttt{AB a}.$
  $$(\underbrace{(\mathcal{P}(\texttt{i}) \wedge \mathcal{P}(\texttt{d}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Bin i r d})}_{\text{T.I.}})$$

Entonces $\forall \texttt{x} :: \texttt{AB a}.\ \mathcal{P}(\texttt{x})$.

Ejemplo: principio de inducción sobre polinomios

```
data Poli a = X
            | Cte a
            | Suma (Poli a) (Poli a)
            | Prod (Poli a) (Poli a)
```

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `Poli a` tal que:

- $\mathcal{P}(\texttt{X})$
- $\forall \texttt{k} :: \texttt{a}.\ \mathcal{P}(\texttt{Cte k})$
- $\forall \texttt{p} :: \texttt{Poli a}.\ \forall \texttt{q} :: \texttt{Poli a}.$
  $$(\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Suma p q})}_{\text{T.I.}})$$
- $\forall \texttt{p} :: \texttt{Poli a}.\ \forall \texttt{q} :: \texttt{Poli a}.$
  $$(\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Prod p q})}_{\text{T.I.}})$$

Entonces $\forall \texttt{x} :: \texttt{Poli a}.\ \mathcal{P}(\texttt{x})$.

Si no podemos continuar en un paso de la demostración, optamos por demostrar un lema sobre la operación.

Ejercicios:

2

Demostrar las siguientes igualdades utilizando el principio de extensionalidad funcional:

con la definición usual de la composición: `(.) f g x = f (g x)`.

I.

```
flip . flip = id

-- Por extensionalidad {EXT} alcanza ver que
-- ∀f::a->b->c. ∀x::a. ∀y::b.

(flip . flip) f x y = id f x y

-- Demostración:

         (flip . flip) f x y
{(.)}  = flip (flip f x y)
{flip} = flip f y x
{flip} = f x y
{id}   = id f x y
```

QED

II.

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

-- Por extensionalidad para pares {EXT} alcanza con ver que
-- ∀ f::(a,b)->c. ∀p :: (a,b) ∃x::a. ∃y::b. p = (x,y)

uncurry (curry f) (x,y) = f (x,y)

--Demostración

        uncurry (curry f) (x,y)
{UN1} = (curry f) x y
{C1}  = f (x,y)
```

QED

IV.

```
∀ f::a->b . ∀ g::b->c . ∀ h::c->d . ((h . g) . f) = (h . (g . f))

-- Por extensionalidad alcanza ver que
-- ∀ f::a->b . ∀ g::b->c . ∀ h::c->d . ∀x::a

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

-- Demostración

        ((h . g) . f) x
{(.)} = (h . g) (f x)
{(.)} = h (g (f x))
{(.)} = h . ((g . f) x)
{(.)} = h . (g . f) x
```

QED

3

Demostrar las siguientes propiedades:

I.

```
∀ xs::[a]. P(xs): length (duplicar xs) = 2 * length xs

-- Por inducción sobre listas

-- Caso base P([]): length (duplicar []) = 2 * length []

↓         length (duplicar [])
↓ {D0}  = length []
  {L0}  = 0
↑ {INT} = 2*0
↑ {L0}  = 2 * length []

-- Paso inductivo P(x:xs), con HI: P(xs)

--qvq

length (duplicar (x:xs)) = 2 * length xs = 2 * length (x:xs)

-- Demostración

--izq
length (duplicar (x:xs))
 {D1} = length (x:x: duplicar xs)
2{L1} = 1 + 1 + length (duplicar xs)
{INT} = 2 + length (duplicar xs)

--der
2 * length (x:xs)
{L1} = 2* (1 + length xs)
{INT} = 2 + 2 * (length xs)
{HI} = 2 + length (duplicar xs)
```

QED

II.

```
∀ xs::[a]. ∀ ys::[a]. P(xs): length (xs ++ ys) = length xs + length ys

-- inducción en xs

-- Caso P([]) : length ([] ++ ys) = length [] + length ys

--izq
         length ([] ++ ys)
{++0} = length ys

--der
        length [] + length ys
{L0}  = 0 + length ys
{INT} = length ys

-- Paso inductivo P(x:xs), con HI P(xs)
--qvq
length ((x:xs) ++ ys) = length (x:xs) + length ys

--Demostración

--izq
        length ((x:xs) ++ ys)
{++1} = length (x: (xs ++ ys))
{L1}  = 1 + length (xs ++ ys)
{HI}  = 1 + length xs + length ys

--der
```

```
       length (x:xs) + length ys
{L1} = 1 + length xs + length ys
```

QED

III.

```
∀ xs::[a]. ∀ x::a. P(xs): append [x] xs = x:xs

-- Por inducción en xs

-- Caso base P([]): append [x] [] = x:[]

        append [x] []
{A0}  = foldr (:) [] [x]
{F1}  = (:) x (foldr (:) [] [])
{F0}  = (:) x []
{(:)} = x:[]

-- Paso inductivo, con HI P(xs)

P(x':xs): append [x] (x':xs) = x:x':xs

append [x] (x':xs)
{A0} = foldr (:) (x':xs) [x]
{F1} = (:) x (foldr (:) (x':xs) [])
{F0} = (:) x (x':xs)
{(:)} = x:x':xs
```

QED

IV.

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

-- Por inducción en xs

-- Caso base P([]): length (map f []) = length []

length (map f [])
{M0} = length []

-- Paso inductivo, con HI P(xs)

P(x:xs): length (map f (x:xs)) = length (x:xs)

-- Demo

-- izq

length (map f (x:xs))
{M1} = length (f x : map f xs)
{L1} = 1 + length (map f xs)

-- der

       length (x:xs)
{L0} = 1 + length xs
{HI} = 1 + length (map f xs)
```

QED

V.

```
∀ xs::[a]. ∀ p::a->Bool. ∀ e::a. P(xs): ((elem e (filter p xs)) => (elem e xs)) (asumiendo Eq a)

-- inducción en xs
```

```
-- Caso base P([]): ((elem e (filter p [])) => (elem e []))

        (elem e (filter p []))
{FI0} = elem e []

-- Paso inductivo, con P(xs) de HI

P(x:xs): ((elem e (filter p (x:xs))) => (elem e (x:xs)))

        (elem e (filter p (x:xs))
{FI1} = ((elem e (if p x then x : (filter p xs) else (filter p xs)))

-- por inducción en bools

-- Caso no p x

         elem e (filter p xs) {HI} ⇒ elem e xs
{BOOl} = True

-- Caso p x

        elem e (x : (filter p xs))
{E1} =  elem e == x | elem (filter p xs)
{HI} => elem e == x | elem e xs

-- Caso e == x

         elem e == x | elem (filter p xs)
{BOOL} = True | elem (filter p xs)
{BOOL} = True

-- Caso e != x

elem   = e == x | elem e xs
{HI}   = e == x | True
{BOOL} = True
```

QED

VI.

```
∀ xs::[a]. ∀ x::a. P(xs): ponerAlFinal x xs = xs ++ (x:[])

-- inducción en xs

-- Caso base P([]): ponerAlFinal x [] = [] ++ (x:[])

-- izq
        ponerAlFinal x []
{P0} = (foldr (:) (x:[])) []
{F0} = x:[]

-- der
        [] ++ (x:[])
{++0} = x:[]

-- Paso inductivo P(y:xs): ponerAlFinal x (y:xs) = (y:xs) ++ (x:[]), asumo P(xs)

-- izq
        ponerAlFinal x (y:xs)
{P0}  = foldr (:) (x:[]) (y:xs)
{F1}  = (:) y (foldr (:) (x:[]) xs)
{(:)} = y : (foldr (:) (x:[]) xs)
```

```
-- der
        (y:xs) ++ (x:[])
{++1} = y : (xs ++ x:[])
{HI}  = y : (ponerAlFinal x xs)
{P0}  = y : (foldr (:) (x:[]) xs)
```

QED

VII.

```
reverse = foldr (\x rec -> rec ++ (x:[])) []

Por extensionalidad, basta ver que ∀xs::[a]

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

-- Caso base P([]): reverse [] = foldr (\x rec -> rec ++ (x:[])) [] []

        reverse []
{R0}  = foldl (flip (:)) [] []
{FL0} = []
{FR0} = foldr (\x rec -> rec ++ (x:[])) [] []

-- Paso inductivo, asumo P(xs), qvq
P(y:xs): reverse (y:xs) = foldr (\x rec -> rec ++ (x:[])) [] (y:xs)

--izq

          reverse (y:xs)
{R0}    = foldl (flip (:)) [] (y:xs)
{FL1}   = foldl (flip (:)) ((flip (:) [] y)) xs
{FLIP}  = foldl (flip (:)) ((:) y []) xs
{(:)}   = foldl (flip (:)) (y:[]) xs
{}      = foldl (flip (:)) [y] xs


--der
        foldr (\x rec -> rec ++ (x:[])) [] (y:xs)
{F1}  = (\x rec -> rec ++ (x:[])) y (foldr (\x rec -> rec ++ (x:[])) [] xs)
β     = (\rec -> rec ++ [y]) (foldr (\x rec -> rec ++ (x:[])) [] xs)
{HI}  = (\rec -> rec ++ [y]) (reverse xs)
β     = (reverse xs) ++ [y]
{R0}  = (foldl (flip (:)) [] xs) ++ [y]

-- Queremos ver que Q(xs): foldl (flip (:)) [y] xs = (foldl (flip (:)) [] xs) ++ [y]

-- Por inducción

-- Caso base

Q([]): foldl (flip (:)) [y] [] =? (foldl (flip (:)) [] []) ++ [y]

        foldl (flip (:)) [y] []
{FL0} = [y]

        (foldl (flip (:)) [] []) ++ [y]
{FL0} = [] ++ [y]
{++0} = [y]

-- Paso inductivo, asumo Q(xs) y qvq

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

```
-- izq
         foldl (flip (:)) [y] (x:xs)
{FL1}  = foldl (flip (:)) ((flip (:)) [y] x) xs
{FLIP} = foldl (flip (:)) ((:) x [y]) xs
{(:)}  = foldl (flip (:)) [x,y] xs

-- der

         (foldl (flip (:)) [] (x:xs)) ++ [y]
{FL1}  = (foldl (flip (:)) ((flip (:)) [] x) xs) ++ [y]
{FLIP} = (foldl (flip (:)) ((:) x []) xs) ++ [y]
{(:)}  = (foldl (flip (:)) [x] xs) ++ [y]


-- hay que probar que ∀x. R(xs): foldl (flip (:)) [x,y] xs = (foldl (flip (:)) [x] xs) ++ [y]

-- caso base

        foldl (flip (:)) [x,y] []
{FL1} = [x,y]

        (foldl (flip (:)) [x] []) ++ [y]
{FL1} = [x] ++ [y]
{++1} = [x,y]

-- Paso inductivo, vale R(xs), qvq

R(w:xs): foldl (flip (:)) [x,y] (w:xs) = (foldl (flip (:)) [x] (w:xs)) ++ [y]

-- izq
        foldl (flip (:)) [x,y] (w:xs)
{FL1} = foldl (flip (:)) (w:[x, y]) xs
{--}  = foldl (flip (:)) [w, x, y] xs

-- der

        foldl (flip (:)) [x] (w:xs)) ++ [y]
{FL1} = foldl (flip (:)) (w:[x]) xs ++ [y]
{--}  = foldl (flip (:)) [w,x] xs ++ [y]

{HI} foldl (flip (:)) [w, x, y] xs = foldl (flip (:)) [w,x] xs ++ [y]
```

QED

VIII.

```
∀ xs::[a]. ∀ x::a. P(xs): head (reverse (ponerAlFinal x xs)) = x

-- induccion

-- Caso base P([])

         head (reverse (ponerAlFinal x []))
{P0}   = head (reverse (foldr (:) [x] []))
{FR0}  = head (reverse [x]))
{R0}   = head (foldl (flip (:)) [] [x])
{FL0}  = head ([x])
{HEAD} = x

-- Paso inductivo, asumo P(xs)

P(y:xs): head (reverse (ponerAlFinal x (y:xs))) = x

        head (reverse (ponerAlFinal x (y:xs)))
{P0}  = head (reverse (foldr (:) [x] (y:xs))
```

```
{FR1} = head (reverse ((:) y (foldr (:) [x] xs)))
{(:)} = head (reverse y : (foldr (:) [x] xs)))

-- Demostramos el  lema: P(xs): foldr (:) [x] xs  = xs ++ [x]

-- Por inducción

-- Caso base

         foldr (:) [x] []
{FR0} = [x]
{++0} = [] ++ [x]

-- Paso inductivo, asumo P(xs)

P(z:xs): foldr (:) [x] (z:xs)  = (z:xs) ++ [x]

        foldr (:) [x] (z:xs)
{FR1} = (:) z (foldr (:) [x] xs)
{HI}  = (:) z (xs ++ [x])
{(:)} = z : (xs ++ [x])

        (z:xs) ++ [x]
{++1} = z : (xs ++ [x])

-- QED, lo llamamos {L1}

-- Seguimos.

       head (reverse y : (foldr (:) [x] xs)))
{L1} = head (reverse (y : (xs ++ [x])))
{--} = head (reverse (ws++[x]) -- notamos  (y : (xs ++ [x])) = (ws++[x])

-- Probemos que P(ws): (ws++[x]) = ponerAlFinal x ws

-- caso base []

        ([]++[x])
{++0} = [x]

        ponerAlFinal x []
{P0}  = foldr (:) [x] []
{FL0} = [x]

-- paso inductivo P(ws): ((w:ws)++[x]) = ponerAlFinal x (w:ws)

        ((w:ws)++[x])
{++1} = w:(ws++[x])
{HI}  = w : ponerAlFinal x ws
{P0}  = w : (foldr (:) [x] ws)

        ponerAlFinal x (w:ws)
{P0}  = foldr (:) [x] (w:ws)
{FR1} = w : (foldr (:) [x] ws)

-- {L2} (ws++[x]) = ponerAlFinal x ws

-- Seguimos.

       head (reverse (ws++[x])
{L2} = head (reverse (ponerAlFinal x ws))
{HI} = x
```

QED

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)]
{Z'0} zip' [] ys = []
{Z'1} zip' (x:xs) ys = if null ys then [] else (x, head ys):zip' xs (tail ys)

-- Demostrar zip = zip'

-- Por extensionalidad, alcanza ver que ∀xs::[a]. ∀ys::[b]
-- P(xs): zip xs ys = zip' xs ys

-- Por inducción en xs.

-- caso base xs = []

          zip [] ys
{Z0}    = foldr (\x rec ys ->
                  if null ys
                    then []
                    else (x, head ys) : rec (tail ys))
                  (const []) [] ys
{F0}    = (const []) ys
{const} = []
{Z'0}   = zip' [] ys

-- paso inductivo, asumo P(xs), vemos P(z:xs)
-- defino
g = (\x rec ys -> if null ys then [] else (x, head ys) : rec (tail ys))
g z = (rec ys -> if null ys then [] else (z, head ys) : rec (tail ys)) <-β

       zip (z:xs) ys
{Z0} = foldr g (const []) (z:xs) ys
{F1} = g z (foldr g (const []) xs) ys
β    = if null ys then [] else (z, head ys) : (foldr g (const []) xs) (tail ys)
{Z0} = if null ys then [] else (z, head ys) : zip xs (tail ys)

        zip' (z:xs) ys
{Z'1} = if null ys then [] else (z, head ys) : zip' xs (tail ys)
{HI}  = if null ys then [] else (z, head ys) : zip xs (tail ys)

-- habría que ver caso [] y caso diferente de [], ya que tail se indefine si [], pero es medio
trivial
```

QED

6

Dadas las siguientes funciones:

```
     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
```

Indicar si las siguientes propiedades son verdaderas o falsas. Si son verdaderas, realizar una demostración. Si son falsas, presentar un contraejemplo.

VI.

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

--Por inducción en xs

--Caso base
P([]): length (union [] ys) ≤ length [] + length ys

        length (union [] ys)
{U0}  = length (nub []++ys)
{++0} = length (nub ys)

        length [] + length ys
{L0}  = 0 + length ys
{INT} = length ys

-- nub ys ≤ ys, ∀ys::[a]

-- Caso inductivo, asumo P(xs)

P(x:xs): length (union (x:xs) ys) ≤ length (x:xs) + length ys

        length (union (x:xs) ys)
{U0}  = length (nub (x:xs)++ys)
{++1} = length (nub x:(xs++ys))
{N1}  = length (x: filter (\y -> x /= y) (nub xs++ys))
β     = length (x: filter x /= (nub xs++ys))
{L1}  = 1 + length (filter x /= (nub xs++ys))

--der
length (x:xs) + length ys
{L1} = 1 + length xs + length ys
{HI} = 1 + length (union xs ys)
{U0} = 1 + length (nub xs++ys)

-- sabemos que por ∀p::(a->Bool). length (filter p xs) ≤ length xs,

length (filter x /= (nub xs++ys)) ≤ length (filter x /= (nub xs++ys)) = True
```

QED

9

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

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

       foldAB :: (b -> a -> b -> b) -> b -> AB a -> b
{FAB0} foldAB _ z Nil = z
{FAB1} foldAB f z (Bin i c r) = f (foldAB f z i) c (foldAB f z r)

     altura :: AB a -> Integer
{AL} altura = foldAB (\i _ r -> 1 + max i r) 0

     cantNodos :: AB a -> Integer
{CA} cantNodos = foldAB (\i _ r -> i+1+r) 0
```

```
-- queremos probar que
∀ x::AB a . P(x): altura x ≤ cantNodos x

-- Por inducción en x

-- Caso base P(Nil)

--qvq altura Nil ≤ cantNodos Nil

        altura Nil
{AL}  = foldAB (\i _ r -> 1 + max i r) 0 Nil
{FAB} = 0

        cantNodos Nil
{CA}  = foldAB (\i _ r -> i+1+r) 0 Nil
{FAB} = 0

--Caso recursivo P(i c r)

-- queremos probar que
foldAB (\i _ r -> 1 + max i r) 0 (i c r)
≤
foldAB (\i _ r -> i+1+r) 0 (i c r)


-- demostración
f = (\i _ r -> 1 + max i r)
g = (\i _ r -> i + 1 + r)

--izq
         foldAB f 0 (Bin i c r)
{FAB1} = f (foldAB f 0 i) c (foldAB f 0 r)
{f}    = (\i _ r -> 1 + max i r) (foldAB f 0 i) c (foldAB f 0 r)
β      = (1 + max (foldAB f 0 i) (foldAB f 0 r))

--der
         foldAB (\i _ r -> i+1+r) 0 (i c r)
{FAB1} = g (foldAB g 0 i) c (foldAB g 0 r)
{g}    = (\i _ r -> i + 1 + r) (foldAB g 0 i) c (foldAB g 0 r)
β      = (foldAB g 0 i) + 1 + (foldAB g 0 i)
{INT}  = 1 + (foldAB g 0 i) + (foldAB g 0 r)


-- vemos que

      (1 + max (foldAB f 0 i) (foldAB f 0 r)) ≤ (1 + (foldAB g 0 i) + (foldAB g 0 r))
     ⇔
       max (foldAB f 0 i) (foldAB f 0 r) ≤ (foldAB g 0 i) + (foldAB g 0 r)
{f}  = max (altura i) (altura r) ≤ (cantNodos i) + (cantNodos r)

-- Por HI:
(altura i) ≤ (cantNodos i)
(altura r) ≤ (cantNodos r)

=>

max (altura i) (altura r) ≤ (cantNodos i) ≤ (cantNodos i) + (cantNodos r)
-- OR
max (altura i) (altura r) ≤ (cantNodos r) ≤ (cantNodos i) + (cantNodos r)
```

QED

12

Dados:

```
data Polinomio a = X
                  | Cte a
                  | Suma (Polinomio a) (Polinomio a)
                  | Prod (Polinomio a) (Polinomio a)

        evaluar :: Num a => a -> Polinomio a -> a
{E0}    evaluar e X = e
{E1}    evaluar e (Cte x) = x
{E2}    evaluar e (Suma p q) = evaluar e p + evaluar e q
{E3}    evaluar e (Prod p q) = evaluar e p * evaluar e q

        derivado :: Num a => Polinomio a -> Polinomio a
{PL0}   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
{SCN}   sinConstantesNegativas = foldPoli True (>=0) (&&) (&&)

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

Queremos probar:

I.

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

-- Por inducción en P(p):

-- Caso base P(X):

          esRaiz r X ⇒ esRaiz r (Prod X q)
2*{ER0} = (evaluar r X) == 0 ⇒ (evaluar r (Prod X q)) == 0
{E0}    = r == 0 ⇒ (evaluar r (Prod X q)) == 0
{E3}    = r == 0 ⇒ (evaluar r X * evaluar r q) == 0
{E0}    = r == 0 ⇒ (r * evaluar r q) == 0

        -- Caso (r==0) = True:

                True ⇒ (0 * evaluar 0 q) == 0
        {E0}    True ⇒ 0 == 0
        {Bool}    True ⇒ True
        {Bool}    True

        -- Caso (r==0) = False:

                False ⇒ r * evaluar r q
        {Bool} = True

-- Caso base P(Cte a):
-- Queremos ver que para cualquier x::a vale

        (esRaiz r (Cte x) ⇒ esRaiz r (Prod (Cte x) q))
{ER0} = (evaluar r (Cte x)) == 0 ⇒ (evaluar r (Prod (Cte x) q)) == 0
{E1}  = x == 0 ⇒ (evaluar r (Prod (Cte x) q)) == 0
{E3}  = x == 0 ⇒ (evaluar r (Cte x) * evaluar r q) == 0
{E1}  = x == 0 ⇒ (x * evaluar r q) == 0
```

```
        -- Caso (x==0) = True:

                True ⇒ (0 * evaluar r q) == 0
        {INT}   True ⇒ 0 == 0
        {Bool}  True ⇒ True
        {Bool}  True

        -- Caso (x==0) = False:

                False ⇒ (0 * evaluar r q) == 0
        {Bool} = True

{-
Paso inductivo:
Nuestra HI será P(p): (esRaiz r p ⇒ esRaiz r (Prod p q))

Por el tipo de dato, tenemos que ver 2 cosas:
        Caso recursivo P(Suma n m):
                ∀n::a. ∀m::a.
                P(Suma n m): (esRaiz r (Suma n m) ⇒ esRaiz r (Prod (Suma n m) q))
        Caso recursivo P(Prod n m):
                ∀n::a. ∀m::a.
                P(Prod n m): (esRaiz r (Prod n m) ⇒ esRaiz r (Prod (Prod n m) q))
-}

-- Caso P(Suma n m):

        (esRaiz r (Suma n m) ⇒ esRaiz r (Prod (Suma n m) q))
{ER0} = (evaluar r (Suma n m)) == 0 ⇒ (evaluar r (Prod (Suma n m) q))) == 0
{E2}  = (evaluar r n + evaluar r m) == 0 ⇒ (evaluar r (Prod (Suma n m) q))) == 0
{E3}  = (evaluar r n + evaluar r m) == 0 ⇒ (evaluar r (Suma n m) * evaluar r q) == 0
{E2}  = (evaluar r n + evaluar r m)==0 ⇒ ((evaluar r n + evaluar r m)*evaluar r q)==0

-- Caso ((evaluar r n + evaluar r m) == 0):
         0 == 0 ⇒ (0 * evaluar r q) == 0
INT    = 0 == 0 ⇒ 0 == 0
2*Bool = True ⇒ True
Bool   = True

-- Caso ((evaluar r n + evaluar r m) != 0), llamamos
-- (evaluar r n + evaluar r m) = k, k!=0, luego:
       k == 0 ⇒ (k * evaluar r q) == 0
Bool = False ⇒ (k * evaluar r q) == 0
Bool = True

-- Caso P(Prod n m):

        (esRaiz r (Prod n m) ⇒ esRaiz r (Prod (Prod n m) q))
{ER0} = (evaluar r (Prod n m)) == 0 ⇒ (evaluar r (Prod (Prod n m) q)) == 0
{E3}  = (evaluar r (Prod n m)) == 0 ⇒ ((evaluar r (Prod n m)) * (evaluar r q))) == 0

-- Caso (evaluar r (Prod n m)) == 0:

       0 == 0 ⇒ (0 * (evaluar r q)) == 0
INT  = 0 == 0 ⇒ 0 == 0
Bool = True ⇒ True
Bool = True

-- Caso (evaluar r (Prod n m)) != 0,
-- Decimos que (evaluar r (Prod n m)) == k, k != 0

       k == 0 ⇒ (k * (evaluar r q)) == 0
Bool = False ⇒ (k * (evaluar r q)) == 0
Bool = True
```