Principes de Programmation
Partiel 2017

18 juin 2018

Le partiel dure 2 heures. Aucun document n'est autorisé.

Les exercices sont indépendants. Il est autorisé de sauter des questions, mais il faut **indiquer à quelle question vous répondez** et **utiliser une nouvelle copie à chaque exercice**.

Certaines questions sont très difficiles, mais :

— à part les questions longues marquées (†), aucune ne nécessite plus d'une demie page,

— répondre partiellement à une question difficile peut être suffisant,

— il y a 50 points accessibles, la note final sera capée à 20, mais pas renormalisée.

Exercice 1 (Egalités et monades, 15pt)**.**

1. Rappelez les axiomes des monades.
2. Montrer que le type suivant est bien une monade : [1]

```
type Reader r a = r -> a
instance Monad (Reader r) where
  return x = \_ -> x
  f =<< u  = \r -> f (u r) r
```

3. (†) Montrer que le datatype suivant est bien un foncteur :[1]

```
data PTree a = Leaf a | Node (Char -> PTree)
```

4. (†) Montrer qu'il s'agit même d'une monade.

La typeclass des monades gradées n'existe pas en Haskell, mais pourrait être ajoutée. Elle est définie comme suit :

```
class GradMonad mon where
  returnG :: a ->  mon () a
  +<<     :: (Ord cle) => (a -> mon cle1 b) -> mon cle2 a -> mon (cle1,cle2) b
  proj1   :: (Ord cle) => mon (cle,()) b -> mon cle b
  proj2   :: (Ord cle) => mon ((),cle) b -> mon cle b
  assos   :: (Ord cle) => mon ((cle1,cle2),cle3) b -> mon (cle1,(cle2,cle2)) b
```

1. à wraper près.

comme une monade, il doit vérifier les axiomes suivants :

```
proj1 (returnG +<< x    )  ~=   x
proj2 (f +<< (returnG x))  ~=   f x
assos (f +<< (g +<< x)  )  ~=   (\y -> f +<< (g y) ) +<< x
```

5. (†) Montrez que `Dictionaire` est une monade gradée où :

   ```
   type Dictionaire cle cont = [(cle,cont)]
   ```

   est tel que une clé est présente au plus une fois et elles sont rangés en ordre croissant.

Exercice 2 (Programmation en situation, 15pt)**.**
les balises *(1),(2),(3)* renvoient à des explications contextuelles à la fin du sujet (à lire après l'examen).

Les arbres de Merkle et les blockchains sont deux structures de données récursives utilisant des technologies de Hash similaires pour garantir des protocole cryptographiques. Les premiers sont utilisés pour le P2P, et les second pour les monnaies électroniques (dont *Bitcoin*). Dans cet exo nous allons essayer de les analyser. [2]]

Un Hash est un entier générée par une fonction de Hash. En haskell, cette fonction est définie par la classe `Hashable` :

```
type Hash = Int
type Sel = Int
class Hashable a where
  hash :: a -> Hash
  hashWithSalt :: Salt -> a -> Hash
```

Pour cette classe il n'y a pas d'équations à part que `HashWithSalt 0 x = hash x`, mais il y a des conditions plus complexes garantissant la sécurité du `hashWithSalt` (sa non-inversion et la bonne répartition des Hash en particulier). *(1)*

Les arbres de Merkle sont définis comme des arbres binaires dont le contenu est stocké dans les feuilles, et avec des hashs systématiques de tous les sous-arbres.

```
data MerkleTree a =  Feuille Hash Salt a
                   | Noeud Hash Salt (MerkleTree a) (MerkleTree a)
```

L'idée est simple : le premier argument de type `Hash` est de hash de la somme des hashs que l'on peut trouver dans les deux fils, pour les feuilles, on prend le hash de l'objet qui s'y trouve. Par exemple, l'arbre
`Noeud 184551889 8 (Feuille 16777691 1 'H') (Feuille 33555278 2 'h')`
est correcte car
`16777691 = hashWithSalt 1 'H'`,
`33555278 = hashWithSalt 2 'h'`
`184551889 = hashWithSalt 8 (16777691+33555278)`

2. Disclamer : Bien que le principe des bitcoins et de la bulle financière qui l'entoure soient très discutable ; il faut admettre que la technologie est scientifiquement intéressante.

1. Écrire une fonction qui vérifie que l'arbre est correcte.
2. Écrire une fonction qui prend deux noeuds et crée l'arbre correct avec ces deux noeuds comme fils.
3. On définit la classe `Hfunctor` comme des foncteurs uniquement applicable sur des hashables : [3]

   ```
   class HFunctor f where
     hmap :: (Hashable a,Hashable b) => (a -> b) -> f a -> f b
   ```

   Instanciez `HFunctor` avec `MerkleTree` (on ne vous demande pas de vérifier les équations).
4. Implémentez `Eq` en supposant que deux arbres différents n'ont jamais le même Hash.

Les blockchains sont définis de façon semblable mais avec un seul fils (c'est une liste chaînée) et chaque noeud porte un contenu (un "block") de type `a` :

```
data Blockchain a = Feuille | Noeud Hash Sel a (Blockchain a)
```

5. Écrire une fonction qui vérifie que la chaîne est correcte.
6. Instanciez `HFunctor` avec `Blockchain` (on ne vous demande pas de vérifier les équations).
7. Écrire une fonction qui prend une chaîne, un sel et un bloc à intégrer et étend correctement la chaîne.
8. On demande maintenant à ce que les hash commencent toujours par `111` (en écriture binaire), écrire une fonction qui vérifie cette condition. On utilisera la fonction `testBit :: (Bits a) => a -> Int -> Bool` qui test si le ième bit d'un type "Bits", sachant que `Int` implémente bien la classe `Bits`.
9. Écrire une fonction qui prend un bloc, et une chaîne et essaie de trouver un sel pour pouvoir commencer par `111`.
10. Généralisez votre fonction pour prendre en entrée un entier `n` caractérisant le nombre de `"1"` à la suite sur les bits de poids fort.

(2) Il reste un inconnu dans l'algo, comment choisir un bloc si plusieurs mineurs en trouvent en même temps ? Dans ces cas on prend la plus longue chaîne :

11. Implémentez `Eq` en supposant que deux chaînes différentes n'ont jamais le même Hash.
12. Implémentez `Ord` en utilisant l'ordre préfixe sur la chaîne.
13. Récupérez la plus longue chaîne de la liste. *(3)*

Explications contextuelles de l'exercice 2
(1) Dans la pratique, un `hash` simple est typiquement un caste sur un entier.

3. Les équations sont les mêmes excepté que l'on peut supposer que les éléments sont bien hashables pour le résolution.

Un `hashWithSalt` est typiquement créé par un générateur psedo-aléatoire en prenant comme seed `((hash a) + salt)` sauf pour la seed nulle où on rend le hash :

```
hashWithSalt 0    x = hash x
hashWithSalt salt x = random gen
              where gen = mkStdGen ((hash a) + salt)
```

(2) Le principe derrière bitcoin est de n'autoriser que les bloques dont le hash commence par n `"1"` où le n dépend du temps moyen des transactions [4] précédentes. Les mineurs sont ceux qui insèrent les hash dans la chaîne. De cette façon, plus il y a de demande pour faire des transferts de bitcoins, plus la transaction est cher à miner. L'un des drawback est que ce genre d'algorithme passe mal à l'échelle : cela commence à devenir trop cher de faire une transaction pour que le bitcoin ai un intérêt en tant que monnaie (à par pour blanchir de l'argent...).
(3) C'est pourquoi un mineur ne recevra ses bénéfices que s'il a la "chance" de trouver rapidement un sel pour avoir suffisamment de 1 de poids fort avant les autres puis s'il a encore la "chance" de s'insérer dans une chaîne qui va grossir vite. Cela veut dire qu'il est plus ou moins inutile d'espérer miner beaucoup si on n'a pas la puissance de calcule. [5] Moins intuitif, cela veut aussi dire qu'un mineur à intérêt à interagir rapidement avec d'autres mineurs suffisamment rapide...

Exercice 3 (Programmation découverte, 15pt)**.**
Avec une extension de syntaxe, on peut redéfinir les arbre rouge-noir ainsi :

```
data Z   = Z
data S a = S
data R   = R
data N   = N

data ARN c n a where
  Feuille :: ARN N Z a
  NoeudN  :: ARN c1 n a -> a -> ARN c2 n a -> ARN N (S n) a
  NoeudR  :: ARN N  n a -> a -> ARN N  n a -> ARN R n a
```

1. Décrivez ce datatype. En quoi il est différent des datatypes habituels ? En quoi représente-t-il mieux les ARN que ceux utilisés en projet ?
2. Pendant l'insertion dans un arbre rouge noir, il y a des moment où un invariant n'est pas respecté. Lequel ? Écrire un datatype dans le même style décrivant cet situation intermédiaire.
3. (†) Écrire l'insertion dans cette structure (attention à respecter les types).
4. (†) Écrire la suppression sans traiter tous les cas (on s'intéresse à la structure du code, non les cas algorithmiques).

4. L'ajout d'un bloque à la chaîne est appelé transaction car il contiens les informations sur un groupe de transferts de bitcoin.
5. Surtout si on a pas un accès particulièrement peu cher à cette puissance de calcule.