FUNDAMENTOS DE COMPUTACIÓN
Repartido: Matemática de Listas

5. La inducción Primitiva para Listas

Acá otra vez nos encontramos con el problema de demostrar afirmaciones del tipo

(∀ l :: [a]) P(l)

Del mismo modo que hicimos para los Números Naturales, podemos justificar un principio de inducción para el tipo de las listas.

La **inducción primitiva** en listas nos permite demostrar ese tipo de afirmaciones, a partir de la demostración de la propiedad para la lista vacía [ ] y de demostraciones de que la operación de agregar un elemento a una lista propaga la propiedad considerada.

O sea,

Para demostrar (∀ l :: [a]) P(l) es suficiente demostrar:

1. **<u>Caso [ ]</u>: P([ ]) se cumple.**
2. **<u>Caso (:)</u>: (∀ x :: a) (∀ xs :: [a]) P(xs) ⇒ P(x:xs) se cumple.**

Observemos aquí que el **caso (:)** debe demostrarse **para todos los posibles x de tipo a** que pueden agregarse al principio de la lista xs. Esto lo hacemos explícito poniendo (∀ x :: a) en ese caso.

Del mismo modo que con los números naturales, justificamos este método como sigue:

Debemos mostrar que si se cumplen los dos componentes (**1 y 2**) del método, entonces todas las listas de tipo [a] cumplen **P**:

- Claramente **[ ]** cumple **P** por **(1)**.

- Luego, **P([z])** también se cumple para todos los posibles z de tipo a ya que por **(2)**, el constructor **(:)** propaga la propiedad (recordemos que la lista [z] es en realidad la lista z:[ ]).

- Entonces, por **(2)** se cumple que también para cualquier y::a, tendremos que **y:[z]** (o sea, cualquier lista **[y,z]** con dos elementos) cumplirá **P,** y luego también cualquier lista con tres elementos **[x,y,z]**, con cuatro elementos **[w,x,y,z]**, y así sucesivamente **todas las posibles listas que podamos obtener aplicando (:) a una lista existente cumplirán P**.

Como estas son todas las listas de tipo [a] que existen (por la definición dada del tipo [a]), esto concluye la justificación.

Resumiendo, las demostraciones de afirmaciones de la forma (∀l :: [a]) P(l) por inducción primitiva, serán de la forma:

Caso l = [ ]: P([ ]) Dem. .... **Caso l = x:xs** con x::a, xs::[a] cualesquiera. HI) P(xs) TI) P(x:xs) Dem............. ....

6. Ejemplos

Vamos a comenzar demostrando algunas propiedades de la función (++), que concatena dos listas:

> **(++) :: [a] → [a] → [a]**
> **(++) = λl1 l2 → case l1 of { [ ] → l2 ; x:xs → x: (xs ++ l2) }**

Esta función se parece mucho a la suma de naturales, de hecho, al igual que la suma, tiene un neutro, tanto a izquierda como a derecha: la lista vacía **[ ]**.
O sea, que **(∀l :: [a]) [ ] ++ l = l y l ++ [ ] = l.**

La primera demostración sale simplemente haciendo cálculos:

Lema$_{[\,]++}$: (∀l :: [a]) [ ] ++ l = l
Dem. Sea l :: [a].
La igualdad se cumple por def. (++), βx2 y case.

La segunda demostración requiere inducción, ya que para saber cuánto vale **l ++ [ ]**, debemos saber qué es **l**:

Lema$_{++[\,]}$: ($\forall$ l :: [a]) l ++ [ ] = l
Dem. Por inducción en l :: [ ]. *{la propiedad a demostrar es P =* ● **++ [ ] =** ●*}*
Caso l = [ ]: [ ] ++ [ ] =$^?$ [ ] *{P(0)}*
Dem. Se cumple por def. de (++), βx2 y case.
Caso l = x:xs con x::a, xs::[a] cualesquiera.
HI) xs ++ [ ] = xs *{P(xs)}*
TI) (x:xs) ++ [ ] =$^?$ x:xs *{($\forall$ x::a)P(x:xs)}*
Dem. Sea x::a.

    (x:xs) ++ [ ]
    = (def. (++), βx2, case)
    x : (xs ++ [ ])
    = (HI)
    x : xs.

Otra propiedad que vincula directamente **(++)** a la suma, es cuando la combinamos con la función **length**, ya que la longitud de la concatenación de dos listas es igual a la suma de sus longitudes: **($\forall$ l1 :: [a])($\forall$ l2 :: [a]) length (l1 ++ l2) = length l1 + length l2**, con length definida como:

length :: [a] → N
length = λl→ case l of { [ ] → 0 ; x:xs → S(length xs) }

La demostración se hará por inducción en l1, ya que (++) se definió por por casos en el primer argumento.

La propiedad a demostrar es: **P = ($\forall$ l2 :: [a]) length (** ● **++ l2) = lenght** ● **+ length l2**, y la misma puede demostrarse para una lista l2 genérica, por lo cual podemos tomar una l2 ::[a] cualquiera, y "deshacernos" del ($\forall$ l2 :: [a]) en los casos de la demostración.

Ahora procedemos a la demostración:

Lema$_{length++}$: ($\forall$ l1 :: [a])($\forall$ l2 :: [a]) length (l1 ++ l2) = length l1 + length l2
Dem. Por inducción en l1 :: [a].
Sea l2 :: [a] cualquiera.
Caso l = [ ]: length ([ ] ++ l2) =$^?$ lenght [ ] + length l2 *{P([ ])}*
Dem.

length ([ ] ++ l2)	lenght [ ] + length l2
= (def. (++), βx2, case)	= (def. length, β, case)
length l2	0 + length l2
	= (def. (+), βx2, case)
	length l2

Ambas expresiones son iguales por reducción a la misma expresión.

Caso l = x:xs con x::a, xs::[a] cualesquiera.
HI) length (xs ++ l2) = length xs + length l2 *{P(xs)}*
TI) length ((x:xs) ++ l2) $=^?$ length (x:xs) + length l2 *{(∀ x::a)P(x:xs)}*
Dem. Sea x::a.

length ((x:xs) ++ l2)	length (x:xs) + length l2
= (def. (++), βx2, case)	= (def. length, β, case)
length (x : (xs ++ l2)	S (length xs) + length l2
= (def. length, β, case)	= (def. (+), βx2, case)
S (length (xs ++ l2))	S (length xs + length l2)
= (HI)	
S (length xs + length l2)	

Ambas expresiones son iguales por reducción a la misma expresión.

Otra propiedad que comparte (**++**) con la suma, es que es asociativa:
(∀ l1 :: [a])(∀ l2 :: [a])(∀ l3 :: [a]) l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3.

Otra vez la demostración se hará por inducción en **l1**, ya que (**++**) se definió por por casos en el primer argumento, y la misma puede demostrarse para listas **l2** y **l3** genéricas, por lo que **P quedará sin los (∀ l2 :: [a])(∀ l3 :: [a])** en los casos.

La propiedad a demostrar es:
P = (∀ l2 :: [a])(∀ l3 :: [a]) length (● ++ l2) = lenght ● + length l2.

Ahora la demostración:

Lema$_{++asoc}$: (∀ l1 :: [a])(∀ l2 :: [a])(∀ l3 :: [a]) l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3
Dem. Por inducción en l1 :: [a].
Sean l2,l3 :: [a] cualesquiera.
Caso l= [ ]: [ ] ++ (l2 ++ l3) $=^?$ ([ ] ++ l2) ++ l3
Dem

[ ] ++ (l2 ++ l3)	([ ] ++ l2) ++ l3
= (def. (++), βx2, case)	(def. (++), βx2, case)
l2 ++ l3	l2 ++ l3

Ambas expresiones son iguales por reducción a la misma expresión.
Caso l = x:xs con x::a, xs::[a] cualesquiera.
HI) xs ++ (l2 ++ l3) = (xs ++ l2) ++ l3
TI) (x:xs) ++ (l2 ++ l3) $=^?$ ((x:xs) ++ l2) ++ l3
Dem. Sea x :: a

(x:xs) ++ (l2 ++ l3)	((x:xs) ++ l2) ++ l3
= (def. (++), βx2, case)	(def. (++), βx2, case)
x : (xs ++ (l2 ++ l3))	(x : (xs ++ l2)) ++ l3
= (HI)	(def. (++), βx2, case)
x : ((xs ++ l2) ++ l3)	x : ((xs ++ l2) ++ l3

Ambas expresiones son iguales por reducción a la misma expresión.

Ahora nos enfocaremos en la función **reverse**, que invierte una lista:

> **reverse :: [a] →[a]**
> **reverse = λl→ case l of { [ ] → [ ] ; x:xs → reverse xs ++ [x] }**

Nuestro objetivo es demostrar que cuando invertimos una lista dos veces, obtenemos la lista original, o sea: **(∀l :: [a]) reverse (reverse l) = l**.

Pero para demostrarlo, debemos primero probar algunos lemas que nos serán útiles.

El primero sale haciendo cálculos simples:

Lema$_{rev[x]}$: (∀x :: a) reverse [x] = [x]
Dem. Sea x :: a cualquiera.

reverse [x]	*{recordar que [x] = x : [ ]}*
= (def. reverse, β, case)	
reverse [ ] ++ [x]	
= (def. reverse, β, case)	
[ ] ++ [x]	
= (def. (++), βx2, case)	
[x]	

El segundo vincula **reverse** con **(++)**: concatenar dos listas y luego invertirlas es lo mismo que invertirlas primero y luego concatenarlas, pero en el orden inverso.

Lema$_{rev++}$: (∀l1 :: [a])(∀l2 :: [a]) reverse (l1 ++ l2) = reverse l2 ++ reverse l1
Dem. Por inducción en l1 :: [a].
Sea l2 :: [a] cualquiera.
Caso l = [ ]: reverse ([ ] ++ l2) =$^?$ reverse l2 + reverse [ ]
Dem.

reverse ([ ] ++ l2)	reverse l2 ++ reverse [ ]
= (def. (++), βx2, case)	= (def. reverse, β, case)
reverse l2	reverse l2 ++ [ ]
	= (Lema$_{++[\,]}$)
	reverse l2

Ambas expresiones son iguales por reducción a la misma expresión.

Caso l = x:xs con x::a, xs::[a] cualesquiera.
HI) reverse (xs ++ l2) = reverse l2 ++ reverse xs
TI) reverse (x:xs ++ l2) $=^?$ reverse l2 ++ reverse (x:xs)
Dem. Sea x :: a

reverse (x:xs ++ l2)	reverse l2 ++ reverse (x:xs)
= (def. (++), βx2, case)	= (def. reverse, β, case)
reverse (x :(xs ++ l2))	reverse l2 ++ (reverse xs ++ [x])
= (def. reverse, β, case)	= (Lema$_{++asoc}$)
reverse (xs ++ l2) ++ [x]	(reverse l2 ++ reverse xs) ++ [x]
= (HI)	
(reverse l2 ++ reverse xs) ++ [x]	

Ambas expresiones son iguales por reducción a la misma expresión.

Y ahora finalmente el resultado prometido:

Lema$_{revrev}$: (∀ l :: [a]) reverse (reverse l) = l
Dem. Por inducción en l :: [a].
Caso l = [ ]: reverse (reverse [ ]) $=^?$ [ ]
Dem.

reverse (reverse [ ])
= (def. reverse, β, case) x 2
[ ]

Caso l = x:xs con x::a, xs::[a] cualesquiera.
HI) reverse (reverse xs) = xs
TI) reverse (reverse (x:xs)) $=^?$ x : xs
Dem. Sea x :: a

reverse (reverse (x : xs))
= (def. reverse, β, case)
reverse (reverse xs ++ [x])
= (Lema$_{rev++}$)
reverse [x] ++ reverse (reverse xs)
= (Lema$_{[x]}$ e HI)
[x] ++ xs
= (def. (++), βx2, case)
x: ([ ] ++ xs)
= (def. (++), βx2, case)
x: xs

Finalmente, demostraremos algunas propiedades de la función **filter**, que recibe un predicado y una lista y devuelve la lista con los elementos para los cuales el predicado es verdadero:

filter :: (a → Bool) → [a] → [a]
filter = λp l → case l of { [ ] → [ ] ; x:xs → case p x of { False → filter p xs;
True → x: filter p xs } }

Primero demostraremos que aplicar dos veces **filter** a una misma lista con la misma propiedad es lo mismo que aplicarla una vez sola:

Lema$_{\text{filter2}}$: (∀p :: a → Bool)(∀l :: [a]) filter p (filter p l) = filter p l
Dem. Por inducción en l :: [a]. Sea p :: a → Bool.
Caso l = [ ]: filter p (filter p [ ]) =$^?$ filter p [ ]
Dem. filter p (filter p [ ])
= (def. filter (de adentro), βx2, case)
filter p [ ]
Caso l = x:xs con x::a, xs::[a] cualesquiera.
HI) filter p (filter p xs) = filter p xs
TI) filter p (filter p (x:xs)) =$^?$ filter p (x : xs)
Dem. Sea x :: a.
{observemos acá que no es posible saber el valor de filter p (x:xs), a menos que sepamos cuánto vale p x, luego deberemos hacer casos en ese booleano}
Por casos en p x :: Bool
Caso p x = False

filter p (filter p (x:xs))	filter p (x:xs)
= (def. filter, βx2, casex2)	= (def. filter, βx2, casex2)
filter p (filter p xs)	filter p xs
= (HI)	
filter p xs	

Ambas expresiones son iguales por reducción a la misma expresión.
Caso p x = True

filter p (filter p (x:xs))	filter p (x:xs)
= (def. filter, βx2, casex2)	= (def. filter, βx2, casex2)
filter p (x: filter p xs)	x : filter p xs
= (def. filter, βx2, casex2)	
x : filter p (filter p xs)	
= (HI)	
x : filter p xs	

Ambas expresiones son iguales por reducción a la misma expresión.

Ahora demostraremos la siguiente desigualdad:

Lema$_{filter\leq}$: $(\forall$ p :: a → Bool)($\forall$ l :: [a]) length (filter p l) ≤ length l

Dem. Por inducción en l :: [a]. Sea p :: a → Bool.

Caso l = [ ]: length (filter p [ ]) ≤$^?$ length [ ]

Dem. length (filter p [ ])
= (def. filter, βx2, case)
length [ ]
= (def. length, β, case)
0
≤ (Lema$_{0\leq}$) *{Lema$_{0\leq}$: $(\forall n :: N)\ 0 \leq n$}*
length [ ]

Caso l = x:xs con x::a, xs::[a] cualesquiera.

HI) length (filter p xs) ≤ length xs

TI) length (filter p (x:xs)) ≤$^?$ length (x : xs)

Dem. Sea x :: a

Por casos en p x :: Bool

Caso p x = False

length (filter p (x:xs))
= (def. filter, βx2, casex2)
length (filter p xs)
≤ (HI)
length xs
≤ (Lema$_{\leq S}$) *{Lema$_{\leq S}$: $(\forall n :: N)\ n \leq S\ n$}*
S (length xs)
= (def. length, β, case)
length (x : xs)

Caso p x = True

length (filter p (x:xs))
= (def. filter, βx2, casex2)
length (x: filter p xs)
= (def. length, β, case)
S (length (filter p xs))
≤ (HI + Lema$_{\leq SS}$) *Lema$_{\leq SS}$: $(\forall m,n::N)\ m \leq n \Rightarrow S\ m \leq S\ n$*
S (length xs)
= (def. length, β, case)
length (x : xs).