TD6 - Structures algébriques

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2021

Dans ce TD, nous allons manipuler les structures algébriques `Functor`, `Semigroup` et `Monoid` (avec `Foldable`).

Exercice 1 : des foncteurs et des lois

On rappelle ci-dessous les lois de préservation que les instances de `Functor` doivent respecter :

```
-- en théorie :  fmap id = id
-- en pratique :
law_Functor_id :: (Eq (f b), Functor f) => (a -> b) -> f a -> Bool
law_Functor_id g x = (fmap g (id x)) == (id (fmap g x))

-- en théorie : fmap (g . f) = fmap g . fmap f
-- en pratique :
law_Functor_comp :: (Functor f, Eq (f c)) => (b -> c) -> (a -> b) -> f a -> Bool
law_Functor_comp g f x = fmap (g . f) x == (fmap g . fmap f) x
```

avec dans le prélude :

```
id :: a -> a
id x = x


infixr 9 .
(.) :: (b -> c) -> (a -> b) -> a -> c
g . f x = g (f x)
```

Question 1.1

On rappelle le foncteur pour le type `List` utilisé en cours, et isomorphe aux listes Haskell.

```
data List a = Nil | Cons a (List a) deriving (Show, Eq, Ord)

listMap :: (a -> b) -> List a -> List b
listMap _ Nil = Nil
listMap f (Cons x xs) = Cons (f x) (listMap f xs)

instance Functor List where
  fmap = listMap
```

Montrer, par raisonnement équationnel, que les lois de préservation sont bien respectées par ce foncteur.

Question 1.2.

On rappelle le type `Pair a b` isomorphe aux couples `(a, b)`.

```
data Pair a b = Pair a b
  deriving (Show, Eq)
```

On souhaite instancier `Functor` pour cette construction.

— Quelle devait-être la valeur de `fmap (+1) (Pair 1 2)` ?

Donner les définitions permettant de réaliser cette évaluation (on pourra considérer `(Pair e a)` plutôt que `(Pair a b)` pour faciliter les constructions).

En complément (à la maison), on démontrera les lois de préservation pour ce foncteur.

Question 1.3.

L'expression de type `a -> b` correspond aux fonctions de `a` vers `b`. En Haskell `(->)` est un constructeur de type à part entière, donc on peut aussi écrire : `(->) a b`.

— Quel est le *kind* de `(->)` ?
— Pouvez-vous en déduire une instance de foncteur ? Si oui proposer une définition du `arrowMap` correspondant.
— Que deviennent les lois de préservation pour ce foncteur ?

Soit la définition suivante :

```
newtype Reader e a = Reader (e -> a)
```

Est-il possible de transformer `Reader` en monoïde ?

Exercice 2 : Les listes non-vides semi-groupées

On introduit un type pour représenter les listes non-vides :

```
data NonEmptyList a = NECons a [a]
```

(une définition équivalente existe dans le module `Data.List.NonEmpty`).

On rappelle la *typeclasse* des semi-groupes :

```
class Semigroup a where
  (<>) :: a -> a -> a
```

Question 2.1.

Instancier la *typeclasse* `Semigroup` pour le type `NonEmptyList a` (pour tout type `a`).

Question 2.2.

Rappeler le code Haskell de la loi d'associativité pour les semi-groupes (avec `Eq`). Montrer que les listes non-vides respectent cette loi.

Est-il possible d'instancier `Monoid` ?

Question 2.3.

Il existe dans `Data.Semigroup` un type `Max`, instance de `Semigroup`, et tel que, par exemple :

```
>>>  (Max 42) <> (Max 17)
Max {getMax = 42}
```

Expliquer en quoi ce semi-groupe ne peux être un monoïde. En supposant que l'on ne s'intéresse qu'aux entiers naturels, peut-on définir un tel monoïde `NatMax` ? Si oui effectuer cette définition et les instances associées.

Exercice 3 : Monoides et arbres foldables

Soit le type suivant :

```
data Tree a =
  Tip
  | Node a (Tree a) (Tree a)
```

```
  deriving (Show, Eq)

exTree :: Tree Integer
exTree = Node 17 (Node 24 (Node 12 (Node 9 Tip Tip) Tip)
                          (Node 42 Tip Tip))
         (Node 19 Tip (Node 11 Tip Tip))
```

Question 3.1.

Définir une fonction de *fold* pour `Tree a` avec la signature suivante :

```
treeFoldMap :: Monoid m => (a -> m) -> Tree a -> m
```

et telle que, par exemple :

```
>>> treeFoldMap NatMax exTree
NatMax {getNatMax = 42}

>>>  treeFoldMap Sum exTree
Sum {getSum = 134}
```

Remarque : le type `Sum` (dans `Data.Monoid`) est un monoïde pour l'addition. Par exemple :

```
>>> (Sum 42) <> (Sum 33) <> (Sum 9)
Sum {getSum = 84}
```

Question 3.2.

En déduire une instance de la *typeclasse* `Foldable`, donc un extrait de la définition est donné ci-dessous:

```
class Foldable t where
   foldMap :: Monoid m => (a -> m) -> t a -> m
   -- ... etc ...
```

Pouvez-vous, en utilisant `foldMap`, retourner la liste des étiquettes de l'arbre ? Si oui quel est l'ordre naturellement retournée pour cette liste ?

Question 3.3.

Il est possible d'instancier `Foldable` à partir :

- soit de `foldMap` comme nous l'avons fait précédemment
- soit de `foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b`

Donner une définition de `foldMap` (disons `myFoldMap`) en fonction de `foldr`.

Donner ensuite une définition de `foldr` (disons `myFoldr`) en fonction de `foldMap`.

Pour cela, on pourra s'inspirer d'une variante *"interne"* de `Reader` :

```
newtype Intern b = Intern (b -> b)
```

Remarque : cette dernière question (définir `foldr` en fonction de `foldMap`) est plutôt difficile, donc facultative.