TP 5: Sets

1 Sets – First Attempt

Pour cet exercice on va programmer une structure de données qui représente la notion d'ensemble. On va donc définir un type `Myset` qui représente des objets qui contiennent d'autres objets. Contrairement aux listes, les `Myset` ne sont pas ordonnés, et ne contiennent pas de doublons.

Pour cet exercice on vous demande de programmer une implémentation directe du type `Myset` qui utilise des listes. On commence donc avec une déclaration du style :

```
data Myset a = Myset [a]
```

Notez bien que `Myset` est un type paramétrique, on peut donc utiliser de `Myset` de `Integer, Char, Double` etc. Notez aussi que ce type contient en réalité un seul élément : une liste de type `[a]`. Nous devons faire en sorte que cette liste ne contienne jamais de doublons, et donc représente un vrai ensemble. L'ordre des éléments contenus dans la liste est sans importance.

On vous demande de programmer (au moins) les fonctions suivantes pour manipuler les objets de type `Myset`. Faites attention aux signatures et aux contraintes de type.

1. Fonctions `list2set` et `set2list` pour faire la conversion entre listes et `Myset`. La fonction `list2set` devrait ignorer des doublons éventuels dans la liste passée comme argument.
2. Faire en sorte que le type `Myset` soit une instance de la classe `Show`. Pour l'affichage, la fonction `show` devrait retourner une liste des éléments séparés par des virgules, entre les caractères `{` et `}`. (Voir exemples ci-dessous).
3. Une fonction `belongs` qui étant donné un élément x et un ensemble S vérifie si $x \in S$.
4. Une fonction `size` qui retourne la cardinalité de l'ensemble donné.
5. Des fonctions `union, intersection, difference` qui, étant donné deux ensembles S, T, retournent $S \cup T$, $S \cap T$, $S \setminus T$ respectivement.
6. Une fonction `subset` qui, étant donné S, T vérifie si $S \subseteq T$.
7. Une fonction `powerset` qui, étant donné S, retourne $\mathcal{P}(S)$, l'ensemble de tous les sous–ensembles de S.
8. Faire en sorte que le type `Myset` soit une instance de la classe `Eq`.

1.1 Comment faire

Puisqu'on demande une implémentation qui utilise des listes, la plupart des fonctions de cet exercice devraient être faciles à implémenter de façon directe. Il faut cependant faire attention aux types. L'efficacité de votre implémentation n'est pas trop importante pour cette partie (mais soyez raisonnable !)

Pour tester que votre programme fonctionne correctement, essayez (parmi d'autres) les exemples ci-dessous.

```
*Main> list2set "hellothere"
{ 'h', 'e', 'l', 'o', 't', 'r' }
*Main> intersection (list2set "abcde") (list2set "abracadabra")
{ 'a', 'b', 'c', 'd' }
*Main> map (\x -> belongs x  (list2set [1,3..10])) [1..10]
[True,False,True,False,True,False,True,False,True,False]
*Main> s1 = list2set "letitbe"
*Main> s2 = list2set "shelovesyouyeahyeahyeah"
*Main> union s1 s2
{ 't', 'i', 'b', 's', 'h', 'e', 'l', 'o', 'v', 'y', 'u', 'a' }
*Main> difference s1 s2
{ 't', 'i', 'b' }
*Main> difference s2 s1
{ 's', 'h', 'o', 'v', 'y', 'u', 'a' }
*Main> intersection s1 s2
{ 'l', 'e' }
*Main> union s1 s2 == union (difference s1 s2) (union (difference s2 s1)
                                                      (intersection s1 s2))
True
*Main> powerset (list2set [1,2,3])
{ { }, { 3 }, { 2 }, { 2, 3 }, { 1 }, { 1, 3 }, { 1, 2 }, { 1, 2, 3 } }
```

2 Sets – Second Attempt

Le désavantage d'une implémentation du type `Myset` avec des listes est sa complexité : tester si un élément appartient à un ensemble (`belongs`) prend de temps linéaire ($O(n)$), alors que tester si un ensemble est un sous-ensemble d'un autre (`subset`) prend de temps quadratique ($O(n^2)$). Pour cet exercice on vous demande de donner une implémentation alternative du type `Myset` en utilisant la notion des arbres binaires de recherche (cf. TD 7). On va commencer avec une déclaration du style :

```
data Myset a = Myset (BinTree a)
```

où vous devez aussi donner une définition pour le type `BinTree` (comme la définition du TD7). L'idée est que le type `BinTree` représente un arbre où chaque nœud a deux enfants (gauche, droit), et une valeur x et satisfait la condition que toutes les valeurs qui se trouvent dans le sous–arbre gauche (respectivement droit) sont $< x$ (respectivement $> x$). Comme on a vu au TD (en vous avez aussi vu dans le cours d'algorithmique et programmation), l'intérêt d'utiliser une telle structure est que la complexité de rechercher une clé est proportionnelle à la hauteur de l'arbre, qui pourrait être largement inférieure à n.

- On vous demande de programmer toutes les fonction de l'exercice précèdent avec cette nouvelle formulation, sauf la fonction `powerset`.

Vérification

Tout d'abord, tester votre nouvelle implémentation avec les exemples de l'exercice précèdent. Vous devrez avoir les mêmes résultats.

Maintenant, l'intérêt de cette nouvelle implémentation et son efficacité (espérée), et son efficacité dépend de la hauteur de l'arbre construit. Pour vous faciliter la tâche, on va tester avec des listes de valeurs qui sont triées en ordre (quasi–)aléatoire. Grâce à cette supposition, la fonction `list2set` peut facilement construire un arbre de hauteur raisonnable, sans faire un effort pour trouver l'élément médian de la liste et donc partitionner les données en deux moitiés.

Ajoutez les lignes suivantes à votre programme :

```
-- test: random array

myrand :: Integer -> Integer
myrand x = (134775813*x+1) `mod` (2^32)

randlist :: Int -> [Integer] -> [Integer]
randlist 0 xs = xs
randlist n (x:xs) = randlist (n-1) ((myrand x):x:xs)
```

Vous pouvez jouer avec la fonction `randlist` en donnant par exemple `randlist length [seed]`, où `length, seed` sont des entiers. Cette fonction répond avec une liste de taile `length` de `Integer` pseudo–aléatoires.

Pour testez l'efficacité de votre programme, sur `ghci` donnez la commande `:set +s`. Avec cette commande `ghci` va vous afficher après chaque expression le temps d'exécution qui a été utilisé. On fait donc la vérification suivante :

```
*Main> longlist = randlist 100000 [1]
(0.00 secs, 0 bytes)
*Main> longlist2 = randlist 100000 [(longlist !! 10)]
(0.00 secs, 0 bytes)
*Main> intersection (list2set longlist) (list2set longlist2)
{ 27508586, 252834151, 331400544, 689833192, 1326109444, 2942154889, 3258279187,
              3426959022, 3626250005, 4069164283, 4085865185 }
(7.44 secs, 2,819,094,440 bytes)
*Main> size (union (list2set longlist) (list2set longlist2))
199991
(8.33 secs, 3,029,428,032 bytes)
```

Notez que les deux premières lignes sont exécutées en 0 secondes : grâce à l'évaluation paresseuse l'affectation `longlist =..` n'a aucun effet jusqu'au moment où on tente vraiment d'utiliser la variable `longlist`.

• Pour les questions suivantes, donnez une courte réponse en commentaire dans votre programme.

1. Pourquoi est-ce que la dernière ligne retourne 199991 ?
2. Pouvez–vous faire la même vérification avec l'implémentation qui utilise des listes ? Cela vous prend combien de temps ?
3. Pourquoi est-ce qu'on utilise une liste pseudo–aléatoire ? Que va–t–il se passer si on vous demande la même vérification mais avec `longlist = [1..100000]` et `longlist2 = [10..100010]` ? Pourquoi ?