TD de *Logique et Circuits* n° 10

Induction sur les listes

Exercice 1 Le schéma de définition inductive sur les listes vu en cours est :

$$f(l) = g(l, f(h(l)))$$

ou, pour des fonctions à $k+1$ arguments :

$$f(l, a_1, ..., a_k) = g(l, a_1, ..., a_k, f(h(l), i_1(a_1)..., i_k(a_k)))$$

.

Pour chaque fonction Ocaml de la liste suivante, donner l'équation correspondante :

1. ```
   let rec parite liste=
     match liste with
       [] -> true
     |a::l-> not(parite l);;
   ```
2. ```
   let rec membre a l =
     match l  with
       []    -> false
     | b::j  -> a=b || membre a j;;
   ```
3. ```
   let rec somme_impair liste=
     match liste with
       [] -> 0
     | [a]-> a
     | a::b::k-> a + somme_impair k
   ```

Exercice 2 Le principe du "tri par fusion" (*merge-sort*) est le suivant : pour trier une liste l, on commence par construire les deux sous-listes l_p et l_i, qui contiennent respectivement les éléments d'indice pair et impair de l (cette opération est appelée ici "séparer").

On trie (par fusion) l_p et l_i, puis on fusionne les deux listes triées l_p^{triee} et l_i^{triee}, obtenant ainsi l^{triee}.

On suppose disposer d'une implantation en Ocaml des fonctions de séparation et fusion :

```
separer : 'a liste -> ('a liste * 'a liste)
fusion : ('a liste * 'a liste)-> 'a liste
```

1. Ecrire en Ocaml la fonction `tri_fusion : 'a liste -> 'a liste`.
2. Vérifier que `tri_fusion` n'appartient pas à l'ensemble de fonctions définissables avec le schéma $f(l) = g(l, f(h(l))$, et proposer un schéma plus général, qui inclut des fonctions comme `tri_fusion` (ou `tri_rapide`).
3. Calculer la complexité de `tri_fusion` en termes du nombre de comparaisons, sachant que le nombre de comparaisons nécessaires pour fusionner deux listes de longueur n est au mieux n, au pire $2n-1$, et que l'opération de séparation ne fait pas de comparaisons.
```

Exercice 3 L'intersection de deux listes sans répétitions l_1 et l_2 est une liste qui contient les éléments qui apparaissent à la fois dans l_1 et dans l_2.

Une fonction auxiliare utile pour programmer l'intersection est `membre : 'a -> 'a list -> bool` (exercice 1).

1. Ecrire la fonction Ocaml `intersection : 'a list -> 'a list -> 'a list`, qui réalise l'équation :
   $f(l_1, l_2) = g(l_1, l_2, f(h(l_1), i_1(l_2)))$
   $g(a :: l_1, l_2, r) = \begin{cases} a :: r & \text{si } \texttt{membre}(a, l_2) \\ r & \text{sinon} \end{cases}$
   $h(a :: l) = l$
   $i_1(l) = l$

Pour calculer l'intersection de deux listes triées, il n'est pas nécessaire d'utiliser le test d'appartenance.

2. Ecrire la fonction
   `intersection2 : ('a list * 'a list) -> 'a list`
   utilisant `tri_fusion` et n'utilisant pas `membre`.
3. Comparer les complexités de `intersection` et `intersection2`, en termes du nombre de comparaisons et tests d'égalité (qu'on ne distingue pas).

Exercice 4 Prouver la proposition suivante : Pour toute liste l `intersection` $l\ l = l$.