Ejercicio 1

I.

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

Para demostrar esto, veo la definición de intercambiar:

```
intercambiar (x,y) = (y,x)
```

Veo que intercambiar está definido solo para pares, y como sé que p es un par (de tipo (a,b)) puedo aplicar extensionalidad para pares.

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

Entonces

Ejercicio 3

VII

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

Para demostrar esta propiedad, veo las definiciones de las funciones involucradas:

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

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

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x : xs) = f x (foldr f z xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f ac [] = ac
foldl f ac (x : xs) = foldl f (f ac x) xs

flip f x y = f y x
```

Aplicando R0, reemplazamos reverse por su definición.

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

Por extensionalidad funcional, sabemos que dados f y g :: [a] -> [a] entonces f = g si y solo (∀xs :: [a] . f xs = g xs). Sabiendo esto, agregamos el parámetro que falta.

```
foldl (flip (:)) [] xs = foldr (\x rec -> rec ++ (x:[])) [] xs
```

Ahora hago inducción estructural sobre la lista xs. Siendo el predicado unario P(xs)

Caso base xs = []

```
foldl (flip (:)) [] [] = foldr (\x rec -> rec ++ (x:[])) [] []
-- Por definición de foldr y foldl, al recibir una lista vacía, devuelve una lista vacía.
[] = []
```

Son iguales el caso base se cumple.

Caso inductivo x:xs
Por hipotesis inductiva vale que:

```
∀x :: a. ∀xs :: [a]. P(xs) -> P(x:xs)
```

La hipotesis inductiva asume P(xs) como verdad. Es decir:

```
foldl (flip (:)) [] xs = foldr (\x rec -> rec ++ (x:[])) [] xs
```

Es verdad.

Queremos ver que con un elemento mas, la propiedad sigue cumpliendose.

```
foldl (flip (:)) [] x:xs = foldr (\x rec -> rec ++ (x:[])) [] x:xs
```

Por definición de foldr y foldl:

```
foldl flip (:) (flip (:) [] x) xs = (\x rec -> rec ++ (x:[]) x (foldr (\x rec -> rec +
+ (x:[])) [] xs)
```

Podemos desarrollar el lambda:

```
foldl flip (:) (flip (:) [] x) xs = (foldr (\x rec -> rec ++ (x:[])) [] xs) ++ (x:[])
```

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

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

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x : xs) = f x (foldr f z xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f ac [] = ac
foldl f ac (x : xs) = foldl f (f ac x) xs

flip f x y = f y x
```