Práctica N° 2 - Razonamiento ecuacional e inducción estructural

Para resolver esta práctica se recomienda tener a mano las soluciones de los ejercicios de la práctica 1, así como también los apuntes de las clases teóricas y prácticas de Programación Funcional.

En las demostraciones por inducción estructural, justifique **todos** los pasos: por qué axioma, por qué lema, por qué puede aplicarse la hipótesis inductiva, etc. Es importante escribir el **esquema de inducción**, planteando claramente los casos base e inductivos, e identificando la hipótesis inductiva y la tesis inductiva.

El alcance de todos los cuantificadores que se utilicen debe estar claramente definido (si no hay paréntesis, se entiende que llegan hasta el final).

Demuestre todas las propiedades auxiliares (lemas) que utilice.

Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

Extensionalidad

Ejercicio 1 ★

Sean las siguientes definiciones de funciones:

```
- intercambiar (x,y) = (y,x)
- espejar (Left x)  = Right x
  espejar (Right x) = Left x
- asociarI (x,(y,z)) = ((x,y),z)
- asociarD ((x,y),z)) = (x,(y,z))
- flip f x y = f y x
- curry f x y = f (x,y)
- uncurry f (x,y) = f x y
```

Demostrar las siguientes igualdades usando los principios de extensionalidad cuando sea necesario:

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

II. ∀ `p::(a,(b,c))` . `asociarD (asociarI p)` = `p`

III. ∀ `p::Either a b` . `espejar (espejar p)` = `p`

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

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

Ejercicio 2 ★

Demostrar las siguientes igualdades utilizando el principio de extensionalidad funcional:

I. `flip . flip` = `id`

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

III. `flip const` = `const id`

IV. ∀ `f::a->b` . ∀ `g::b->c` . ∀ `h::c->d` . `((h . g) . f)` = `(h . (g . f))`
con la definición usual de la composición: `(.) f g x = f (g x)`.

Inducción sobre listas

Ejercicio 3 ★

Considerar las siguientes funciones:

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

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

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

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

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

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

Demostrar las siguientes propiedades:

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

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

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

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

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

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

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

Nota: en adelante, siempre que se necesite usar `reverse`, se podrá utilizar cualquiera de las dos definiciones, según se considere conveniente.

Ejercicio 4

Demostrar las siguientes propiedades utilizando inducción estructural sobre listas y el principio de extensionalidad.

I. `reverse . reverse` = `id`

II. `append` = `(++)`

III. `map id` = `id`

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

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

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

Ejercicio 5 ★

Dadas las siguientes funciones:

```
       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 que `zip` = `zip'` utilizando inducción estructural y el principio de extensionalidad.

Ejercicio 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.

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

II. `Eq a =>` ∀ `xs::[a]` . ∀ `e::a` . `elem e xs` = `elem e (nub xs)`

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

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

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

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

Ejercicio 7

Dadas las definiciones usuales de `foldr` y `foldl`, demostrar las siguientes propiedades:

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

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

Otras estructuras de datos

Ejercicio 8

Demostrar que la función `potencia` definida en la práctica 1 usando `foldNat` funciona correctamente mediante inducción en el exponente.

Ejercicio 9 ★

Dadas las funciones `altura` y `cantNodos` definidas en la práctica 1 para árboles binarios, demostrar la siguiente propiedad:
∀ `x::AB a` . `altura x` ≤ `cantNodos x`

Ejercicio 10

Dada la siguiente función:

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

Y los siguientes lemas:

1. ∀ `x::Int` . ∀ `y::Int` . ∀ `z::Int` . max (min x y) (min x z) = min x (max y z)
2. ∀ `x::Int` . ∀ `y::Int` . ∀ `z::Int` . z + min x y = min (z+x) (z+y)

Demostrar las siguientes propiedades:

I. $\forall$ `t::AB a` . altura t ≥ 0

II. $\forall$ `t::AB a` . $\forall$ `n::Int` . (`n` $\geq$ `0` $\Rightarrow$ `(altura (truncar t n)` $=$ `min n (altura t)))`

Ejercicio 11

Considerar las siguientes funciones:

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

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

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

Demostrar la siguiente propiedad:
`Eq a =>` $\forall$ `e::a` . `elemAB e` $=$ `elem e . inorder`

Ejercicio 12 ★

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

```
derivado :: Num a => Polinomio a -> Polinomio a
derivado poli = case poli of
   X        -> Cte 1
   Cte _    -> Cte 0
   Suma p q -> Suma (derivado p) (derivado q)
   Prod p q -> Suma (Prod (derivado p) q) (Prod (derivado q) p)


sinConstantesNegativas :: Num a => Polinomio a -> Polinomio a
sinConstantesNegativas = foldPoli True (>=0) (&&) (&&)

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

Demostrar las siguientes propiedades:

I. `Num a =>` $\forall$ `p::Polinomio a` . $\forall$ `q::Polinomio a` . $\forall$ `r::a` . (`esRaiz r p` $\Rightarrow$ `esRaiz r (Prod p q)`)

II. `Num a =>` $\forall$ `p::Polinomio a` . $\forall$ `k::a` . $\forall$ `e::a` .
`evaluar e (derivado (Prod (Cte k) p))` $=$ `evaluar e (Prod (Cte k) (derivado p))`

III. `Num a =>` $\forall$ `p::Polinomio a`. (`sinConstantesNegativas p`$\Rightarrow$`sinConstantesNegativas (derivado p)`)

La recursión utilizada en la definición de la función `derivado` ¿es estructural, primitiva o ninguna de las dos?