FUNDAMENTOS DE COMPUTACIÓN
Repartido: Listas

1. El tipo de las Listas

Dado cualquier tipo t, queremos definir el tipo [t], cuyos objetos son las listas (secuencias) de objetos de tipo t.

Antes de definirlo, daremos algunos ejemplos:

- [True, False, False, True] :: [Bool]
- [0, S 0, S (S 0)] :: [N]
- [par, (>0), pos] :: [N → Bool]
- [[True], [False,False], [ ]] :: [[Bool]]

La tercera línea muestra que es posible formar listas de funciones, y la cuarta es una lista de listas.

Una lista puede contener cualquier cantidad de elementos (también llamados sus miembros).
Esto incluye el caso de la lista vacía la cual, como es de esperar, se escribe [ ].

En Haskell todos los miembros de una lista deben ser del mismo tipo.

Además, es importante diferenciar la noción de lista de la noción de conjunto.
Esto queda claro en las siguientes características:

- El orden de los elementos es importante, o sea, las listas [True, False, False] y [False, True, False] son listas distintas de tipo [Bool].
- La cantidad de veces que un elemento aparece en una lista también importa, o sea, la lista [0, 0, S 0] y la lista [0, S 0] son listas distintas de tipo [N].

La notación con corchetes: $[a_1, a_2, ..., a_n]$ se utiliza en Haskell para formar listas concretas. Pero, en los hechos, esta notación es una abreviatura de otra notacióon más básica, que permite formar las listas de cualquier tipo de manera sistemática y **recursiva**:

Así, el tipo de las listas se define como sigue:

Por definición, una lista es:
- o bien vacía (es decir, la lista [ ], también llamada ***nil***)
- o bien, formada por agregar un elemento a una lista previamente formada.
La operación de agregar se nota (**:**) en Haskell y se pronuncia ***cons***.

Ahora estamos en condiciones de dar una **definición inductiva del tipo infinito [t]**:

- **La semilla es la lista vacía [ ]**
- **El generador es la función (:) :: t → [t] → [t]**

Las reglas que definen al tipo **[t]** son entonces:

$$\frac{\text{t tipo}}{\text{[t] tipo}}$$

y sus valores constructores anteriormente mencionados:

$$\frac{}{\text{[ ] :: [t]}} \qquad \frac{\text{a :: t} \quad \text{l :: [t]}}{\text{a : l :: [t]}}$$

Así pues, si traducimos la lista **[True, False, False]** a la notación "*cons-nil*", resulta que se obtiene por agregar el elemento **True** a la lista **[False, False]**, lo cual nos da como notación correcta: **True : [False, False]**.
En este caso, como en el de toda lista no vacıa, el primer elemento se denomina la cabeza (en inglés ***head***) de la lista, mientras que la lista que sigue a continuación se denomina su resto o cola (en inglés: ***tail***).

Si continuamos descomponiendo la cola de la lista, obtenemos que la lista original puede escribirse también como **True : (False : [False])**, y finalmente, una lista de un único elemento, tal como **[False]** no es otra cosa que agregar ese elemento a la lista vacía **[ ]**.
Es decir que la lista **[True, False, False]** corresponde a la expresión Haskell
True : (False : (False : [ ])) :: [Bool].

En Haskell los paréntesis que aparecen en esta última expresión no son necesarios ya que el operador **:** asocia a derecha, por lo que podemos escribir directamente:
True : False : False : [ ].

En definitiva, una lista **[a_1, a_2, ..., a_n]** se corresponde con la expresión **a_1 : a_2 : ... a_n : [ ].**

Esta notación, junto con la convención de asociación a derecha recién mencionada, nos da una descripción del proceso de creación o construcción (recursiva) de la lista:

- Comenzamos por la lista más simple posible y siempre disponible, es decir **[ ].**

- Luego vamos agregando cada vez un nuevo elemento a la lista que ya hemos formado utilizando el operador **:**.

Las listas crecen entonces hacia la izquierda o, como también puede verse, agregando elementos adelante.

Definimos en Haskell el tipo de las listas del siguiente modo:

data [t] where { [ ] :: [t] ; (:) :: t → [t] → [t] }

En esta declaración hay algunos detalles a explicar:

- El nombre de tipo introducido es **[t]**, donde **t** es un **parámetro de tipo**.
Es decir, estamos diciendo: para todo tipo t tenemos el tipo inductivo (data) **[t]**, leído "**listas de elementos de tipo t**". O sea que, de hecho, estamos introduciendo una cantidad infinita de tipos mediante una única declaración (observar que el parámetro t se escribe con minúscula, indicando que no es un tipo específico con Bool o N).

- El primer constructor es la lista vacía **[ ]**.

- Luego viene otro constructor, que es **(:)**. A diferencia del generador **S** de los naturales, éste no forma un valor de por sí, sino que requiere dos parámetros:
  - La cabeza de la nueva lista, o sea el elemento a agregar (obviamente de tipo **t**)
  - La lista a alargar con ese nuevo elemento (obviamente de tipo **[t]**).

Asociado al tipo definido tenemos la expresión **case** correspondiente, cuya regla será:

$$\frac{e :: [\,t\,] \qquad e1 :: s \qquad e2 :: s\ [x::t,\ xs::[t]]}{\textbf{case } e \textbf{ of } \{\,[\,] \rightarrow e1\ ;\ x{:}xs \rightarrow e2\,\} :: s}$$

Observemos que en la segunda rama del case estamos considerando que el discriminante (**e**) resulte ser una lista construída agregando un elemento **x** a una lista **xs** (o sea, **x:xs**). Indicamos que tanto **x** como **xs** pueden aparecer en **e2** en la tercer premisa de la regla, donde escribimos el juicio **e2 :: s [x::t, xs::[t]].**

Por supuesto, los nombres utilizados para estos parámetros son arbitrarios y bien podrían haber sido cabeza y cola, h y t, o cualesquiera otros. Pero la elección x y xs se ha impuesto en la comunidad de programadores Haskell al punto de constituir prácticamente un estándar. Viene del hecho de que hace notar muy gráficamente que x es un elemento arbitrario y que el resto son "otros xs", o sea, una lista del mismo tipo de cosas.

Las igualdades serán las siguientes para este case:

case [ ] of { [ ] → e1 ; x:xs → e2 } = e1
case a:l of { [ ] → e1 ; x:xs → e2 } = e2 [x:=a , xs:= l]

Observar que el resultado de la expresión case cuando el discriminante es una lista (no vacía) de la forma **a:l** es **e2**, donde debemos instanciar **x** con **a** y **xs** con **l**, lo cual se define utilizando la sustitución **[x:=a , xs:= l].**

2. Recursión en Listas

Veamos algunos ejemplos de funciones sobre listas.

Comenzamos con un ejemplo sencillo, la función **null** que verifica si una lista está vacía:

null :: [a] → Bool

(en general en Haskell los parámetros de tipo se escriben con las primeras letras del alfabeto: a, b, c...)

Esta es una función sencilla, que devuelve **True** para el caso **[ ]** y **False** para el caso **x:xs**:

null = λl→ case l of { [ ] → True ; x:xs → False}

La siguiente función **sum :: [N] → N,** suma todos los elementos de una lista de naturales.

Planteemos su definición utilizando la expresión **case**:

sum = λl→ case l of { [ ] → ?1 ; x:xs → ?2 }

Donde:

- ?1 corresponde al caso de la lista vacía.
La incógnita **?1** debe ser sustituida por el resultado de sumar todos los elementos de la lista vacía. Una breve reflexión nos indica que ese resultado es **0**.

- En **?2** estamos considerando el caso de una lista formada por el constructor (**:**), es decir, se trata de **una lista no vacía**, cuyo primer elemento es **x** y su resto (o cola) es **la lista xs**.
Ahora la cuestión es resolver la incógnita **?2** y así terminar de definir la función.
En este punto es que la **recursión** viene en nuestra ayuda:
¿Cuál es el resultado de sumar todos los elementos de la lista **x:xs**?
Bueno, basta con sumar **x** a la suma de los elementos de la lista **xs**, y esta última suma la podemos obtener por medio de la llamada recursiva: **sum xs.**
Entonces **?2 = x + sum xs**.

Luego, la función **sum** queda definida como:

sum :: [N] → N
sum= λl→ case l of { [ ] → 0 ; x:xs → x + sum xs }

Al igual que con los Naturales, podemos plantear un **esquema de recursión estructural en listas** que nos da un método de programación de funciones:

Para definir una función f :: [a] → t, definimos:

f = λl→ case l of { [ ] → b ; x:xs → r }

donde:

- **b** es el resultado de **f** en **[ ]**

- **r** es el paso recursivo **r**, es decir el resultado de **f** en **x:xs**, para lo cual se puede utilizar el resultado de **f xs**.

3. Ejemplos

Terminamos este repartido definiendo algunas funciones muy útiles del Preludio de Haskell:

El primer ejemplo es la función l**ength :: [t] → N**, que calcula la longitud (cantidad de elementos) de una lista.

length = λl→ case l of { [ ] → ?1 ; x:xs → ?2 }

Otra vez, debemos determinar las incógnitas **?1** y **?2**.

Claramente **?1** = 0.

Para definir **?2** podemos utilizar la llamada recursiva **length xs**.
Con esto, podemos definir **?2** = **S (length xs)**.

La función queda definida como:

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

La función **elem :: Eq a => a → [a] → Bool**, verifica si un elemento pertenece a una lista.

Observemos el tipo de **elem**: el mismo empieza con **Eq a =>**.
Esta es la forma de indicarle a Haskell que para poder definir la función **elem**, el tipo **a debe ser instancia de la clase Eq** . Así que podemos usar == y /= entre elementos de tipo a en elem (recordemos que Eq es la clase de tipos que tienen definidas las funciones == y /=).

Ahora podemos definir:

elem = λe l → case l of { [ ] → False ; x:xs → x == e || elem e xs }

La siguiente función es **map**, que recibe una función de tipo **(a → b)** y la aplica a cada uno de los elementos de una lista de tipo **[a]**, devolviendo una lista de tipo **[b]**.
El tipo de **map** será entonces:

```
map ::  (a → b) → [a] → [b]
```

y la definición será

```
map = λf l → case l of { [ ] → [ ]  ; x:xs →  f x : map f xs }
```

Observemos que **map** recibe como parámetro una función.
Ejemplos de su aplicación son:

```
map (>0) [0, S 0, S(S 0), S(S(S 0))] = [False, True, True,True]

map length [ [True] , [False,False] , [ ] ] = [S 0, S(S 0), 0]
```

La función **filter** recibe un predicado y una lista, y devuelve otra lista con aquellos elementos para los cuales el predicado se cumple.

```
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 } }
```

La función **(++)** concatena dos listas del mismo tipo, poniendo la segunda a continuación de la primera.

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

Finalmente, la función **reverse** da vuelta una lista, invirtiendo el orden de sus elementos:

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

Notar que en caso x:xs, debemos poner a x al final de la lista xs invertida. Para hacer eso, no podemos utilizar el constructor (:), ya que éste recibe primero un elemento y después una lista (y no al revés). Por lo tanto, debemos utilizar la función ++ que permite concatenar dos listas, para lo cual, creamos la lista [x] que contiene sólo al elemento x, y lo colocamos detrás de reverse xs.