Principes de Programmation
TP5: La do-construction

15 mars 2019

Exercice 1 (La monade des listes). [1]

1. Voici un programme sur les listes en style monadique utilisant ce que l'on appel une do-construction :

```
zipAll :: [a] -> [b] -> [(a,b)]
zipAll p q = do
  x <- p
  y <- q
  [(x,y)]
```

   que fait-il ? (testez-le)

2. En voici un autre,

```
ajoutNeg :: [Int] -> [Int]
ajoutNeg l = do
  x <- l
  [x,-x]
```

   que fait-il ? (testez-le)

3. Il faut compredre les programme ci-dessus comme cecis : le premier séléctionne non-déterministiquement un élément de `p` et un élément de `q` et renvoit le couple `(p,q)`, le resultat est la liste de tous les résultats possibles. Le second séléctionne non-déterministiquement un élément `x` de `l` et renvoit, non-déterministiquement, `x` ou `-x`, le resultat est la liste de tous les résultats possibles. Écrire la fonction `double :: [Int] -> [Int]` qui, sur une liste (ex : `[10,2,40]`) renvoit toutes les multiplications possibles (ex : `[100,20,400,4,80,1600]`).

4. Écrivez un programme `sousListes :: [a]->[[a]]` qui calcule l'ensemble des parties en style monadique (dans la monade des listes). On rappel que pour donner non-déterministiquement une partie d'une liste `(x:l)`,

1. Dans la vrais vie, on ne calcule jamais l'ensemble des parties ou les partitions d'une liste, car le seul résultat prend un espace exponentiel (il y a un nombre exponentiel de parties), du coup, même si on a besoin d'en calculer quelques-uns, on en calcule une et on la jette... Celà n'empèche pas l'exercice d'être très intéressant et de montrer plein d'optimisations que l'on peut faire en programmation fonctionnelle.

il suffit ou bien de donner (non-déterministiquement) une partie de l, ou bien la même partie de `l` avec `x` en tête.

5. (bonus) Dessinez la représentation mémoire (chainage des pointeurs) de `(sousListes [1,0])`, puis celle de `(sousListes [2,1,0])`, puis celle de `(sousListes [3,2,1,0])`. Que remarquez-vous ?
6. (bonus) Montrez que la taille en mémoire de l'ensemble des parties d'une liste à n éléments est $(\frac{n}{2}+1)$ fois plus compacte que la représentation naive. Ce genre de gain de complexité en espace est un avantage des structures non mutables, [2] il apporte rarement un changement de classe de complexité, mais il permet souvant de gagner un bon facteur.
7. Essayer d'écrire le programme `parties :: Int->[a]->[[a]]` qui calcule l'ensemble des parties d'une taille donnée en style monadique et en suivant l'algorithme (buggué) suivant :
   — il y a une unique partie à zéro élément (la liste vide) quelque soit la liste entrée,
   — sur une liste vide, il n'y a pas de parties de taille $n \neq 0$,
   — pour donner (non déterministiquement) un élément de `(parties n (x:l))`, je choisi un élément `p` de `(parties (n-1) l)`, un élément `q` de `(parties n l)` et on renvoit l'un ou bien le premier avec x ajouté en tête, ou bien le second sans le x.
8. testez votre programme, vous devriez avoir une mauvaise surprise...
9. le problème est que l'on récupère un élément de `(parties (n-1) l)`, puis un élément de `(parties n l)`, ce qui revient à récupérer un couple, il peut donc y avoir des doublons, ou au contraire (le soucis ici), ne pas en avoir car `(parties n l)` peut être vide sans que `(parties n (x:l))` le soit, on ne trouve alors pas de `q` et on ne renvoit rien.
10. une solution est de faire le choix non-déterministe avant de récupérer la partie en question. Pour ca on peut simplement tirer un booléen non-déterministiquement, puis faire un if. Voici le code pour ca :

```
partie n (x:l) = do
  b <- [True,False]
  if b then do
          p <- parties (n-1) l
          return (x:p)
       else do
          parties n (x:l)
```

   Completez avec l'initialisation de la récurence et vérifiez que ca marche.
11. une autre solution moins élégante mais équivalente (peut-être même un peu plus rapide) est d'utiliser `map` et `(++)` plutôt que la structure de monade :

2. Faire ca avec des mutables est théoriquement possible, mais systématiquement buggué car trop complèxe.

```
partie n (x:l) = (map (\p->x:p) (parties (n-1) l))
                 ++ parties n (x:l)
```

Completez avec l'initialisation de la récurence et vérifiez que ca marche.

12. Écrire la fonction `appliqueUneFois :: (a->a) -> [a] -> [[a]]` qui prend une fonction, une liste et applique la fonction non déterministiquement sur l'un des éléments de la liste. Par exemple `appliqueUneFois (*10) [1,2,3]` rendrait `[[10,2,3][1,20,3][1,2,30]]`.
13. Essayez maintenant d'énumérer les partitions `partions :: [a]->[[[a]]]`, une partition de `l` (on suppose que `l` est sans duplication) est séquence `[l1,...,ln]` de listes qui sont deux à deux disjointes et dont l'union donne `l` (à réorganisation près). L'idée de l'algorithme est d'utiliser le fait que si `[l1,....,ln]` est une partition de `l`, alors `[l1,..,x:li,...ln]` est une partition de `x:l` et que toutes peuvent être obtenu ainsi. On utilisera la fonction `appliqueUneFois`.
14. Écrivez `permutations :: [a]->[[a]]` qui donne toutes les permutations possibles de la liste entrée.

1 À faire chez soi :

Exercice 2 (Mémoïsation).
Le calcule de `partie` n'était pas "optimal", au sense où on recalcule plusieurs fois la même chose, en effet, pour calculer `parties 2 [1,2,3,4]`, on doit calculer `partie 1 [2,3,4]` et `partie 2 [2,3,4]`, sauf que dans le calcule de chacun des deux on a le calcule de `partie 1 [3,4]`, qui est donc fait deux fois. Celà explose la complexité (qui était déjà exponentielle donc on s'en fiche, mais quand même). Ce genre de problème est résolut par de la mémoïsation, aussi appellée programation dynamique. On va essayer de voir comment le faire en Haskell. On va commencer avec les binomiaux (c'est à dire du nombre de partitions possibles)

15. Écrire la fonction naive (et exponentielle) de calcule des binomiaux.
16. Voici un code mémoïsé de la même fonction :

```
binomial :: Int -> Int -> Int
binomial 0 j = 1
binomial i 0 = 0
binomial i j = (binList !! i !! (j-1)) + (binList !! (i-1) !! (j-1))

binList :: [[Int]]
binList = map (\i-> map (binomial i) [0..]) [0..]
```

L'idée est simple : `binList` est un matrice infinie qui en position (i, j) contient le binomial `binomial i j`, gràce à la paraisse, cette matrice se remplie uniquement lorsque l'on calcule un binomial, de plus une fois qu'on a calculé un binomial intermédiaire, il est stoqué dans le tableau et on n'a pas besoin de le recalculer.

17. Écrire une version mémoisé de fibonacci.
18. On ne peut pas adapter cette technique directement au calcule des parties, car on ne peut pas créer une structure avec toutes les parties de toutes les listes possibles. Par contre, ce que l'on peut faire c'est, étant donné une liste, dréer une matrice contenant, en position (i,j), les patries de taille i dans la sous liste amputée des j premiers éléments. Compléter, ainsi, le début de programme suivant :

```
parties n l = partMemo n l
  where
    partListe = map (\i-> map (partMemo (drop i l)) [0..]) [0..]
    partMemo l k = undefined
```

19. Utiliser `(drop i l)` n'est pas élégant car on reparcout la liste à chaque fois (en vrais ce n'est pas grave car on est déjà exponentiel), trouvez un moyen d'écrire `partListe` directement

Exercice 3 (Graphes).
Un noeud du graphe est donné par une information et un liste de noeuds ; un graphe est alors une liste de noeuds :

```
data Noeud  = NOEUD [Noeud]
type Graphe = [Noeud]
```

Attention, le graphe devra rester bidirectionnel.

1. Dérivez les classes habituelles.
2. Créez un graphe avec deux noeuds "noeud1" et "noeud2", pointant l'un sur l'autre.
3. Affichez ce graphe (en utilisant show).
4. Comment remédier au problème rencontré ? [3]
5. Écrivez une fonction :

   ```
   nouveauNoeudLie :: Graphe -> String -> Graphe
   ```

   qui rajoute un noeud (de nom donné) lié uniquement au premier noeud du graphe (celui en tête de liste).
6. Testez cette fonction sur votre graphe pour ajouter "noeud0". Tout va bien ?
7. Écrivez une fonction `getAdj2 :: Noeud -> [Noeud]` affichant les noeuds à distance 2 de l'entrée.
8. Maintenant afficher les noeuds à distance 2 de "noeud2" ; que se passe-t-il ?
9. Comment remédier à ce problème ? Quelle est la complexité de la solution ?

3. voir TP 4

En fait, le problème réel est le suivant : les données de haskell ne sont pas "mutable", c'est à dire que si l'on change quelque chose, on crée une nouvelle donnée et tous les pointeurs vers l'ancienne restent là où ils sont. Cela ajoute de la sécurité et ne coûte pas grand chose en terme de ressource, par contre il faut faire atention lorsque l'on a besoin de modifier une structure avec des boucles comme des automates ou des graphes.

Le meilleur moyen de s'en sortir est alors d'implémenter nous même des pointeurs ! Pour cela, c'est très simple, il faut changer le type du graphe par :

```
data Noeud a = NOEUD String a [String]
type Graphe a = [Noeud a]
```

L'information sert alors de clé pour retrouvé un noeud dans le graphe. On voit alors qu'il n'y a qu'un seul pointeur réel sur un noeud : celui dans le graphe. On a juste créé un nouveau pointeur virtuel qui utilise l'information d'un noeud comme "adresse".

10. Écrivez une fonction d'ajout d'une information dans un graphe en donnant une clé fraîche :

    ```
    nouveau :: Graphe -> Graphe
    ```

11. Implémentez quelques fonctions utiles sur ce graphe et faites quelques tests...

Le "soucis" ici est que celà prend du temps de trouver le pointeur dans la liste des noeud du graphe (si le graphe est gros). C'est pour ca que l'on utilise généralement des dictionaires, comme Data.Map (à ne pas confondre avec le fonction map :s), au lieu d'une liste. Pour la phase 4, il va faloir transformer vos ARN en dictionaires (en stoquant, en plus de la clé, une valeur dans chaque noeud).

Exercice 4 (Monade à état).
La monade à état `State` est déjà implémenté dans haskell. Elle a moralement la définition suivante :

```
type State s a = (s -> (a,s))
```

où `s` est le type de l'état courant et `a` est le type sur lequel on travaille.

En fait, il serait plus correcte de dire que c'est `(State s)` qui est une monade, ce pour tout type `s`. En fait, travailler dans `(State s)` veux dire que l'on dispose, dans son environnement, d'une "référence", ou d'un "pointeur" vers un objet de type `s`. Il est alors possible de lire et de modifier cet objet. C'est une autre façon de remplacer les "mutable" (voire exercice précédent).

Cette monade, en plus du `return` et du `(=<<)`, dispose de trois autres fonctions intéressantes : [4]

4. Atention, dans haskell, si vous utilisez les set et get de Control.Monad.State, ces fonctions ne sont pas définies par rapport à State, mais à n'importe quelle type générique de la classe MonadState, dont State est l'instanciation canonique.

```haskell
return       :: a -> State s a
return a     = (\s -> (a,s))

(=<<)        :: (a -> State s b) -> State s a -> State s b
f =<< u      = (\s -> let (a,s')= u s in f a s')

get          :: State s s
get          = (\s -> (s,s))

put          :: s -> State s ()
put s        = (\_ -> ((),s))

runState     :: State s a -> s -> (a,s)
runState u s = u s
```

`get` permet de récupérer l'état, `put` permet de changer l'état, et `runState` permet de récupérer le résultat étant donné un état de départ.

1. Importez `Control.Monad.State`.
2. En utilisant `set` et `get`, écrivez une fonction qui incrémente l'état courant (de type `Int`) et l'affiche :

   ```haskell
   incr :: State Int Int
   ```

On considère des graphe définit par :

```haskell
data Noeud = NOEUD String [String]
type Graphe = [Noeud]
```

L'idée est que le graph est vu comme un mutable ; pour ca on suppose que le graphe fait partie de l'état, c'est à dire que l'on travail dans la monade *StateGraphe*.

3. Retranscrir des fonctions de la question 1.9 avec un type monadique (vous avez le droit d'appeler les originales).

Le soucis, maintenant est que l'on travail souvent avec deux monades à la fois car il faut toujours faire des recherche (due aux pointeurs) et que l'on a des Maybe qui apparaissent. Heureusement, Maybe et State intéragissent bien ensemble.

Moralement veut maintenant travailler avec la monade suivante :

```haskell
type GraphEtat a = (Graphe -> (Maybe a, Graphe))

return x = (\gr -> (Just x,gr))

lift :: Maybe a -> GraphEtat a
lift m = return m

transforme :: Maybe (GraphEtat a) -> GraphEtat a
transforme Nothing  = \gr -> (Nothing, gr)
```

```haskell
transforme (Just u) = u

f =<< u  = (\g -> let (a,gr')= u g in transforme (f =<< a) gr')

runStateT :: GraphEtat a -> Maybe (a, Graphe)
```

4. Écrire la monade : [5]

   ```haskell
   type GraphEtat = StatT Graphe Maybe
   ```

   et vérifiez que le résultat est bien une monade enregistré grace à la commande `:info Monad GraphEtat`

5. Écrire une fonction qui récupère (dans un maybe), ajoute ou change un noeud à l'aide de son identifiant :

   ```haskell
   getById  :: Int -> GraphEtat Noeud
   setEtatById  :: Int -> GraphEtat ()
   setLienById  :: (Int,Int) -> GraphEtat ()
   ```

6. Utiliser une do construction pour construire un petit graphe et le modifier.
7. Utiliser une do construction pour récupérer la liste les éléments à distance 3 d'un noeud. Vous pouvez utiliser la fonction `mapM :: Monad m => (a -> m b) -> [a] -> m [b]` ou la fonction `liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c`
8. Remplacer les listes par un tas (un arbre dont l'élément maximal est le premier).

5. On utilise ici un "MonadTransformer" StateT qui compose deux monades en une seule