Principes de Programmation
TP3

11 février 2019

Exercice 1 (Show, Read, Eq, Ord et Enum).
On rappel qu'un prédicat logique (cf. TP1) est définit par :

```
data PredClos =   VRAIS | FAUX  | OU PredClos PredClos
                | ET PredClos PredClos   | NON PredClos
```

1. Dérivez automatiquement les instanciations de `Show`, `Read`, `Eq` et `Ord`. à l'aide du mot clef `deriving` après le datatype.
2. Ouvrez un interpréteur *ghci* dans un terminal (dont le dossier courant est celui du TP).
3. Chargez le TP dans ghci à l'aide de la commande `:load votreTP.hs`
4. Tapez `:t show`, puis essayez la sur `show (ET (OU (VRAIS) FAUX) FAUX)`.
5. Tapez `:t read`, puis essayez la sur `read "ET (OU (VRAIS) FAUX) FAUX"`, qu'avez vous fait ?
6. Que se passe-t-il sur l'exécution de la commande `FAUX < VRAIS` dans *ghci* ?
7. Inversez les positions de `FAUX` et `VRAIS` dans la définition de `Pred` et re-chargez le TP dans *ghci*. Que se passe-t-il maintenant sur `FAUX < VRAIS` ?

Dans le TP2, un état d'un automate est définit par :

```
data EtatD = Noeud Bool (Char -> EtatD)
```

Si l'on veut faire de l'algorithmique un peu plus complexe sur les automates, il faut pouvoir s'apercevoir que l'on est revenu sur un état déjà visité.

8. Essayez de dériver `Eq` sur cette définition.

Le compilateur n'y arrive pas, car il doit savoir si deux états `Noeud True f` et `Noeud True g` sont égaux. Pour ça il faudrait savoir si `f` et `g` sont égaux, ce qui n'est pas vérifiable dans le cas général.

9. Changez la définition de `EtatD` pour qu'il lui soit associé un nom (que l'on supposera unique) et écrivez l'instanciation de l'identité.
10. Dans ghci, tapez `['a'..'k']`. Qu'obtient-on ?

11. Dans ghci tapez `let entre x y = [x..y]` puis regardez son type avec `:t entre`.
12. La type-class `Enum` est un classe importante, regardez sa doc restreinte [1] avec `:i Enum`. Il s'agit des types qui peuvent s'injecter dans les entiers, avec, notamment, un successeur et un prédécesseur.
13. Essayez de dériver `Enum` sur `PredClos`, puis écrivez un datatype `data ValVerite = Vrais|Faux` et dérivez `Enum`.

En fait `Enum` ne peut être dérivé que pour une énumération finie, autrement il faut l'implémenter. La classe `Bounded` qui définit un plus petit élément `minBound::(Bounded a)=>a` et `maxBound::(Bounded a)=>a` fonctionne de la même manière.

14. (difficile) Ajoutez les puits à la définition de `EtatD` et écrivez une fonction qui transforme tous les états qui sont *de facto* des puits (càd qu'ils pointent toujours sur eux-mêmes quelque soit la lettre entre 'a' et 'z') en puits syntaxiques [2] `Puits nom b f`.

1 À faire chez soi :

Exercice 2 (Arbre Préfixes).
On définit les arbres de préfixes ainsi :

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

Contrairement à ce que son nom pourrait faire penser, un arbre préfixe est plus proche d'un automate que d'un arbre : il permet, à partir d'un mot appelé "clé", de récupérer un objet de type `a`. Les clés sont les mots qui terminent sur une feuille lorsque l'on suit les transitions, l'objet associé est celui au bout le la feuille.

1. écrivez `valide :: PTree a -> String -> Bool` qui cherche si le mot est bien une clé,
2. écrivez `rechercheParClef :: PTree a -> String -> a` qui récupère l'élément avec un mot dont le préfixe est une clé valide (on retournera une erreur sinon),
3. (difficile) écrivez `parser` qui utilise un arbre préfixes pour lire une phrase, rendant une liste des éléments (de types `a`) ; en effet, puisque les clefs sont des préfixes, une phrase ne peut être découpée qu'en une seule suite de clefs (plus éventuellement quelques lettres à la fin),
4. Instanciez `Functor` avec `Ptree`.
5. donnez le type attendu de `rechercheParObjet`, qui recherche si on objet donné est dans l'arbre (et retourne un booléen).

Pour pouvoir le calculer, il faut pouvoir énumérer tous les charactères. Pour ca on utilise les types classes Enum et Bounded.

1. Il y a une doc plus complète sur internet
2. Voire la seconde correction du TP2.

6. `Enum` permet d'énumérer tous les éléments d'un même type entre un minimum, et potentiellement un maximum : dans *ghci*, tapez `[2..10]`, puis `['a'..'Z']`, puis `[True..]`
7. `Bounded` permet de récupérer le premier et le dernier élément d'un certain type. Dans *ghci*, tapez `minBound::Int`, puis `maxBound::Bool`, puis `[minBound..maxBound]::[Char]`
8. (difficile) écrivez `rechercheParObjet`,
9. (difficile) écrivez `toList` qui donne la liste des éléments apparaissant l'arbre,
10. (difficile) imstenciez `Eq a => Eq PTree a`.

Pour la question (2), on renvoit une erreur si on ne trouve pas la clé dans l'arbre. Pour la gestion d'erreur, on utilise la structure suivante :

```
data Maybe a = Nothing | Just a
```

11. écrivez `rechercheParClef :: PTree a -> String -> Maybe a` qui récupère l'élément dans un `Just` si la clé est valide et retourne `Nothing` sinon.

Exercice 3 (Quand P2P rencontre bitcoins [3] (Bonus)).
Les arbres de Merkle sont définis comme des sommes des arbres binaires avec des hashs systématiques.

```
import Data.Hashable
type Hash = Int
type Sel = Int
data Hashable a => MerkleTree a =  Feuille a
                                 | Noeud 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 par :

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

avec `hashWithSalt` comme définition minimale.

1. Écrire une fonction qui vérifie que l'arbre est correcte.
2. On définit la classe `Hfunctor` comme des foncteurs uniquement applicable sur des hashables : [4]

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

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

Instanciez `Functor` avec `MerkleTree`

3. Écrire une fonction qui prend deux noeuds et crée l'arbre correct avec ces deux noeuds comme fils.
4. Implémentez `Eq` en supposant que deux arbres différents n'ont jamais le même Hash.

Les blockchains sont définis de façon assez semblable :

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

Chaque élément de type `a` doit être hashable, de sorte à ce que le premier argument de type `Hash` est de hash de la somme des hash de a, et du hash que l'on peut trouver dans son fils, [5] 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.

5. Écrire une fonction qui vérifie que la chaîne est correcte.
6. Instanciez `HFunctor` avec `Blockchain`
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 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.

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 [6] 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 :

11. Implémentez `Eq` en supposant que deux chaînes différents n'ont jamais le même Hash.
12. Implémentez `Ord` en utilisant l'ordre préfixe sur la chaîne.

5. ou 0 si le fils est une feuille.
6. L'ajout d'un bloque à la chaîne est appelé transaction car il contiens les informations sur un groupe de transferts de bitcoin.

13. Récupérez la plus longue chaîne de la liste.

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. [7] Moins intuitif, cela veut aussi dire qu'un mineur à intérêt à interagir rapidement avec d'autres mineurs suffisamment rapide...

Juste pour s'amuser, voici un autre petit exo intéressant (non utilisé pour bitcoin) :

14. Écrire une autre implémentation de `Ord` qui cette fois 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.
15. Écrire une fonction de sélection `select` qui prend une liste de chaînes l, et renvoi la plus grandes de ces extensions.

Enfin on peut utiliser Hash et arbres préfixes pour construire une table de hashage :

16. Changer la définition d'arbre préfixe en enlevant la fonction de transfert et en ne mettant que deux fils connus, correspondant au 0 (fils gauche) et au 1 (fils droit) successif du hash.
17. Écrire `valide :: HTable a -> Hash -> Bool`
et `recherche :: Hashable a => HTable a -> Hash -> a`.

7. Surtout si on a pas un accès particulièrement peu cher à cette puissance de calcule.