TD7 - Foncteurs applicatifs

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2021

Dans ce TD nous étudions les foncteurs applicatifs, selon la typeclasse suivante :

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

et avec `f` un contexte fonctoriel/applicatif de *kind* `* -> *` (l'opérateur `<*>` se nomme *apply*)

Les lois principales des foncteurs applicatifs sont les suivantes :

- **identité** : `pure id <*> v = v`
- **homomorphisme** : `pure g <*> pure x = pure (g x)`
- **interchange** : `u <*> pure y = pure ($ y) <*> u = pure (\f -> f y) <*> u`
- **fonctorialité** : `f <$> x = pure f <*> x`

Exercice 1 : des applicatifs simples

Dans cet exercice, nous instancions `Applicative` pour deux types «simples».

Question 1.1.

Dans le TD précédent, nous avons redéfini le type *reader* qui encapsule les fonctions. Voici une variante un peu plus intéressante, car on peut récupérer la fonction encapsulée, grâce à l'accesseur `runReader`.

```
newtype Reader e a = Reader { runReader :: (e -> a) }
```

Et nous avons vu qu'il s'agissait d'un contexte fonctoriel :

```
instance Functor (Reader e) where
  -- fmap g (Reader h) = Reader (g . h)
  fmap g r = Reader (g . (runReader r))
```

Définir deux fonction `pureReader` et `applyReader` permettant d'instancier `Applicative` pour `Reader e`. En déduire cette instance.

Donner le résultat des interactions suivantes :

```
>>> let r1 = (+1) <$> (Reader (-1))
>>> :t r1
????
>>> (runReader r1) 5
????
>>> (runReader r1) 10
????

>>> let r2 = (+) <$> (Reader (+3)) <*> (Reader (*100))
>>> :t r2
????
```

```
>>> (runReader r2) 5
????
>>> (runReader r2) 10
????
```

Question 1.2.

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

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

Et dans le TD précédent, on a défini le contexte fonctoriel pour ce type.

```haskell
instance Functor (Pair e) where
  -- fmap :: a -> b -> (Pair e a) -> (Pair e b)
  fmap g (Pair x y) = Pair x (g y)
```

On ne peut pas instancier directement `Applicative` sur `(Pair e)` pour n'importe quel `e`. Donner une raison à cela.

Mais on peut en revanche le faire si on suppose que `e` est un monoïde.

Proposer des fonctions `purePair` et `applyPair` permettant d'instancier `Applicative` et en déduire cette instance.

Donner les résulats des interactions suivantes :

```
>>> (+1) <$> (Pair "x" 41)
????

>>> (+) <$> (Pair "Hello" 39) <*> (Pair " world!" 3)
????
```

Exercice 2 : un applicatif alternatif pour les listes

On a vu en cours que le contexte applicatif pour les listes correspondait, par défaut, à un principe de programmation non-déterministe. On a par exemple :

```
>>> (+1) <$> [1, 2, 3]
[2, 3, 4]

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

>>> (*) <$> [1, 2, 3] <*> [10, 20, 30]
[10,20,30,20,40,60,30,60,90]

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

Une second interprétation est possible : voir les listes comme des conteneurs de valeurs ordonnées. Dans le prélude, cette interprétation se nomme un `ZipList`, et voici quelques exemples :

```
>>> (+1) <$> (ZipList [1, 2, 3])
ZipList {getZipList = [2,3,4]}
```

Pour l'instant on ne voit pas trop de différence (ce qui est normal, il n'existe qu'un foncteur possible), mais continuons :

```
>>> (+) <$> (ZipList [1, 2, 3]) <*> (ZipList [10, 20, 30])
ZipList {getZipList = [11,22,33]}
```

Ici on voit bien la différence, on a une addition membre à membre. Essayons quelques autres exemples :

```
>>> (+) <$> (ZipList [1, 2, 3, 4]) <*> (ZipList [10, 20])
ZipList {getZipList = [11,22]}

>>> (+) <$> (ZipList [1, 2]) <*> (ZipList [10, 20, 30, 40])
ZipList {getZipList = [11,22]}

>>> (ZipList [(+), (*)]) <*> (ZipList [1, 2, 3]) <*> (ZipList [10, 20, 30])
ZipList {getZipList = [11,40]}
```

Question 2.1.

Définir le type `ZList`, isomorphe à `ZipList`, ainsi que le foncteur correspondant, de sorte que par exemple :

```
>>> (+1) <$> (ZList [1, 2, 3])
ZList [2, 3, 4]
```

Question 2.2.

L'implémentation du *apply* est la plus simple. Soit la signature suivante :

```
applyZList :: ZList (a -> b) -> ZList a -> ZList b
```

Dans `applyZList (ZList gs) (ZList xs)`, l'idée est simplement d'appliquer la fonction à la i-ème position dans la liste `gs` à la valeur de même position dans `xs`. Si les listes sont de longueurs différentes, on injecte simplement la liste vide.

Par exemple :

```
>>> applyZList (ZList [(*2), (*3), (*4)]) (ZList [2, 3, 4])
ZList {getZList = [4,9,16]}

>>> applyZList (ZList [(*2), (*3)]) (ZList [2, 3, 4])
ZList {getZList = [4,9]}

>>> applyZList (ZList [(*2), (*3), (*4)]) (ZList [2, 3])
ZList {getZList = [4,9]}

>>> applyZList (ZList []) (ZList [2, 3, 4])
ZList {getZList = []}

>>> applyZList (ZList [(*2), (*3), (*4)]) (ZList [])
ZList {getZList = []}
```

Question 2.3.

Pour l'implémentation de *pure* pour le type `ZList`, on s'inspire de la loi d'identité des applicatifs : `(pure id) <*> v = v`.

Avant d'instancier la *typeclasse* cette équation devient :

```
(pureZList id) `applyZList` v = v
```

En considérant les exemples suivant :

```
>>> (pureZList id) `applyZList` (ZList [1, 2, 3, 4])
ZList {getZList = [1,2,3,4]}

>>> take 10 $ getZList $ (pureZList id) `applyZList` (ZList [1..])
[1,2,3,4,5,6,7,8,9,10]

>>> take 10 $ getZList $ (pureZList (+1)) `applyZList` (ZList [1..])
[2,3,4,5,6,7,8,9,10,11]
```

En déduire une implémentation de `pureZList :: a -> ZList a`, et finalement l'instanciation de `Applicative` pour `ZList`.

Exercice 3 : des arbres traversables

Dans le TD précédent, nous avons défini un type d'arbres *foldables*.

```
data Tree a =
  Tip
  | Node a (Tree a) (Tree a)
  deriving (Show, Eq)

exTree1 :: Tree Integer
exTree1 = Node 17 (Node 24 (Node 12 (Node 9 Tip Tip) Tip)
                           (Node 42 Tip Tip))
          (Node 19 Tip (Node 11 Tip Tip))

exTree2 :: Tree Integer
exTree2 = Node 18 (Node 24 (Node 12 (Node 8 Tip Tip) Tip)
                           (Node 42 Tip Tip))
          (Node 20 Tip (Node 12 Tip Tip))

treeFoldMap :: Monoid m => (a -> m) -> Tree a -> m
treeFoldMap _ Tip = mempty
treeFoldMap f (Node v l r) = (f v) <> (treeFoldMap f l) <> (treeFoldMap f r)

instance Foldable Tree where
  foldMap = treeFoldMap
```

Dans cet exercice, nous nous intéressons à la typeclasse `Traversable` dont (une partie de) la définition est la suivante :

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence :: Monad m => t (m a) -> m (t a)
  {-# MINIMAL traverse | sequenceA #-}
    -- Defined in ‘Data.Traversable’
```

Dans cette définition, `f` doit être un contexte fonctoriel applicatif de *kind* `* -> *`. Il s'agit de plus d'une extension de `Foldable`, via le paramètre `t` dans la définition (également de *kind* `* -> *`). La méthode la plus importante est `traverse` qui s'apparente à une sorte de `fmap` mais dans un contexte applicatif.

Question 3.1.

Définir le foncteur pour le constructeur de type `Tree`.

En déduire une implémentation de la fonction suivante :

```
traverseTree :: Applicative f => (a -> f b) -> Tree a -> f (Tree b)
```

(le principe est d'enrichir la définition du foncteur pour le faire "passer" au contexte applicatif `f`).

En déduire une instance de `Traversable` pour `Tree`, i.e. rendre les arbres *traversables*.

Question 3.2.

Définir une fonction `evenTree :: Integral a => Tree a -> Maybe (Tree a)` qui :

— accepte les arbres dont tous les noeuds sont des nombres pairs

```
>>> evenTree exTree2
Just (Node 18 (Node 24 (Node 12 (Node 8 Tip Tip) Tip) (Node 42 Tip Tip)) (Node 20 Tip (Node 12 Tip Tip)))
```

— rejette les arbres qui contiennent des nombres impairs

```
>>> evenTree exTree1
Nothing
```

On utilisera le fait que `Tree` est *traversable*, et pour cela on pourra penser à une fonction : `acceptEven :: Integral a => a -> Maybe a`

Et on rappelle :

```
even :: Integral a => a -> Bool

>>> even 42
True

>>> even 17
False
```

Exercice 4 : validation applicative (préparation du TME)

Cet exercice est une préparation au TME sur la validation applicative. Il s'agit d'une des applications pratiques les plus communes du style applicatif. Les réponses à cet exercice sont à placer dans le module `Validation` du projet stack correspondant.

On souhaite mettre en place un système de validation, basé sur le principe de `Either` avec :

— `Left errs` indique une liste d'erreurs (échec de la validation)
— `Right x` est une valeur `x` correctement validée

On considère l'alias de type suivant :

```
type Error = String
```

Question 4.1.

L'idée est de représenter un résultat de validation par le type suivant :

```
Either [Error] a
```

où `a` est le type du résultat attendu, et en cas d'erreur une listes de messages d'erreurs est retournée (pour simplifier, on prend `String` plutôt que `Text`).

L'instance de `Applicative` par défaut pour `Either e` ne permet pas de gérer les erreurs de validation.

Considérons la fonction suivante :

```
checkEven :: (Integral a, Show a) => a -> Either [Error] a
checkEven n | even n = Right n
            | otherwise = Left ["Ce n'est pas un nombre pair: " <> (show n)]
```

Par exemple :

```
>>> checkEven 42
Right 42

>>> checkEven 17
Left ["Ce n'est pas un nombre pair: 17"]
```

Pour l'instant le comportement semble satisfaisant, mais adoptons le style applicatif maintenant :

```
>>> (,) <$> (checkEven 42) <*> (checkEven 38)
Right (42,38)
```

Voici un bon exemple du style applicatif, on peut reconstruire une valeur complexe (ici un couple) en combinant les résultats en style applicatif.

Remarque : le constructeur de couple est bien une fonction en Haskell, on a `(,) :: a -> b -> (a, b)`

Essayons quelques cas invalides :

```
>>> (,) <$> (checkEven 17) <*> (checkEven 38)
Left ["Ce n'est pas un nombre pair: 17"]

>>> (,) <$> (checkEven 42) <*> (checkEven 33)
Left ["Ce n'est pas un nombre pair: 33"]
```

Ce dernier exemple pose problème :

```
>>> (,) <$> (checkEven 17) <*> (checkEven 33)
Left ["Ce n'est pas un nombre pair: 17"]
```

Le `Either` n'accumule pas les erreurs, car ce que l'on voudrait en sortie est plutôt : `Left ["Ce n'est pas un nombre pair: 17", "Ce n'est pas un nombre pair: 33"]`

On définit donc un nouveau type qui encapsule `Either` :

```haskell
newtype Validation e a = Validation (Either e a)
  deriving (Show, Eq)
```

Donner l'implémentation du foncteur `fmap` pour ce type (on travaille toujours sur le type `a` de la construction).

Question 4.2.

On souhaite instancier la *typeclasse* `Applicative` pour `Validation` de la façon suivante :

```haskell
instance Monoid e => Applicative (Validation e) where
  pure = pureVal
  (<*>) = applyVal
```

avec :

```haskell
pureVal :: a -> Validation e a
```

et

```haskell
applyVal :: Monoid e => Validation e (a -> b) -> Validation e a -> Validation e b
```

Définir cette instance en s'inspirant des exemples suivants.

```haskell
type Valid a = Validation [Error] a

valid :: a -> Valid a
valid x = Validation (Right x)

invalid :: String -> Valid a
invalid msg = Validation (Left [msg])

validateEven :: (Integral a, Show a) => a -> Valid a
validateEven n | even n = valid n
               | otherwise = invalid $ "Ce n'est pas un nombre pair: " <> (show n)
```

```
>>> (,) <$> (validateEven 42) <*> (validateEven 38)
Validation (Right (42,38))

>>> (,) <$> (validateEven 17) <*> (validateEven 38)
Validation (Left ["Ce n'est pas un nombre pair: 17"])
```

```
>>> (,) <$> (validateEven 17) <*> (validateEven 33)
Validation (Left ["Ce n'est pas un nombre pair: 17","Ce n'est pas un nombre pair: 33"])
```

Question 4.3.

Le projet du TME est un carnet de contacts (très simplifié) où chaque contact est représenté par un *record* avec les champs `nom`, `prenom`, `email` et `tel`.

Dans cette question, nous allons nous intéresser à la validation du nom (et en fait du prénom). Dans le TME il s'agira de valider tous les autres champs.

Ce que l'on souhaite, pous le `nom` c'est qu'il soit représenté par une chaîne non-vide, composée d'une lettre majuscule en tête, suivie d'une liste de lettres minuscules. Il faudrait une validation un peu plus "intelligente" en pratique, mais pour illustrer note propos cela suffit largement.

Voici les fonctions de validation correspondantes :

```
validateNonEmpty :: String -> Valid ()
validateNonEmpty [] = invalid "La chaine est vide"
validateNonEmpty xs = valid ()

validateMinuscule :: Char -> Valid Char
validateMinuscule ch
  | (ch >= 'a' && ch <= 'z') = valid ch
  | otherwise = invalid ("Ce n'est pas une miniscule : "  <> (show ch))


validateMajuscule :: Char -> Valid Char
validateMajuscule ch
  | (ch >= 'A' && ch <= 'Z') = valid ch
  | otherwise = invalid ("Ce n'est pas une majuscule : "  <> (show ch))
```

Compléter la fonction suivante :

```
validateNom :: String -> Valid String
validateNom str = (:) <$> (vNE *> vH) <*> vT
  where vNE = undefined
        vH = undefined
        vT = traverse undefined undefined
```

de telle sorte que :

```
>>> validateNom "Peschanski"
Validation (Right "Peschanski")

>>> validateNom "peschanski"
Validation (Left ["Ce n'est pas une majuscule : 'p'"])

>>> validateNom "peschaNskI"
Validation (Left ["Ce n'est pas une majuscule : 'p'"
                  ,"Ce n'est pas une miniscule : 'N'"
                  ,"Ce n'est pas une miniscule : 'I'"])

>>> validateNom ""
Validation (Left ["La chaine est vide"])
```

Remarque : on utilise ici la variante de *apply*, l'opérateur `*>`, qui "oublie" le résultat calculé. On peut l'appeler *apply right* (et il existe aussi *apply left* `<*` que l'on utilise plus rarement).

On a par exemple :

```
>>> (+) <$> (Just "hello" *> Just 39) <*> (Just 3)
Just 42
```

Le résultat `Just "hello"` a été “oublié” en faveur du `Just 39`.

```
>>> (+) <$> (Nothing *> Just 39) <*> (Just 3)
Nothing
```

En revanche, si `Nothing` apparaît il est bien propagé à l'ensemble de l'expression applicative. Pour la validation, c'est pareil, seules les erreurs seront remontées, les valeurs calculées

```haskell
headDefault :: [a] -> a -> a
headDefault [] d = d
headDefault (x:_) _ = x

tailDefault :: [a] -> [a] -> [a]
tailDefault [] tl = tl
tailDefault (_:xs) _ = xs

validateNom :: String -> Valid String
validateNom str = (:) <$> (vNE *> vH) <*> vT
  where vNE = validateNonEmpty str
        vH = validateMajuscule (headDefault str 'X')
        vT = traverse validateMinuscule (tailDefault str [])
```

Les autres questions du TME sont ... dans le TME.