Introducción a los Algoritmos - 2do. cuatrimestre 2012

Guía 2: Listas, recursión e inducción

Docentes: Walter Alini, Luciana Benotti, Marcos Gómez y Gisela Rossi

En esta guía comenzaremos a trabajar con listas. Para familiarizarnos con ellas, los primeros ejercicios son de evaluación y tipado. Los siguientes serán sobre definición de funciones recursivas y no recursivas, y finalmente, inducción.

Listas

A partir de esta sección extendemos el lenguaje de nuestras expresiones con el tipo de las listas. Una lista (o secuencia) es una colección ordenada de valores, que deben ser todos del mismo tipo. Denotamos a la lista vacía con $[\,]$*. El operador* $\triangleright$ *(llamado "pegar" y notado* `:` *en Haskell) es fundamental (se lo denomina constructor) ya que permite construir listas arbitrarias a partir de la lista vacía.* $\triangleright$ *toma un elemento* x *(a izquierda) y una lista* xs *y devuelve una lista con primer elemento* x *seguido de los elementos de* xs*. Por ejemplo* $3 \triangleright [\,] = [3]$*, y* $1 \triangleright [2,3] = [1,2,3]$*. Para denotar listas no vacías utilizamos expresiones de la forma* $[x, y, \ldots, z]$*, que son abreviaciones de* $x \triangleright (y \triangleright \ldots \triangleright (z \triangleright [\,]))$*. Como el operador* $\triangleright$ *es asociativo a derecha, es lo mismo escribir* $x \triangleright (y \triangleright \ldots \triangleright (z \triangleright [\,])$ *que* $x \triangleright y \triangleright \ldots \triangleright z \triangleright [\,]$*. Otros operadores sobre listas son los siguientes:*

- $\#$*, llamado cardinal, toma una lista* xs *y devuelve su cantidad de elementos. Ej:* $\#[1,2,0,5] = 4$*. En Haskell* $\#xs$ *se escribe:* `length xs`*.*
- $.$ *toma una lista* xs *(a izquierda) y un natural* n *que indica una posición, y devuelve el elemento de la lista que se encuentra en la posición* n *(contando a partir de la posición 0). Ej:* $[1,3,3,6,2,3,4,5].4 = 2$*. Este operador, llamado índice, asocia a izquierda, por lo tanto* $xs.n.m$ *se interpreta como* $(xs.n).m$*. En Haskell* $xs.n$ *se escribe:* `xs !! n`*.*
- $\uparrow$ *toma una lista* xs *(a izquierda) y un natural* n *que indica una cantidad, y devuelve la sublista con los primeros* n *elementos de* xs*. Ej:* $[1,2,3,4,5,6] \uparrow 2 = [1,2]$*. Este operador, llamado tomar, asocia a izquierda, por lo tanto* $xs \uparrow n \uparrow m$ *se interpreta como* $(xs \uparrow n) \uparrow m$*. En Haskell* $xs \uparrow n$ *se escribe:* `take n xs`*.*
- $\downarrow$ *toma una lista* xs *(a izquierda) y un natural* n *que indica una cantidad, y devuelve la sublista sin los primeros* n *elementos de* xs*. Ej:* $[1,2,3,4,5,6] \downarrow 2 = [3,4,5,6]$*. Este operador, llamado tirar, se comporta igual al anterior, interpretando* $xs \downarrow n \downarrow m$ *como* $(xs \downarrow n) \downarrow m$*. En Haskell* $xs \downarrow n$ *se escribe:* `drop n xs`*.*
- $+\!\!+$ *toma una lista* xs *(a izquierda) y otra* ys*, y devuelve la lista con todos los elementos de* xs *seguidos de los elementos de* ys*. Ej:* $[1,2,4] +\!\!+ [1,0,7] = [1,2,4,1,0,7]$*. Este operador, llamado concatenación, es asociativo por lo que podemos escribir sin ambigüedad expresiones sin paréntesis, como* $xs +\!\!+ ys +\!\!+ zs$*. En Haskell* $xs +\!\!+ ys$ *se escribe:* `xs ++ ys`*.*
- $\triangleleft$ *toma una lista* xs *(a izquierda) y un elemento y devuelve una lista con todos. los elemento de* xs *seguidos por* x *como último elemento. Ej:* $[1,2] \triangleleft 3 = [1,2,3]$*. Este operador, llamado "pegar a izquierda", es asociativo a izquierda, luego es lo mismo* $([\,] \triangleleft z) \ldots \triangleleft y) \triangleleft x$ *que* $[\,] \triangleleft z \ldots \triangleleft y \triangleleft x$*. En Haskell* $xs \triangleleft x$ *se escribe:* `xs:[x]`*.*

Existen además dos funciones fundamentales sobre listas que listamos a continuación. Notar que como son funciones se hace explícita la aplicación de función con el símbolo (.).

- `head`*, llamada cabeza, toma una lista* xs *y devuelve su primer elemento. Ej:* `head [1,2,3]` $= 1$*.*
- `tail`*, llamada cola, toma una lista* xs *y devuelve la sublista que resulta de eliminar el primer elemento. Ej:* `tail [1,2,3]` *= [2,3]*

La aplicación de función asocia a izquierda, por lo tanto en general es necesario utilizar paréntesis para que la expresión quede bien tipada. Si se quiere escribir la expresión `tail (tail xs)` *no se pueden eliminar los paréntesis, puesto que* `tail tail xs` *(que se interpreta como* `(tail tail) xs` *no tiene sentido.*

A continuación, listamos los niveles de precedencia de estos operadores. Los que están más arriba tienen mayor precedencia. Cuando hay más de un operador en un nivel de precedencia, es necesario poner paréntesis para evitar la ambigüedad. Por ejemplo $x \triangleright xs \uparrow n$ *se interpreta como* $x \triangleright (xs \uparrow n)$*.*

., #, head y tail	*índice, cardinal, head y tail*
$\uparrow, \downarrow$	*tomar y tirar elementos de una lista*
$\triangleright, \triangleleft$	*pegar a derecha y pegar a izquierda*
$+\!\!+$	*concatenar dos listas*

Te recomendamos leer las secciones 7.2, 7.4, 7.5, 7.6, 7.8, 7.10, del libro para profundizar en estos conceptos.

El objetivo de los siguientes ejercicios es familiarizarse con el tipo de listas y extender el método para justificar el tipado de expresiones, considerando expresiones más complejas que las que veníamos trabajando.

1. Utilizá las definiciones intuitivas de los operadores de listas para evaluar las siguientes expresiones. Subrayá la subexpresión resuelta en cada paso justificado. Luego usá un intérprete de `haskell` para verificar los resultados. Por ejemplo:

$$\begin{array}{l} \quad \underline{[23,45,6].(\mathit{head}\ [1,2,3,10,34])} \\ = \{\ \text{def. de } \mathit{head}\ \} \\ \quad \underline{[23,45,6].1} \\ = \{\ \text{def. de .}\ \} \\ \quad 45 \end{array}$$

En `haskell` los distintos operadores se pueden escribir así:

$head.xs$	`head xs`
$tail.xs$	`tail xs`
$x \triangleright xs$	`x:xs`
$xs \triangleleft x$	`xs ++ [x]`
$xs \uparrow n$	`take n xs`
$xs \downarrow n$	`drop n xs`
$xs +\!\!+ ys$	`xs ++ ys`
$\#xs$	`length xs`
$xs.n$	`xs !! n`

   a) $\#[5,6,7]$
   b) $[5,3,57].1$
   c) $[0,11,2,5] \triangleright [\,]$
   d) $[5,6,7] \uparrow 2$
   e) $[5,6,7] \downarrow 2$
   f) $\mathit{head}\ (0 \triangleright [1,2,3])$

2. Teniendo en cuenta la definición intuitiva de los operadores de listas de la introducción a esta sección, escribí las reglas de tipado de cada uno de ellos. Por ejemplo, el operador *head* toma una lista y devuelve el primer elemento de ella. La lista puede contener elementos de cualquier tipo (todos del mismo), ya sean números, valores booleanos, otras listas, etc. Para denotar esta situación utilizamos variables (en mayúsculas). Entonces podemos decir que el operador *head* toma una lista de tipo $[A]$, donde la variable A representa cualquier tipo ($Num, Bool, [Num], \ldots$) y devuelve un elemento de esa lista, por lo tanto debe ser un elemento de tipo A. Esto lo escribimos en *notación funcional* (izq.) o en *notación de árbol* (der.):

$$\mathit{head} : [A] \to A \qquad\qquad \frac{\mathit{head}\ [A]}{A}$$

3. Decidí si las siguientes expresiones están bien escritas, agregando paréntesis para hacer explícita la precedencia y la asociatividad, y justificando con el árbol del tipado correspondiente. Usá un intérprete de `haskell` para verificar los resultados.

   a) $-45 \triangleright [1,2,3]$
   b) $([1,2] +\!\!+ [3,4]) \triangleleft 5$
   c) $0 \triangleleft [1,2,3]$
   d) $[\,] \triangleright [\,]$
   e) $([1] +\!\!+ [2]) \triangleleft [3]$
   f) $[1, 5, \mathit{False}]$
   g) $\mathit{head}\ [[5]]$
   h) $\mathit{head}\ [\mathit{True}, \mathit{False}] +\!\!+ [\mathit{False}]$

Funciones recursivas

*Una **función recursiva** es una función tal que en su definición puede aparecer su propio nombre. Una buena pregunta sería ¿Cómo lograr que no sea una definición circular? La clave está en el principio de inducción: en primer lugar hay que definir la función para el (los) caso(s) más "pequeño(s)", que llamaremos **caso base** y luego definir el caso general en términos de algo más "chico", que llamaremos **caso inductivo** . El caso base no debe aparecer el nombre de la función que se está definiendo. El caso inductivo es donde aparece el nombre de la función que se está definiendo, y debe garantizarse que el (los) argumento(s) al cual se aplica en la definición es más "chico" (para alguna definición de más chico) que el valor para la que se está definiendo.*

4. Una función de **filter** es aquella que dada una lista devuelve otra lista cuyos elementos son los elementos de la primera que cumplan una determinada condición, en el mismo orden y con las mismas repeticiones (si las hubiere). Por ejemplo: $soloPares : [Int] \to [Int]$ devuelve aquellos elementos de la lista que son pares.

   Definí recursivamente las siguientes funciones filter.

   a) *soloPares* : $[Int] \to [Int]$, que dada una lista de enteros xs devuelve una lista sólo con los números pares contenidos en xs, en el mismo orden y con las mismas repeticiones (si las hubiera).
      Por ejemplo: *soloPares* $[3, 0, -2, 12] = [0, -2]$

   b) *mayoresQue10* : $[Int] \to [Int]$, que dada una lista de enteros xs devuelve una lista sólo con los números mayores que 10 contenidos en xs,
      Por ejemplo: *mayoresQue10* $[3, 0, -2, 12] = [12]$

   c) *mayoresQue* : $Int \to [Int] \to [Int]$, que dado un entero n y una lista de enteros xs devuelve una lista sólo con los números mayores que n contenidos en xs,
      Por ejemplo: *mayoresQue* $2\ [3, 0, -2, 12] = [3, 12]$

   Preguntas:

   a) ¿Se te ocurre algún otro ejemplo de una función de este tipo?

   b) ¿Cómo describirías una regla general para este tipo de funciones? **Ayuda:** Considerá que la condición que se testea se representa con una condición booleana p.

5. Una función de **map** es aquella que dada una lista devuelve otra lista cuyos elementos son los que se obtienen de aplicar una función a cada elemento de la primera en el mismo orden y con las mismas repeticiones (si las hubiere). Por ejemplo: $duplica : [Int] \to [Int]$ devuelve cada elemento de la lista multiplicado por 2.

   Definí recursivamente las siguientes funciones de map.

   a) *duplica* : $[Int] \to [Int]$, que dada una lista de enteros duplica cada uno de sus elementos.
      Por ejemplo: *duplica* $[3, 0, -2] = [6, 0, -4]$

   b) *multiplica* : $Int \to [Int] \to [Int]$, que dado un número n y una lista, multiplica cada uno de los elementos por n.
      Por ejemplo: *multiplica* $3\ [3, 0, -2] = [9, 0, -6]$

   Preguntas:

   a) ¿Se te ocurre algún otro ejemplo de una función de este tipo?

   b) ¿Cómo describirías una regla general para este tipo de funciones? **Ayuda:** considerá que la función a aplicar a cada elemento se llama f.

6. Una función de **fold** es aquella que dada una lista devuelve un valor resultante de combinar los elementos de la lista. Por ejemplo: $sum : [Int] \to Int$ devuelve la sumatoria de los elementos de la lista.

   Definí recursivamente las siguientes funciones fold.

   a) *todosMenores10* : $[Int] \to Bool$, que dada una lista devuelve *True* si ésta consiste sólo de números menores que 10.

   b) *hay0* : $[Int] \to Bool$, que dada una lista decide si existe algún 0 en ella.

   c) *sum* : $[Int] \to Int$, que dada una lista devuelve la suma de todos sus elementos.

   Preguntas:

   a) ¿Se te ocurre algún otro ejemplo de una función de este tipo?

   b) ¿Cómo describirías una regla general para este tipo de funciones?

Funciones generales

En las funciones vistas hasta el momento hemos trabajado con tipos específicos de funciones sobre listas. A continuación trabajaremos con funciones recursivas en términos generales. También analizaremos el tipo de las funciones.

7. Definí recursivamente los operadores básicos de listas: $\#, \, . \, , \triangleleft, \uparrow, \downarrow, +\!\!+$. Para los operadores $\uparrow, \downarrow$ y . deberás hacer recursión en ambos parámetros, en el parámetro lista y en el parámetro numérico.

8. (i) Definí funciones por recursión para cada una de las siguientes descripciones. (ii) Evaluá los ejemplos manualmente (iii) Identificá si las funciones son de algún tipo ya conocido (filter, map, fold). (iv) Programálas en `haskell` y verificá los resultados obtenidos.

   a) *maximo* : $[Int] \to Int$, que calcula el máximo elemento de una lista de enteros.
   Por ejemplo: *maximo* $[2, 5, 1, 7, 3] = 7$
   Ayuda: Ir tomando de a dos elementos de la lista y 'quedarse' con el mayor.

   b) *sumaPares* : $[(Num, Num)] \to Num$, que dada una lista de pares de números, devuelve la sumatoria de todos los números de todos los pares.
   Por ejemplo: $sumaPares \; [(1, 2)(7, 8)(11, 0)] = 29$

   c) *todos0y1* : $[Int] \to Bool$, que dada una lista devuelve *True* si ésta consiste sólo de 0s y 1s.
   Por ejemplo: *todos0y1* $[1, 0, 1, 2, 0, 1] =$ *False*, *todos0y1*.$[1, 0, 1, 0, 0, 1] =$ *True*

   d) *quitar0s* : $[Int] \to [Int]$ que dada una lista de enteros devuelve la lista pero quitando todos los ceros.
   Por ejemplo *quitar0s* $[2, 0, 3, 4] = [2, 3, 4]$

   e) *ultimo* : $[A] \to A$, que devuelve el último elemento de una lista.
   Por ejemplo: *ultimo* $[10, 5, 3, 1] = 1$

   f) *listasIguales* : $[A] \to [A] \to Bool$, que determina si dos listas son iguales, es decir, contienen los mismos elementos en las mismas posiciones respectivamente.
   Por ejemplo: *listasIguales* $[1, 2, 3]$ $[1, 2, 3] =$ *True*, *listasIguales* $[1, 2, 3, 4]$ $[1, 3, 2, 4] =$ *False*

Inducción

Una técnica poderosa para demostrar propiedades sobre un dominio inductivo, como son los naturales o las listas, es usar el ***principio de inducción****. La idea que rige este principio consiste en demostrar dos cosas. Por un lado verificar que la propiedad se satisface para los elementos "más chicos" del dominio (por ejemplo, la lista* $[\,]$*). Por otro lado demostrar para un elemento arbitrario del dominio (por ejemplo, la lista* $x \triangleright xs$*) que si suponemos que la propiedad es cierta para todos los elementos más chicos que él (por ejemplo* xs*), entonces la propiedad también es satisfecha por ese elemento. Dado que todo elemento de un dominio inductivo puede ser "construido" a partir de elementos más simples, este procedimiento demuestra que la propiedad es satisfecha por todos los elementos del dominio, y por lo tanto es válida.*

a) Demostrá por inducción las siguientes propiedades. **Ayuda:** Recordá la definición de cada uno de los operadores implicados en cada expresión.

1) $xs +\!\!+ [\,] = xs$ (la lista vacía es el elemento neutro de la concatenación)
2) $\#xs \geqslant 0$
3) $xs +\!\!+ (ys +\!\!+ zs) = (xs +\!\!+ ys) +\!\!+ zs$ (la concatenación es asociativa)
4) $(xs +\!\!+ ys) \uparrow \#xs = xs$
5) $(xs +\!\!+ ys) \downarrow \#xs = ys$
6) $xs +\!\!+ (y \triangleright ys) = (xs \triangleleft y) +\!\!+ ys$
7) $xs +\!\!+ (ys \triangleleft y) = (xs +\!\!+ ys) \triangleleft y$

b) Considerando la función $sum : [Num] \to Num$ que toma una lista de números y devuelve la suma de ellos, definí *sum* y demostrá que:

$$sum\ (xs +\!\!+ ys) = sum\ xs + sum\ ys$$

c) Considerando la función $repetir : Nat \to Num \to [Num]$, que construye una lista de un mismo número repetido cierta cantidad de veces, definida recursivamente como:

$$\begin{aligned} repetir\ 0\ x &\doteq [\,] \\ repetir\ (n+1)\ x &\doteq x \triangleright repetir\ n\ x \end{aligned}$$

demostrá que $\#repetir\ n\ x = n$.

d) Considerando las funciones *sum* y *duplica* de los ejercicios 6*c* y **??**, respectivamente, demostrá que:

$$sum\ (duplica\ xs) = 2 * sum\ xs$$

e) Considerando la función $concat : [[A]] \to [A]$ que toma una lista de listas y devuelve la concatenación de todas ellas, definida recursivamente como:

$$\begin{aligned} concat\ [\,] &\doteq [\,] \\ concat\ (xs \triangleright xss) &\doteq xs +\!\!+ concat\ xss \end{aligned}$$

demostrá que $concat\ (xss +\!\!+ yss) = concat\ xss +\!\!+ concat\ yss$

f) Considerando la función $rev : [A] \to [A]$ que toma una lista y devuelve una lista con los mismos elementos pero en orden inverso, definida recursivamente como:

$$\begin{aligned} rev\ [\,] &\doteq [\,] \\ rev\ (x \triangleright xs) &\doteq rev\ xs \triangleleft x \end{aligned}$$

demostrá que $rev\ (xs +\!\!+ ys) = rev\ ys +\!\!+ rev\ xs$