Principes de Programmation
TP3: Le hashage dans les structures de données

18 février 2019

Exercice 1 (P2P).
Les arbres de Merkle sont définis comme des sommes des arbres binaires avec des hashs systématiques.

```
import Data.Hashable
type Hash = Int
type Salt = Int
data MerkleTree a =  Leaf a
                   | Node Hash (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.

Pour ça, on utilise la classe `Hashable` définie dans la librairie `Data.Hashable` par :

```
class Hashable a where
  hash :: a -> Hash
  hashWithSalt :: Salt -> a -> Hash
```

avec `hashWithSalt` comme définition minimale.

Le hash d'une structure est un compression en un entier de toute la structure, le sel permettant de changer de "clé" de compression. Cette compression n'est pas réversible, on perd tout de la structure excepté le fait qu'elle ai ce hash, ce qui est étrangement beaucoup, car en 64 bit, il est extrêmement difficile de trouver deux éléments avec le même hash (en pratique c'est impossible), on peut donc supposer que deux éléments avec des hash différents sont différents.

1. Écrire une fonction qui vérifie que l'arbre est correcte.

   ```
   checkMT :: (Hashable a) => MerkleTree a -> Bool
   ```

2. Écrire une fonction qui prend une contenu et créer une feuille correcte :

   ```
   buildFMT :: (Hashable a) => a -> MerkleTree a
   ```

3. Écrire une fonction qui prend deux noeuds et crée l'arbre correct avec ces deux noeuds comme fils.

```
buildMT :: (Hashable a) => MerkleTree a -> MerkleTree a -> MerkleTree a
```

4. Implémentez `Eq` en supposant que deux arbres différents n'ont jamais le même Hash.
5. Définir la fonction map :

   ```
   hmap :: (Hashable b) => (a -> b) -> MerkleTree a -> MerkleTree b
   ```

6. (difficile) construisez un arbre équilibré [1] à partir d'une liste (prendre des sels quelconques :

   ```
   buildMTequilibre :: (Hashable a) => [a] -> MerkleTree a
   ```

Les arbres de Merkle sont utilisés par des protocoles comme P2P ou Git pour vérifier l'intégrité des paquets. Si l'ensemble est faux, on peut remonter au premier paquet, ou ensemble de paquet qui a le mauvais hash et le redemander.

Exercice 2 (bitcoins [2]).
Les blockchains sont définis de façon assez semblable :

```
data Blockchain a = BLeaf | BNode Hash Salt a (Blockchain a)
```

Chaque élément de type `a` doit être hashable, de sorte à ce que le premier argument de type `Hash` est le hash de la somme des hash de a, et du hash que l'on peut trouver dans son fils, [3] tout ceci "salé" par le sel donné en second argument. Les types de données classiques (entiers, strings, listes... sont hashables). L'objet de type `a` est appelé bloc.

1. Écrire une fonction qui vérifie que la chaîne est correcte.

   ```
   buildBC :: (Hashable a) => Blockchain a -> Bool
   ```

2. écrire un map pour `Blockchain`
3. Écrire une fonction qui prend une chaîne, un sel et un bloc à intégrer et étend correctement la chaîne.
4. 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 classe Bits.

   ```
   triple1Hash :: Hash -> Bool
   ```

5. (difficile) Écrire une fonction qui prend un bloc, et une chaîne et essaie de trouver un sel pour pouvoir insérer le block tout en ayant un hash commençant par `111`.
6. Généralisez votre fonction pour prendre en entrée un entier `n` caractérisant le nombre de `1` à la suite sur les bits de poids fort.

1. Le plus équilibré que vous pourrez.
2. Disclamer : Bien que le principe des bitcoins et de la bulle financière qui l'entoure soient très discutables ; la technologie est scientifiquement intéressante.
3. ou 0 si le fils est une feuille.

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...).

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 :

7. Implémentez `Eq` en supposant que deux chaînes différents n'ont jamais le même Hash.
8. Écrire une autre implémentation de `Ord`, `Blockchain a`, qui considère aussi qu'une chaîne `c1` est plus grande qu'une chaîne `c2` si à leur point de divergence, le hash de la première a plus de `1` en tête.
9. (difficile) On aurait voulut implémenter `Ord` en utilisant la longueur de la chaîne, mais les chaînes sont très longues et on compare des chaînes qui ont un même suffixe connu (la chaîne à l'instant d'avant) et diffèrent de leur préfixe, qui est relativement cours. Ce que l'on veut vraiment, c'est comparer les différentiels entre l'ancienne et les nouvelles valeurs. Mais puisque l'on ne peut pas couper une blockchaine (on perdrait la propriété sur les hashs), on peut donner le type suivant pour un diff :

   ```
   data Diff a = Diff {ancienne::Blockchain a , nouvelle::Blockchain a  }
   ```

   qui est la même chose que de définir

   ```
   data Diff a = Diff (Blockchain a) (Blockchain a)
   nouvelle :: Diff a -> Blockchain a
   nouvelle (Diff x y) = x
   ancienne :: Diff a -> Blockchain a
   ancienne (Diff x y) = y
   ```

   puis implémentez Ord dessus en parcourant les deux nouvelles listes et supposant que la plus courte est celle qui arrive à son ancêtre en premier.
10. Récupérez la plus longue chaîne d'une liste de chaînes avec un ancêtre commun connu en construisant des diffs, puis en utilisant `maximum :: Ord a => [a]->a` qui récupère l'élément maximum d'une liste, puis en allant chercher la nouvellle liste dans le diff retourné.

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" en début de hash, 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 calcul. [5] Moins intuitif, cela veut aussi dire qu'un mineur à intérêt à interagir rapidement avec d'autres mineurs suffisamment rapide...

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 bon marché à cette puissance de calcule.

1 À faire chez soi :

Exercice 3 (Table de hashage).
Enfin on peut utiliser Hash pour construire une table de hashage. Une table de hashage est un dictionnaire dont l'implémentation naïve fonctionne ainsi : il est implémenté par un tableau de taille 2^n contenant des valeurs, lorsque l'on cherche la valeur associée à la clef c, on hash cette clé, on récupère les n premiers bits et on va à la case associée.

Le problème est le cas où il y a un clash entre deux clef dont les hashs comportent les même n premiers bits... Si le tableau est beaucoup plus grand que le nombre de données à stocker à la fois [6], on peut s'attendre à ce que ça n'arrive pas, sinon il faut utiliser des stratégies plus complexes pour agrandir le tableau et le réorganiser tout en gardant un accès constant aux données et une insertion rapide.

Les tables de hashages sont des structures éminemment mutables, et donc très difficile à implémenter en Haskell (où tout est immutable pas défaut). On verra peut être la bonne version plus tard, mais pour l'instant on va faire une version non mutable et donc pas optimal du tout (à part dans les cas où le compilateur pourra optimiser, on aura une liste au lieu d'une table...).

1. écrire le type suivant :

   ```
   data HashTable val = HashTable Int [Maybe val]
   ```

   L'entier fournit est le n donné précédemment. Remarquez que le type `cle` n'apparaît pas car il est utilisé pour savoir dans quelle case aller grâce à son hash.

2. Écrire une fonction

   ```
   rechercherParCle :: (Hashable cle) => HashTable val -> Maybe val
   ```

   qui hash la clef, et rend la valeur potentiellement associée.

3. Écrire une fonction

   ```
   rechercherCle :: (Hashable cle) => HashTable val -> Bool
   ```

   qui répond vrais si la case trouvée n'est pas un `Nothing`.

4. Écrire une fonction

   ```
   modifier :: (Hashable cle) => HashTable val -> cle -> val -> HashTable val
   ```

   qui insère la valeur dans la case correspondante, sans vérifier s'il y a quelque chose, le problème est que l'on a peut-être modifier une valeur qui est sur une clé avec le même début de hash...

5. Écrire une fonction

   ```
   insererM :: (Hashable cle, Eq val) =>
               HashTable val -> cle -> val -> Maybe (HashTable val)
   ```

6. De l'ordre du carré...

qui n'insère l'élément que s'il n'y a pas déjà quelque chose dans la case en question ou qu'il s'agit de la même chose.

Pour pouvoir gérer les conflits, on considère maintenant une liste chaînée dans chaque case :

6. écrire le type suivant : [7]

   ```
   data HashTable val = HashTable Int [[(Hash,val)]]
   ```

7. Écrire une fonction

   ```
   rechercheParCle :: (Hashable cle) => HashTable val -> Maybe val
   ```

   qui hash la clef, récupère les n bits de poids faible, récupère la liste présente dans la case associée, puis trouve la valeur associée au bon hash.

8. Écrire

   ```
   inserer :: (Hashable cle) => HashTable val -> cle -> val -> HashTable val
   ```

9. Écrire

   ```
   supprimer :: (Hashable cle) => HashTable val -> cle -> HashTable val
   ```

7. Le `Hash` si dessous est le type `Int`