TD8 - Monades

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2021

Dans ce TD, nous allons manipuler les *monades* tant d'un point de vue théorique que d'un point de vue pratique.

Exercice 1 : des monades et des lois

Nous avons vu en cours les lois gouvernant la *typeclasse* `Monad` mais en utilisant comme base l'opérateur de composition (de gauche à droite) :

```
>>> :t (>=>)
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
```

On rappele ces lois :

— pour tout `a, b` et `f::a -> m b` on a :

(1) `pure >=> f = f` (*identité gauche*)

(2) `f >=> pure = f` (*identité droite*)

— pour tout `a, b, c, d` et `f:: a -> m b`, `g:: b -> m c` et `h:: c -> m d` on a :

(3) `(f >=> g) >=> h = f >=> (g >=> h)` (*associativité*)

Question 1.1.

Implémenter la vérification des lois en Haskell avec les propriétés `law_leftIdentity`, `law_rightIdentity` et `law_associativity` correspondant aux lois ci-dessus.

Remarque : on fera attention à comparer les valeurs avec `==`, l'égalité sur les fonctions n'est pas définie (ni définissable).

Question 1.2.

Proposer une implémentation de l'opérateur de composition "fish" `>=>` à partir de *bind* `>>=`. En déduire une variante des lois des monades s'appliquant directement à `>>=`, que l'on nommera `law_leftIdentity'`, `law_rightIdentity'` et `law_associativity'`.

Question 1.3.

La monade la plus simple est `Identity`, définie ainsi :

```haskell
newtype Identity a = Identity { runIdentity :: a }
  deriving (Show, Eq)

instance Functor Identity where
  fmap g (Identity x) = Identity (g x)

instance Applicative Identity where
  pure = Identity
  (Identity g) <*> (Identity x) = Identity (g x)
```

```haskell
instance Monad Identity where
  (Identity x) >>= f = f x
```

Montrer que les lois des monades (celles de la question 1.2 car elles portent plus directement sur la définiti ci-dessus), sont bien respectées.

On pourra utiliser la proposition suivante, dite d'*inversion* de `runIdentity` :

Si `v :: Identity a` alors `v = Identity (runIdentity v)`

Exercice 2 : Des états dans tous leurs états

Nous avons vu dans les précédents cours les *readers* (encapsulation du type flèche `(-> r)`) et *writers* (encapsulation du type pair `(,) e`). On peut voir le *reader* comme une mémoire en lecture seule, et le *writer* comme une mémoire en écriture seule, ce qui explique leurs noms peut-être contre-intuitifs. Ces deux constructions sont monadiques (cf. `Reader` et `Writer` dans `Control.Monad`).

Cette interprétation se généralise en la représentation d'une mémoire en lecture *et* en écriture avec l'encapsulation d'une signature de la forme :

```
s -> (a, s)
```

— où le `s` à gauche de la flèche représente le type de l'état manipulé (donc la représentation de la mémoire) en entrée d'une opération, et
— dans `(a, s)` le `a` est le type de la valeur retournée par une opération donnée, et `s` en sortie représente les modifications éventuelle de l'état

Bien sûr, dans tous les cas nous parlons d'une sorte de «simulation» de mémoire, toujours en fonctionnel pur. Pour illustrer notre propos, nous allons prendre l'exemple probablement incontournable de la simulation d'une *pile mémoire*.

Les deux opérations élémentaires sur les piles sont :

— `pop` pour dépiler la tête de pile (fonction partielle)
— `push` pour empiler une nouvelle valeur

Pour représenter notre pile, nous allons simplement utiliser une liste Haskell. Voici un code purement fonctionnel qui simule une pile impérative :

```haskell
type Stack a = [a]

pop :: Stack a -> (a, Stack a)
pop [] = error "Pile vide"  -- fonction partielle !
pop (x:xs) = (x, xs)

push :: a -> Stack a -> ((), Stack a)
push x xs = ((), x:xs)
```

On peut par exemple effectuer une série d'opérations en séquence :

```haskell
>>> let (_, s1) = push 1 []
>>> let (_, s2) = push 2 s1
>>> let (_, s3) = push 3 s2
>>> pop s3
(3,[2,1])
```

A la fin de cette séquence, le `pop` retourne `3` et la pile restante est bien `[2,1]`. Nous pouvons également prendre l'exemple d'un compteur avec incrément et décrément.

```haskell
newtype Counter = Counter Integer
  deriving Show
```

```haskell
incr :: Counter -> (Integer, Counter)
incr (Counter c) = (c, Counter (c+1))

decr :: Counter -> (Integer, Counter)
decr (Counter c) = (c, Counter (c-1))
```

Par exemple :

```
>>> let (_, c1) = incr (Counter 40)
>>> let (_, c2) = incr c1
>>> let (_, c3) = incr c2
>>> decr c3
(43,Counter 42)
```

Question 2.1 : mémoire composée

On introduit une mémoire un peu plus complexe, composée d'une pile et d'un compteur, en se basant sur le type suivant :

```haskell
data MyMem a = MyMem { getStack :: Stack a
                     , getCounter :: Counter }
             deriving Show
```

Proposer une implémentation pour les opérations `mpush`, `mpop`, `mincr` et `mdecr` qui effectue les mêmes opérations que ci-dessus mais sur cette mémoire composée.

Question 2.2 : la monade State

Le style de programmation illustré précédemment, qui consiste à simuler de l'"impératif" (en fait plutôt des calculs à état, ou *stateful*) en style fonctionnel, n'est pas très élégant sous cette forme directe. Il est pourtant très utile de gérer une sorte de mémoire qui passe de fonctions en fonctions, autrement dit d'effectuer des calculs en "mémoire partagée".

Heureusement, les opérations que nous avons définies, et plus encore leur composition, peuvent être codées de façon nettement plus élégantes en passant par la monade *state*. Celle-ci n'existe pas directement dans base mais elle est définies dans plusieurs bibliothèques tierces, en particulier `transformers` dont nous reparlerons. Une monade *state* est facile à définir à partir du type suivant :

```haskell
data State s a = State { runState :: (s -> (a, s)) }
```

Voici les *alias* que nous utiliserons pour nos 3 types de mémoire :

```haskell
type StackSt a b = State (Stack a) b

type CounterSt b = State Counter b

type MyMemSt a b = State (MyMem a) b
```

Redéfinir les opérations des questions précédentes avec ces nouvelles représentations.

Question 2.3. Instances

Les réponses à la question précédentes ne semblent pas fournir une grande amélioration, on a même quelque peut complexifié les choses. Cependant, nous allons comprendre l'intérêt de ces variantes dans ce qui suit, c'est-à-dire faire de notre contexte `State s` un contexte monadique.

Définir les instances de `Functor`, `Applicative` et `Monad` pour ce contexte.

Remarque : il n'existe qu'une seule implémentation raisonnable pour toutes les opérations considérées à part *apply* (`<*>`). Pour cette dernière, on effectuera les changements d'états de gauche à droite. De plus, toutes ces implémentations sont presque intégralement dirigées par les types. La seule contrainte importante est que tous les changements d'états soient bien pris en compte.

Question 2.4 : manipulations d'états avec la notation do

Donner les types et/ou les valeurs des occurrences de `???????` dans les exemples suivants :

```haskell
-- stackManip :: ???????
stackManip = do
  pushSt 1
  pushSt 2
  pushSt 3
  x <- popSt
  return x
```

```
>>> runState stackManip []
???????

>>> runState stackManip [4, 5, 6]
???????
```

```haskell
-- counterManip :: ???????
counterManip = do
  incrSt
  incrSt
  incrSt
  x <- decrSt
  return x
```

```
>>> runState counterManip $ Counter 40
???????

>>> runState counterManip $ Counter 0
???????
```

Question 2.5 : mémoire composée en do

Définir une fonction `myMemManip` qui effectue en utilisant la notation `do` les opérations décrites dans `stackManip` et `counterManip` dans cet ordre mais dans le cadre de `MyMemSt a b`. e et donc avec la signature suivante :

```haskell
myMemManip :: MyMemSt Integer (Integer, Integer)
```

Remarque : dans le type `(Integer, Integer)` la première valeur retournée correspond à la manipulation de la pile, et le seconde à la manipulation du compteur).

Donner ensuite les valeurs retournées par les expressions suivantes :

```
>>> runState myMemManip $ MyMem [] (Counter 40)
???????

>>> runState myMemManip $ MyMem [4, 5, 6] (Counter 0)
???????
```

Réécrire finalement la fonction `myMemManip` en traduisant la notation `do` en style monadique «direct» (avec `>>=` etc.)

Exercice 3 : Des compréhensions monadiques pour les séquences

La syntaxe des compréhensions de Haskell est une notation concise et pratique pour créer des listes complexes. Prenons quelques exemples simples :

```
>>> [x*x | x <- [1..4]]
[1,4,9,16]

>>> [(i,j) | i <- [1..4], j <- [i..4]]
```

```
[(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4),(4,4)]

-- avec des gardes

>>> [x*x | x <- [1..8], even x]
[4,16,36,64]

>>> [(i,j,k) | i <- [1..5], j <- [i..5], k <- [j..5], i + j == k]
[(1,1,2),(1,2,3),(1,3,4),(1,4,5),(2,2,4),(2,3,5)]
```

Il est possible d'utiliser l'interprétation non-déterministe des listes pour obtenir des résultats similaires avec des notations monadiques. Pour illustrer ce fait, nous allons construire un petit système de compréhensions monadiques pour les séquences. Pour cela, nous utilisons le *wrapper* de type suivant (car le contexte `Seq` est déjà monadique) :

```
data MSeq a = MSeq (Seq a)
  deriving (Show, Eq)
```

Pour manipuler les séquence encapsulées dans `MSeq`, on utilisera uniquement les fonctions suivantes :

- Les constructeurs `Empty` et `x :<| xs` (avec le premier élément `x` et le reste de la séquence `xs`).
- La fonction `S.fromList` pour créer une séquence à partir d'une liste.
- L'opérateur `><` de concaténation de deux séquences.

Les imports utilisés sont les suivants :

```
import Data.Sequence (Seq (..), (><))
import qualified Data.Sequence as S
```

On pourra aussi utiliser la fonction suivante pour améliorer la lisibilité des exemples :

```
mkMSeq :: [a] -> MSeq a
mkMSeq = MSeq . S.fromList
```

On commence par donner les contextes fonctoriels.

```
instance Functor MSeq where
  fmap _ (MSeq Empty) = MSeq Empty
  fmap g (MSeq (x :<| xs)) =
    let (MSeq xs') = fmap g (MSeq xs)
        in MSeq ((g x) :<| xs')

instance Applicative MSeq where
  pure v = MSeq (S.singleton v)

  (MSeq Empty) <*> _ = MSeq Empty
  (MSeq (g :<| gs)) <*> (MSeq xs) =
    let (MSeq xs') = fmap g (MSeq xs)
        (MSeq ys) = (MSeq gs) <*> (MSeq xs)
    in MSeq (xs' >< ys)
```

On a par exemple :

```
>>> fmap (*2) (mkMSeq [1, 2, 3, 4])
MSeq (fromList [2,4,6,8])

>>> (+) <$> (mkMSeq [1, 2, 3]) <*> (mkMSeq [10, 20, 30])
MSeq (fromList [11,21,31,12,22,32,13,23,33])

>>> (mkMSeq [(+),(*)]) <*> (mkMSeq [1, 2, 3]) <*> (mkMSeq [10, 20, 30])
MSeq (fromList [11,21,31,12,22,32,13,23,33,10,20,30,20,40,60,30,60,90])
```

Ce sont donc des implémentations similaires à ce que nous avons défini en cours sur les listes "maison", et ce qui correspond aux foncteurs et applicatifs des listes (sauf qu'ici les séquences sont finies).

Question 3.1

Définir une fonction `bindMSeq :: MSeq a -> (a -> MSeq b) -> MSeq b` et l'utiliser pour instancer `Monad` en vous inspirant des exemples suivants.

```
>>> (mkMSeq [1..4]) >>= \x -> return (x*x)
MSeq (fromList [1,4,9,16])

>>> (mkMSeq [1..4]) >>= (\i -> (mkMSeq [i..4]) >>= (\j -> return (i,j)))
MSeq (fromList [(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4),(4,4)])
```

Question 3.2.

Réécrire les exemples de la question précédente en utilisant la notation `do`.

Question 3.3.

Les exemples de compréhensions de listes suivant mettent en jeu une notion de *garde* :

```
>>> [x*x | x <- [1..8], even x]
[4,16,36,64]
```

```
triplets :: Integer -> [(Integer,Integer,Integer)]
triplets n = [(i, j, k) | i <- [1..n], j <- [i..n], k <- [j..n], i + j == k]

-- >>> triplets 5
-- [(1,1,2),(1,2,3),(1,3,4),(1,4,5),(2,2,4),(2,3,5)]
```

Dans le premier exemple on ne retient que les valeurs paires de `x`. Dans le deuxième exemple la garde est `i + j == k`. En style monadique, on peut réécrire ces exemples de la façon suivante :

```
>>> do { x <- [1..8] ; guard (even x) ; return (x*x) }
[4,16,36,64]
```

```
mtriplets :: Integer -> [(Integer, Integer, Integer)]
mtriplets n = do
  i <- [1..n] ; j <- [i..n] ; k <- [j..n]
  guard (i + j == k)
  return (i, j, k)

-- >>> mtriplets 5
-- [(1,1,2),(1,2,3),(1,3,4),(1,4,5),(2,2,4),(2,3,5)]
```

La fonction `guard` définie dans le module `Control.Applicative` est la suivante :

```
guard :: Alternative m => Bool -> m ()
guard True  = pure ()
guard _ = empty
```

Lorsque la garde est fausse, l'injection de `empty` va se propager à toute la chaine de *bind*s et donc au filtrage de la solution courante, contrairement à l'injection d'une valeur "vide" (le *unit* `()`) qui n'interfère pas avec le chaînage. Autrement dit : `empty >>= <chaine>` va produit `empty` alors que `pure () >>= <chaine>` va produire `<chaine>`.

Au-delà de cette définition, il suffit pour pouvoir utiliser `guard` comme dans les exemples avec la monade liste d'instancier la *typeclasse* suivante (également définie dans `Control.Applicative`) :

```
class Applicative f => Alternative (f :: * -> *) where
   empty :: f a
   (<|>) :: f a -> f a -> f a
```

accompagnée des lois suivantes :

- `empty` est un élément neutre

```
empty <|> u  =  u
u <|> empty  =  u
```

— `<|>` (dite *alternative*) est associative

```
u <|> (v <|> w)  =  (u <|> v) <|> w
```

L'intuition derrière `Alternative` correspond à la production de solutions possibles à des problèmes donnés. Dans ce contexte, le rôle de `empty` est d'indiquer l'absence de solution. Une expression `u <|> v` s'interprète comme un *alternative* entre la solution `u` ou, si elle n'est pas valable, la solution `v` (avec donc une sorte de préférence pour le premier opérande).

Dans le contexte de la monade liste (et donc celle pour `MSeq` puisque nous adoptons la même interprétation), l'alternative correspond bien sûr à la concaténation, c'est à dire, dans un contexte de calcul non-déterministe, à produire d'abord les solutions de la première séquence, puis d'ajouter ensuite les solutions de la séquence de droite. L'absence de solution est bien sûr la séquence vide.

En déduire une instantiation de `Alternative` pour `MSeq`, et en déduire une version `MSeq` des deux exemples donnés ci-dessus pour les listes.

Question subsidiaire : quelle définition proposeriez-vous pour l'instanciation de `Alternative` avec le constructeur `MMaybe` du cours ?