Paradigmas de Programación

Práctica 10

Nota Importante:

Realice las implementaciones de los ejercicios 1 y 2 de esta práctica en los ficheros `qsort.ml` y `msort2.ml`, respectivamente.

Cuando se solicite la entrega de esta práctica, cada alumno deberá enviar únicamente estos ficheros.

Sea muy cuidadoso a la hora de crear los ficheros, y **respete los nombres indicados**.

Además, **estos ficheros deben compilar sin errores** con las siguientes órdenes:

```
ocamlc -c qsort.ml
ocamlc -c msort2.ml
```

Como siempre, las respuestas "de palabra" que se piden en algunos de los apartados deben ser incluidas como comentarios en los ficheros `.ml`.

Ejercicios:

1. Considere la siguiente implementación del algoritmo *quicksort*:

   ```
   let rec qsort1 ord = function
       [] -> []
     | h::t -> let after, before = List.partition (ord h) t in
               qsort1 ord before @ h :: qsort1 ord after;;
   ```

   ¿En qué casos no será bueno el rendimiento de esta implementación? Para evitar problemas con la no terminalidad de `(@)` podemos hacer el siguiente cambio:

   ```
   let rec qsort2 ord =
     let append' l1 l2 = List.rev_append (List.rev l1) l2 in
     function
         [] -> []
       | h::t -> let after, before = List.partition (ord h) t in
                 append' (qsort2 ord before) (h :: qsort2 ord after);;
   ```

   ¿Tiene `qsort2` alguna ventaja sobre `qsort1`? ¿Permite `qsort2` ordenar listas que no podrían ordenarse con `qsort1`? En caso afirmativo, defina un valor `l1 : int list` que sea ejemplo de ello. En caso negativo, defina `l1` como la lista vacía.

   ¿Tiene `qsort2` alguna desventaja sobre `qsort1`? Compruebe si `qsort2` es más lento que `qsort1`. Si es así, explique por qué y estime la penalización, en porcentaje de tiempo usado, de `qsort2` respecto a `qsort1`.

2. Considere la siguiente implementación de la ordenación por fusión:

   ```
   let rec divide l = match l with
       h1::h2::t -> let t1, t2 = divide t in (h1::t1, h2::t2)
     | _ -> l, [];;
   ```

```
let rec merge = function
    [], l | l, [] -> l
  | h1::t1, h2::t2 -> if h1 <= h2 then h1 :: merge (t1, h2::t2)
                      else h2 :: merge (h1::t1, t2);;

let rec msort1 l = match l with
    [] | _::[] -> l
  | _ -> let l1, l2 = divide l in
         merge (msort1 l1, msort1 l2);;
```

Redefina las funciones `merge` y `msort1`, con tipos `('a -> 'a -> bool) -> 'a list * 'a list -> 'a list` y `('a -> 'a -> bool) -> 'a list -> 'a list`, respectivamente, de modo que puedan ser utilizados con cualquier orden (y no solo con `(<=)`).

¿Puede provocar algún problema la no terminalidad de `divide` o `merge`? En caso afirmativo, defina un valor `l2 : int list` que sea un ejemplo de ello. En caso negativo, defina `l2` como la lista vacía.

Defina de modo recursivo terminal funciones `divide'` y `merge'` que cumplan el mismo cometido que `divide` y `merge`, respectivamente. Realice una implementación, `msort2`, de la ordenación por fusión utilizando `divide'` y `merge'`. Compare el rendimiento en tiempo de ejecución de `msort2` con el de `msort1` y con el de `qsort2`.