TD5 - Les typeclasses

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2021

Dans ce TD nous allons manipuler les *typeclasses* et le principe d'instanciation associé.

Exercice 1 : encodage JSon

Dans ce premier exercice, nous développons un protocole simple de sérialisation en JSon inspiré (dans une version très simplifiée) de la bibliothèque *aeson* (cf. https://hackage.haskell.org/package/aeson).

Question 1.1 : représentation

Une donnée JSon (prononcez *"djay-zone"*), pour *Javascript Object Notation*, est un chaîne de caractère unicode pour représenter des données dites semi-structurées. Dans cette première question, on souhaite décrire le format JSon en utilisant le système de types de Haskell, sous la forme d'un type `JSon` selon les spécifications suivantes :

Les types de base des données JSon sont :

- (constructeur `JString`) les chaînes de caractères, que l'on représentera par le type `Text`
- (constructeur `JNumber`) les nombres, que l'on représentera par le type `Double`
- (constructeur `JBool`) les booléens `True` et `False`
- (constructeur `JNull`) la valeur `null` de javascript

On trouve ensuite deux types structurés :

- (constructeur `JArray`) qui représente une séquence de données JSon, que l'on représentera avec les séquences (type `Seq`) de la bibliothèque `containers`.
- (constructeur `JObject`) qui représente une table d'association entre des clés représentées par des chaînes de caractères et des valeurs de type `JSon`. On utilisera une `Map` (de `containers`) pour ces valeurs.

Définir le type `JSon` introduisant les constructeurs et représentations ci-dessus. On utilisera la dérivation automatique pour les classes `Show`, `Eq` et `Ord`.

Utiliser le type `JSon` pour encoder le document suivant :

```
{ "nom": "Peyton Jones"
 , "prenom": "Simon"
 , "age": 62
 , "travail" : { "rôle": "Principal Researcher"
               , "société": "Microsoft Research"}
 , "contacts" : [ { "mél.": "simonpj@microsoft.com"}
                 ,{ "tél.": "+44 1223 479 848"}]}
```

On utilisera aussi les deux fonctions utilitaires ci-dessous.

```
mkArray :: [JSon] -> JSon
mkArray = JArray . S.fromList

mkObject :: [(Text, JSon)] -> JSon
mkObject = JObject . M.fromList
```

Question 1.2 Encodage vers JSon

On définit la *typeclass* suivante :

```
class Encode a where
  toJSon :: a -> JSon
```

Les types instances de cette *typeclass* doivent expliquer leur encodage en JSon. L'unique loi prévue pour cette *typeclass* est la suivante :

```
-- Injectivité
law_Encode_inj :: (Encode a, Eq a) => a -> a -> Bool
law_Encode_inj x y = (x /= y) ==> (toJSon x /= toJSon y)
```

avec :

```
infixr 5 ==>
(==>) :: Bool -> Bool -> Bool
True ==> False = False
_ ==> _ = True
```

Définissons des instances de cette classe pour quelques types numériques :

```
instance Encode Double where
  toJSon :: Double -> JSon
  toJSon = JNumber

instance Encode Int where
  toJSon :: Int -> JSon
  toJSon = JNumber . fromIntegral
```

Remarque : pour pouvoir (ré-)expliquer les signatures dans les instances, ce qui est redondant mais souvent utile aux lecteurs du programme, il faut ajouter l'extension suivante : `{-# LANGUAGE InstanceSigs #-}`

```
>>> toJSon (42 :: Int)
JNumber 42.0

>>>  toJSon 3.14
JNumber 3.14

>>> law_Encode_inj 4.2 3.0
True
```

Définir les instances suivantes :

- unit `()` se convertit en `JNull`
- les booléens (type `Bool`) se convertissent en `JBool`
- les chaines (type `Text`) se convertissent en `JString`
- les séquences (type `Seq`) se convertissent en `JArray`, de même que les listes
- les maps (type `Map`) se convertissent en `JObject` en supposant que le type des clés instancie `ShowText` (cf. ci-dessous). On fera la même chose pour les listes d'associations de type `[(k,a)]`.

avec :

```
class ShowText a where
  showText :: a -> Text

instance ShowText Text where
  showText :: Text -> Text
  showText = id
```

Question 1.3.

Soit les types suivant :

```haskell
data Person = Person {
  name :: Text
  , firstName :: Text
  , age :: Int
  , work :: Work
  , contacts :: Seq Contact
  } deriving (Show, Eq)

data Work = Work { company :: Text, position :: Text}
  deriving (Show, Eq)

data Contact = Contact { conType :: ContactType, contInfo :: Text }
  deriving (Show, Eq)

data ContactType = Phone | Email
  deriving (Show, Eq)
```

Décrire la traduction en `JSon` de ce type, en se référant à l'exemple en début d'exercice.

Remarque : avec la définition suivante

```haskell
spj :: Person
spj = Person "Peyton Jones" "Simon" 62 (Work "Microsoft Research" "Principal Researcher")
  (S.fromList [(Contact Email "simonpj@microsoft.com")
              ,(Contact Phone "+44 1223 479 848")])
```

On devrait avoir le résultat qui suit :

```
>>> toJSon spj == personJSon
True
```

Exercice 2

Dans cet exercice, nous nous intéressons au procédé inverse, qui consiste à générer des valeurs Haskell à partir d'encodages JSon.

Question 2.1.

Définir une *typeclass* `Decode` qui effectue, via une méthode `fromJSon`, l'opération complémentaire de `Encode` et `toJSon`.

Définir des fonctions de conversion qui permettent de décoder les types de base : `JNull`, `JBool`, `JNumber` et `JString`. Un exemple d'une telle fonction est `decodeBool` qui décode un booléen, de sorte que par exemple :

```
>>> decodeBool (JBool True)
Just True

>>> decodeBool (JNumber 42)
Nothing
```

En déduire les instances associées de la classe `Decode` qui vous semblent les plus intéressantes. On ajoutera un décodage pour les `JArray` en séquences, en prenant en premier argument la conversion désirée pour les éléments du tableau.

Question 2.2.

Définir les instances nécessaires à la traduction d'un `JSon`, comme `personjson` en une valeur de type `Person`, comme `spj`.

Remarque : il sera utile de définir des fonctions auxiliaires de décodage pour les différents types concernés, par exemple `decodeContact` pour les contacts, etc.

Question 2.3

Pouvez-vous définir en Haskell les propriétés suivantes ?

— le décodage de l'encodage en JSon de `x` est égal à `x`
— l'encodage JSon du décodage d'une chaîne `s` est égal à `s`.

Donner des exemples d'utilisation des propriétés (sur les entiers et les personnes).

Peut-on effectuer des tests QuickCheck associés ?

Une petite **remarque** pour finir. La bibliothèque *aeson* automatise la plupart des définitions que nous avons effectuées ci-dessus, mais dans ce TD nous avons levé une partie du voile sur l'envers du décors, largement basé sur la notion de *typeclass*.