Option Informatique en Spé MP et MP*
Programmation Caml : intervalles discrets, le corrigé

Question 1 • La stabilité par réunion est une conséquence immédiate de la définition ; la stabilité par passage au complémentaire est claire. Enfin, pour l'intersection, le plus simple est de faire appel à la formule $X \cap Y = \overline{\overline{X} \cup \overline{Y}}$ (qui sera utilisée à la question 14) ; on peut aussi remarquer que l'intersection de deux intervalles est soit vide, soit un intervalle.

Question 2 • La réponse est brève : $\preceq$ est l'ordre lexicographique sur $\overline{\mathbb{Z}} \times \overline{\mathbb{Z}}$, induit par l'ordre qui a été défini sur $\overline{\mathbb{Z}}$.

Question 3 • Si $\mathcal{I}$ et $\mathcal{J}$ ne sont pas disjoints, ou s'ils sont consécutifs, on peut les remplacer par l'intervalle $[\![\inf \mathcal{I}, \sup \mathcal{J}]\!]$ et diminuer ainsi la taille de la représentation.

Question 4 • Commençons par trier la liste selon l'ordre $\preceq$, pour un coût $\mathcal{O}\big(n \ln(n)\big)$. Il suffit ensuite de parcourir la liste triée : s'il existe deux intervalles consécutifs dans la liste, et qui sont consécutifs dans $\overline{\mathbb{Z}}$ ou non disjoints, on les remplace par l'intervalle résultant de leur fusion. Remarquons que le coût est un $\mathcal{O}(n^2)$ si l'on trie *après* les fusions.

Question 5 • Écriture immédiate :

```
let intervalle_of_bornes p q =
 if p > q then raise (Invalid_argument "intervalle_of_bornes")
         else (Relatif p,Relatif q) ;;
```

Question 6 • Écriture immédiate :

```
let dd_g q = (MoinsInf,Relatif q) ;;
let dd_d p = (Relatif p,PlusInf) ;;
```

Question 7 • Voici une rédaction économique ; l'opérateur infixe `<<` allège les écritures ultérieures.

```
let z_précède_strict x y = match (x,y) with
 | (MoinsInf,MoinsInf) -> false
 | (MoinsInf,_ ) -> true
 | (Relatif p,Relatif q) -> p < q
 | (_,_) -> false ;;

let prefix << = z_précède_strict ;;

let z_précède_large x y = (x = y) or (x << y) ;;
```

Question 8 • Écriture immédiate :

```
let z_consécutifs x y = match (x,y) with
 | (Relatif p,Relatif q) when q = p + 1 -> true
 | (_,_) -> false ;;
```

Question 9 • Écriture immédiate, en filtrant explicitement les couples :

```
let consécutifs (_,ud) (vg,_) = z_consécutifs ud vg ;;
```

Question 10 • Écriture immédiate, cette fois nous utilisons `fst` et `snd` :

```
let est_à_gauche_de u v = (snd u) << (fst v) ;;
```

Question 11 • La fonction `est_avant` réalise l'ordre lexicographique évoqué à la question 2. La fonction `réduire` commence par trier la liste d'intervalles selon cet ordre, puis applique la fonction `compacter` ; le cas 1 est celui de deux intervalles consécutifs ; le cas 2, celui de deux intervalles disjoints non consécutifs ; le cas 3, celui de deux intervalles non disjoints mais dont aucun n'est contenu dans l'autre ; enfin, le cas 4 est celui de deux intervalles dont un est contenu dans l'autre.

```
let rec compacter = function
 | u::v::q when consécutifs u v -> let t = (fst u,snd v) in compacter (t::q) (* 1 *)
 | u::v::q when est_à_gauche_de u v -> let q' = compacter (v::q) in u::q'    (* 2 *)
 | u::v::q when z_précède_large (snd u) (snd v)                              (* 3 *)
   -> let t = (fst u,snd v) in compacter (t::q)
 | u::_::q -> compacter (u::q)                                               (* 4 *)
 | [] -> []
 | [i] -> [i] ;;

let est_avant (ug,ud) (vg,vd) = ug << vg or (ug = vg & ud << vd) ;;

let réduire l = let l1 = sort__sort est_avant l in compacter l1 ;;
```

Question 12 • L'écriture suivante est brutale ; si les listes `l1` et `l2` sont triées, on peut effectuer une simple fusion, de coût $\mathcal{O}\big(|\ell_1|+|\ell_2|\big)$. Rédigez donc cette fusion !

```
let union l1 l2 = réduire (l1 @ l2) ;;
```

Question 13 • L'écriture de la fonction `complémentaire` est assez technique. Les fonctions `successeur` et `prédécesseur` calculent le successeur et le prédécesseur d'un élément de $\overline{\mathbb{Z}}$. Le démarrage et l'arrêt du calcul du complémentaire doivent être traités à part. Rappel : `x as y::z` est compris comme `(x as y)::z`.

```
let successeur = function
 | Relatif p -> Relatif (p+1)
 | _ -> raise (Invalid_argument "successeur") ;;

let prédécesseur = function
 | Relatif p -> Relatif (p-1)
 | _ -> raise (Invalid_argument "prédécesseur") ;;

let fin_complémentaire = function
 | (_,PlusInf) -> []
 | (_,Relatif p) -> [dd_d (p+1)]
 | (_,MoinsInf) -> raise (Invalid_argument "fin_complémentaire") ;;

let rec suite_complémentaire = function
 | [u] -> fin_complémentaire u
 | u::v::q -> let t' = (successeur(snd u),prédécesseur(fst v))
   in t'::(suite_complémentaire (v::q))
 | [] -> raise (Invalid_argument "suite_complémentaire") ;;

let complémentaire_bis = function
 | [] -> [(MoinsInf,PlusInf)]
 | (MoinsInf,_) as t::q -> suite_complémentaire (t::q)
 | (Relatif p,_) as t::q ->
    let t' = (MoinsInf,Relatif (p-1))
    in t'::(suite_complémentaire (t::q))
 | (PlusInf,_)::_ -> raise (Invalid_argument "complémentaire") ;;

let complémentaire l = complémentaire_bis(réduire l) ;;
```

Question 14 • Nous utilisons $X \cap Y = \overline{\overline{X} \cup \overline{Y}}$:

```
let intersection l1 l2 =
 let l1' = complémentaire(réduire l1) and l2' = complémentaire(réduire l2)
 in complémentaire (union l1' l2') ;;
```

FIN