Option Informatique en Sup MPSI
Calculs dans l'algèbre des parties finies ou cofinies de $\mathbb{N}$: le corrigé

Question 1 • L'écriture suivante est récursive terminale, et exploite l'évaluation paresseuse des booléens. Son coût est clairement linéaire en la longueur de la liste, dans le pire des cas.

```
let rec mem x = function
 | [] -> false
 | t::q -> x=t or mem x q ;;
```

Question 2 • Ici, l'écriture n'est pas récursive terminale ; cette propriété pourrait aisément être obtenue, en mettant en œuvre un accumulateur.

```
let rec filtre p = function
 | [] -> []
 | t::q when p t -> t::(filtre p q)
 | _::q -> filtre p q ;;
```

Le coût est proportionnel à la longueur de la liste ℓ, puisque l'on effectue exactement un *Cons* pour chaque élément de cette liste.

Question 3 • L'écriture est immédiate : il suffit de filtrer la liste ℓ_2 avec le prédicat ≪appartenance à la liste ℓ_1≫. Le coût de chaque filtrage est dominé par la longueur de ℓ_1, donc le coût du calcul est dominé par le produit des longueurs des deux listes.

```
let intersection_simple l1 l2 =
 filtre (fun x -> mem x l1) l2 ;;
```

Question 4 • La réponse naïve `let union_simple = prefix @` ne satisfait pas la spécification, puisqu'elle peut introduire des doublons. On va quand même utiliser l'opérateur `@` de concaténation de listes, mais en filtrant l'une des listes, pour ne garder que les éléments qui n'appartiennent pas à l'autre. Le coût est dominé par le produit des longueurs des deux listes, pour la même raison qu'à la question précédente.

```
let union_simple l1 l2 =
 (filtre (fun x -> not (mem x l2)) l1) @ l2 ;;
```

Question 5 • La solution suivante est récursive terminale.

```
let rec for_all p = function
 | [] -> true
 | t::q -> (p t) & for_all p q ;;
```

Le coût est proportionnel à la longueur de la liste ℓ, puisque l'on effectue exactement un *Cons* pour chaque élément de cette liste. On peut imaginer des formulations plus concises ; en voici deux exemples, l'une utilisant `filtre` et l'autre la fonctionnelle `it_list`. Leur coût reste proportionnel à $|\ell|$.

```
let for_all p l = (filtre p l) = l ;;
let for_all p l = it_list (prefix &) true (map p l);;
```

Question 6 • La stabilité par complémentation est évidente. Avec la formule $F \cup G = \overline{\overline{F} \cap \overline{G}}$, la stabilité par réunion se réduit à la stabilité par intersection, qu'il reste donc à établir.

• L'intersection d'une partie finie et d'une partie quelconque (cofinie ou non) est finie ; quant à l'intersection de deux parties cofinies $F = \mathbb{N} \setminus P$ et $G = \mathbb{N} \setminus Q$, elle est elle aussi cofinie car $\mathbb{N} \setminus (F \cap G) = P \cup Q$ est finie en tant que réunion de deux parties finies.

• Soit $\mathcal{V}$ une famille de parties de $\mathbb{N}$ contenant les parties finies et stable pour les opérations booléennes : elle est en particulier stable pour la complémentation, donc elle contient les parties cofinies, si bien que $\mathcal{W} \subset \mathcal{V}$.

Question 7 • L'appartenance à une partie finie se teste directement en appliquant `mem` à la liste qui la représente. L'appartenance à une partie cofinie équivaut à la non-appartenance à son complémentaire.

```
let appartient x = function
 | Finie f -> mem x f
 | Cofinie f -> not (mem x f) ;;
```

Question 8 • Revenons à la définition mathématique : $Q \supset P$ ssi tout élément de P appartient à Q. Nous disposons des fonctions `appartient` et `for_all`. Si Q est finie, on vérifie que tous ses éléments appartiennent à P ; si P et Q sont cofinies, on vérifie que $\overline{Q}$ contient $\overline{P}$; enfin, si P est finie et Q cofinie, la réponse est `false` ûisqu'une partie finie ne peut pas contenir une partie cofinie !

```
let rec contient p q = match (p,q) with
 | (_,Finie qq) -> for_all (fun x -> appartient x p) qq
 | (Cofinie pp,Cofinie qq) -> contient (Finie qq) (Finie pp)
 | (Finie _,Cofinie _) -> false ;;
```

Question 9 • L'écriture est immédiate, avec la structure de données choisie.

```
let complement = function
 | Finie f -> Cofinie f
 | Cofinie f -> Finie f ;;
```

Question 10 • Si les deux parties sont finies, il suffit d'utiliser `union_simple`. Sinon, la réunion est une partie cofinie ; le cas de la réunion de deux parties cofinies est simple, avec la relation :

$$\big(\mathbb{N} \setminus P\big) \cup \big(\mathbb{N} \setminus Q\big) = \mathbb{N} \setminus \big(P \cap Q\big)$$

Il reste le cas de la réunion d'une partie finie P et d'une partie cofinie $\mathbb{N} \setminus Q$. Le résultat est une partie cofinie $\mathbb{N} \setminus R$. On a :

$$R = \mathbb{N} \setminus \big(P \cup (\mathbb{N} \setminus Q)\big) = (\mathbb{N} \setminus P) \cap Q$$

Donc R est l'ensemble des éléments de Q qui *n'appartiennent pas* à P ; le calcul de cette partie se fait avec `filtre`.

```
let union f g = match (f,g) with
 | (Finie p,Finie q) -> Finie (union_simple p q)
 | (Cofinie p,Cofinie q) -> Cofinie (intersection_simple p q)
 | (Finie p,Cofinie q) -> let fp = fun x -> not (mem x p)
    in Cofinie (filtre fp q)
 | (Cofinie p,Finie q) -> let fq = fun x -> not (mem x q)
    in Cofinie (filtre fq p) ;;
```

Question 11 • L'écriture est immédiate si l'on veut bien se souvenir cette fois que $F \cap G = \overline{\overline{F} \cup \overline{G}}$.

```
let intersection f g =
 let f1 = complement f and g1 = complement g
 in complement (union f1 g1) ;;
```

Question 12 • Le test d'appartenance a un coût égal à la longueur de la liste, dans le pire des cas. Le test d'inclusion est réalisé par application à tous les éléments d'une liste du test d'appartenance à l'autre liste ; donc son coût est égal au produit des longueurs des deux listes dans le pire des cas. La complémentation a un coût nul (toujours en nombre de *Cons*). La réunion et l'intersection ont elles aussi un coût au moins égal au produit des longueurs des deux listes dans le pire des cas, puisque l'on applique à tous les éléments d'une liste le test d'appartenance ou de non-appartenance à l'autre liste.

Question 13 • En changeant de structure de données pour représenter les parties finies ou cofinies de $\mathbb{N}$, on peut diminuer le coût de la recherche d'un élément. Avec des arbres binaires de recherche, ce coût passera de n à $\ln(n)$ (en moyenne), où n est le nombre d'éléments dans la structure ; si l'on utilise des arbres binaires *équilibrés*, le coût sera un $\mathcal{O}\big(\ln(n)\big)$ dans le pire des cas. Enfin, en faisant appel à une table de hachage, le coût moyen devient constant.

Question 14 • Il suffit de redéfinir le type `ficof` en le paramétrant :

```
type 'a ficof = Finie of 'a list | Cofinie of 'a list ;;
```

Le type de la fonction `appartient` deviendra `'a -> 'a ficof -> bool`. Le polymorphisme de Caml fera le reste ! Peut-on avoir quand-même le beurre et l'argent du beurre ? On ne pourra implanter une structure de données performante que si le type de base le permet. En particulier, l'emploi d'arbres binaires de recherche n'est possible que si ce type est muni d'une relation d'ordre total ; il se trouve que, sur *tout* type prédéfini ou défini par l'utilisateur, il existe une relation d'ordre. Donc tout se passe bien : c'est un vrai conte de fées !

FIN