Tenemos:

```
elem :: Eq a => a -> [a] -> Bool
{E0} elem e [] = False
{E1} elem e (x:xs) = (e == x) || elem e xs

maximum :: Ord a => [a] -> a
{M0} maximum [x] = x
{M1} maximum (x:y:ys) = if x < maximum (y:ys) then maximum (y:ys) else x
```

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

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

Por inducción en ys:

Caso ys = [ ]

P([ ]) = elem e [ ] ⇒ e ≤ maximum [ ]

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

Por Bool, la implicación es verdadera.

Caso ys = x:xs

HI = P(xs)

qvq ∀ e:: a, elem e (x:xs) ⇒ e ≤ maximum (x:xs)

$\underset{\{E1\}}{=}$ e == x || elem e xs ⇒ e ≤ maximum (x:xs)

L.G. Bool ⇒ e==x = True ∨ e==x = False

L.G. Listas ⇒ xs = [ ] ∨ xs = z:zs

Caso e==x = True ⇒ xs = [ ]

⋆ $\underset{\{Bool\}}{=}$ = True ⇒ e ≤ maximum (x [ ])

$\underset{\{Bool\}}{=}$ e ≤ maximum (x [ ])

$\underset{\{M0\}}{=}$ e ≤ x $\underset{\{Ord\ (e==x)\}}{=}$ True

---

Caso e==x = False ⇒ xs = [ ]

⋆ $\underset{\{Bool\}}{=}$ elem e [ ] ⇒ e ≤ maximum [ ] $\cdot_{\text{vale por caso base}}$

Caso e==x = True, xs = z:zs

⋆ $\underset{\{Bool\}}{=}$ e ≤ maximum (x:z:zs) $\underset{\{M1\}}{=}$ e ≤ if x < maximum (z:zs) then maximum (z:zs) else x

L.G. Bool ⇒ x < maximum (z:zs) es True o False

Caso True:

Por Ord, e==x $<$ maximum (z:zs) $\Rightarrow$ e $\leq$ maximum (z:zs) $\underset{\{\text{if}\}}{=}$

Caso False:

$\underset{\{\text{If}\}}{=}$ e $\leq$ x vale por Ord y e==x

Caso e==x es False, xs es z:zs

$\underset{\{\text{Bool}\}}{=}$ elem e (z:zs) $\Rightarrow$ elem $\leq$ maximum (x:z:zs)

$\underset{\{\text{M1}\}}{=}$ elem e (z:zs) $\Rightarrow$ e $\leq$ if x $<$ maximum (z:zs) then maximum (z:zs) else x

---

LG Bool

Caso True:

elem e(z:zs) $\Rightarrow$ e $\leq$ maximum (z:zs) vale por HI

Caso False:

elem e (z:zs) $\Rightarrow$ e $\leq$ x

LG. Bool

False la implicación es verdadera

Caso True qvq e $\leq$ x

Por HI, Bool e $\leq$ maximum (z:zs) $<$ x para todo e $\leq$ x $\square$

Dadas las siguientes definiciones:

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

foldl :: (b -> a -> b) -> b -> [a] -> b
{F0} foldl f ac [] = ac
{F1} foldl f ac (x:xs) = foldl f (f ac x) xs

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

flip :: (a -> b -> c) -> (b -> a -> c)
{FL} flip f = (\x y -> f y x)
```

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

length ys $\stackrel{?}{=}$ length (reverse ys)

length (reverse ys) $\underset{\{R\}}{=}$ length (foldl flip (:)) [ ]

…

Nota vamos a llegar a foldl (flip (:)) (x:[ ]) xs, luego no se puede usar HI, luego:

Lema: ∀ ac :: [a], ∀ ys:: [a]. length (foldl (flip (:)) ac ys) = length ys + length ac

Lo probamos por inducción en ys:

P(ys) = ∀ ac::[a] length (foldl (flip (:)) ac ys) = length ac + length ys

---

Caseo ys = [ ]

length (foldl flip (:)) ac [ ] $\underset{\{F0\}}{=}$ length ac $\underset{\{Int\}}{=}$ = 0+length ac $\underset{\{L0\}}{=}$ length [] + length ac

Tarea: caso ys = x:xs

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

Dada esta versión alternativa con recursión explícita:

```
take' :: [a] -> Int -> [a]
{T0} take' [] _ = []
{T1} take' (x:xs) n = if n == 0 then [] else x : take' xs (n-1)
```

¿Podemos probar que `take'` = `flipTake`?

Por extensionalidad basta probar que

∀ xs::[a], take' xs = flipTake xs

Por extensionalidad

∀ xs::[a] ∀ n:: Int xs n = flipTake xs n

Inducción en xs

P(xs): ∀ n::Int tak xs n = flipTake xs n

Caso xs = [ ]

take' [] n $\underset{\{T0\}}{=}$ [ ]

- (**notamos** f = (x rec n -> if n == 0 then [] else x : rec (n-1)))

flipTake [] n $\underset{\{FT\}}{=}$ foldr f (const [ ]) n $\underset{\{foldr\}}{=}$ const [] n $\underset{\{const\}}{=}$ []

Caso xs = y:ys

HI = P(xs)

take' (y:ys) n $\underset{\{T1\}}{=}$ if n==0 then [ ] else y take' ys (n-1)

fliptake (y:ys) n = foldr f [ ] (y:ys) n $\underset{\{foldr\}}{=}$ f y (foldr f [ ] ys) n $\underset{\beta}{=}$ (\rec n → if n==0 then [] else y:(rec (n-1)) (foldr f const [ ] ys)) n

$\underset{\{2\beta\}}{=}$ if n==0 then [ ] else y (foldr f (const [ ] ys)) (n-1)

if n==0 then [ ] else y flipTake ys (n-1)) $\underset{\{HI\}}{=}$

if n==0 then [ ] else y take' ys (n-1) □

Dadas las siguientes funciones:

```
cantNodos :: AB a -> Int
{CN0} cantNodos Nil = 0
{CN1} cantNodos (Bin i r d) = 1 + (cantNodos i) + (cantNodos d)
inorder :: AB a -> [a]
{I0} inorder Nil = []
{I1} inorder (Bin i r d) = (inorder i) ++ (r:inorder d)
length :: [a] -> Int
{L0} length [] = 0
{L1} length (x:xs) = 1 + (length xs)
```

Queremos probar:

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

Inducción en t

P(t) = cantNodos t = length (inorder t)

Caso t= Nil

cantNodos Nil $\underset{\{CN0\}}{=}$ 0

???

Caso t = Bin i r d

HI = P(i) $\land$ P(d)

cantNodos (Bin i r d) $\underset{\{CN1\}}{=}$ 1 + (cantNodos i) + (cantNodos d)

$\underset{\{HI\}}{=}$ 1 + length(inorder i) + length(inorder d)

length (inorder (Bin i r d)) $\underset{\{I1\}}{=}$ = (inorder i) ++ (r:inorder d)