Paradigmas de Programación

Práctica 9

Ejercicios:

1. En un examen en el que se pedía una implementación recursiva terminal de la función `concat` del módulo `List`, algunos alumnos hicieron la siguiente definición:

```
let concat l =
  let rec aux acc = function
      [] -> acc
    | h::t -> aux (List.append acc h) t
  in aux [] l
```

   Esta definición puede considerarse terminal porque, a partir de la versión 5.1 de Ocaml, la implementación de `List.append` es recursiva terminal. Por otra parte, dicha definición es totalmente equivalente a esta otra:

```
let concat l =
  List.fold_left (fun acc h -> List.append acc h) [] l
```

   Sin embargo, si bien la lógica de ambas definiciones es impecable, la aplicación de las funciones definidas de esta forma resulta muy ineficiente si la lista a la que se aplican es larga (y las listas que la componen no son vacías). Compruebe que los valores que devuelven al aplicarlas son los mismos que devuelve la función `List.concat`, pero que, si las listas son largas, pueden tardar mucho más tiempo. Puede probar a aplicarlas, por ejemplo, a la lista definida a continuación (que tiene 50.000 elementos):

```
let l = List.init 50_000 (fun i -> [i])
```

   Averigüe por qué sucede esto, explíquelo (como comentario) en un archivo con nombre `concat.ml` e incluya en el mismo archivo dos nuevas definiciones de `concat` que no presenten este problema:

   - Una con nombre `concat'` y escrita utilizando `List.fold_right`. Explique (en un comentario) por qué es más eficiente que la versión escrita con `List.fold_left`.
   - Y otra con nombre `concat''` que sea recursiva terminal. Puede seguir suponiendo que la función `List.append` es terminal, aunque seguramente lo más acertado sea no utilizarla.

2. Añada al archivo `concat.ml` una función `sublists : 'a list -> 'a list list` que, dada una lista, devuelva la lista de todas sus sublistas. Por ejemplo:

```
# sublists [1;2;3];;
- : int list list =
  [[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]]
```

   El orden de las sublistas dentro de la lista resultado podría ser diferente, pero el orden de los elementos dentro de cada sublista debe respetar el orden relativo de dichos elementos dentro de la lista original.

3. Escriba en un archivo de nombre `tail.ml` versiones terminales de las siguientes funciones:

```
let rec front = function
    [] -> raise (Failure "front")
  | h::[] -> []
  | h::t -> h :: front t

let rec compress = function
    h1::h2::t ->
      if h1 = h2 then compress (h2::t)
      else h1 :: compress (h2::t)
  | l -> l

let fold_right =
  List.fold_right
```

4. Ejercicio opcional. Identifique cuáles de las funciones del ejercicio anterior pueden ser implementadas con la transformación *"Tail Modulo Constructor"* y escriba en un archivo `tail2.ml` nuevas definiciones para ellas utilizando esta funcionalidad[1].

   Compare estas nuevas definiciones con las definiciones del ejercicio anterior, estudiando su rendimiento sobre listas largas. Escriba sus conclusiones como comentario en el mismo archivo.

[1] Véase `https://ocaml.org/manual/5.4/tail_mod_cons.html`