Examen 1ère Session

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2023

Durée : 2 heures

Documents autorisés : Documents de cours et notes personelles

$\Longrightarrow$ **La clarté et la concision des réponses seront particulièrement appréciées** $\Longleftarrow$

Exercice 1 : Mise en route

Question 1.1

Définir la fonction dont la signature est la suivante :

```
splitWith :: (a -> Bool) -> [a] -> [[a]]
```

en se basant sur les exemples suivants :

```
>>> splitWith (>3) [1, 2, 3, 4, 3, 2, 1, 4, 5, 6] :: [[Int]]
[[1,2,3],[3,2,1]]
>>> splitWith even [1, 11, 2, 4, 3, 9, 3, 2, 1, 4, 5, 7, 6] :: [[Int]]
[[1,11],[3,9,3],[1],[5,7]]
>>> splitWith odd [1, 11, 2, 4, 3, 9, 3, 2, 1, 4, 5, 7, 6] :: [[Int]]
[[2,4],[2],[4],[6]]
>>> splitWith (==True) []
[]
>>> splitWith (==True) [True, True, True]
[]
>>> splitWith (==True) [False, False, False]
[[False,False,False]]
```

En déduire la définition de la fonction `split` telle que :

```
>>> split 4 [1, 2, 3, 4, 3, 2, 1, 4, 5, 6] :: [[Int]]
[[1,2,3],[3,2,1],[5,6]]
>>> split 4 [4, 1, 2, 3, 4, 3, 2, 1, 4, 5, 4, 6, 4] :: [[Int]]
[[1,2,3],[3,2,1],[5],[6]]
>>> split 4 [4, 1, 2, 3, 4, 4, 4, 3, 2, 1, 4, 5, 4, 6, 4] :: [[Int]]
[[1,2,3],[3,2,1],[5],[6]]
>>> split ' ' "hello   cruel world"
["hello","cruel","world"]
```

Question 1.2

Soit la fonction suivante :

```
comp w = length . filter (==w) . (split ' ')
```

Quelle est la signature de `comp` ?

Expliquer succintement ce que fait cette fonction, et donner 2 exemples d'utilisation *informatifs*.

Exercice 2 : Categories et Arrows

Comme on l'a vu en cours, un objectif fondamental des langages fonctionnels consiste à mettre au premier plan la construction des programmes par composition de fonctions. En Haskell, cette idée culmine avec le style *point free* dont une illustration est la fonction `comp` de l'exercice précédent.

Dans le rapport *Programming with Arrows*[1], John Hugues, rappelle la problématique de la «composition en contexte». Si on considère par exemple le contexte des entrées/sorties, on serait tenter d'écrire des choses comme ce qui suit :

```
compIO :: String -> FilePath -> IO ()
compIO w = putStrLn . show . length . filter (==w) . split ' ' . readFile
                                                               ~~~~~~~~
error:
    • Couldn't match type: IO String
                      with: [Char]
      Expected: FilePath -> String
        Actual: FilePath -> IO String
```

On rappelle que `readFile :: FilePath -> IO String` permet de lire un fichier sous forme de chaîne de caractère.

Question 2.1

On connaît déjà une solution partielle au problème : exploiter la monade `IO`. Proposer une définition correcte de `compIO` *sans* utiliser la notation `do` et en style *point free* (dans la mesure du possible).

Dans cet exercice, nous allons étudier des alternatives à cette approche, en commençant par la *typeclass* des *catégories*, que l'on peut définir ainsi :

```
class Cat cat where
  ident :: cat a a
  compose :: cat b c -> cat a b -> cat a c
```

Question 2.2

Quel est le *kind* de `Cat` ?

Nous l'avons vu en cours, les catégories représentent les structures «qui se composent bien» avec des identités et des compositions, associées à des lois d'associativité (que l'on ne rappelle pas ici).

Les fonctions pure forment une catégorie, l'instanciation correspondante étant la suivante :

```
instance Cat (->) where
  ident = id
  compose = (.)
```

Pour la composition en catégories, on utilise souvent un opérateur «de gauche à droite» que l'on peut définir ainsi :

```
(>>>) :: Cat cat => cat a b -> cat b c -> cat a c
f >>> g = compose g f
```

Ainsi nous pouvons redéfinir `comp` de la façon suivante :

```
comp w = split ' ' >>> filter (==w) >>> length
```

Question 2.3

On introduit maintenant un type des *fonctions contextuelles* :

```
newtype ContextFun m a b = CF {runCF :: a -> m b}
```

[1] https://www.cse.chalmers.se/~rjmh/afp-arrows.pdf

On aura par exemple :

```
readFileCF :: ContextFun IO FilePath String
readFileCF = CF readFile

printCF :: Show a => ContextFun IO a ()
printCF = CF (putStrLn . show)
```

On aimerait pouvoir écrire, par exemple, la composition suivante :

```
printFileCF :: ContextFun IO FilePath ()
printFileCF = readFileCF >>> printCF
```

Pour cela, nous devons instancier la catégorie `Cat` pour les fonctions contextuelles.

Définir les fonctions `identCF` et `composeCF` pour compléter l'instanciation ci-dessous :

```
instance Monad m => Cat (ContextFun m) where
  ident = identCF
  compose = composeCF
```

Remarque : les fonctions contextuelles ne forment une catégorie que dans le cadre d'un contexte monadique.

Question 2.4

On peut maintenant utiliser la composition `>>>` pour les fonctions pures, comme dans `filter (==w) >>> length`

... où pour composer fontions contextuelles, comme dans `readFileCF >>> printCF`.

Cependant, le mélange des deux pose encore problème, on ne peut pas encore écrire, par exemple :

```
compCFWrong = readFileCF >>> split ' ' >>> filter (==w) >>> length >>> printCF
```

Il nous manque un opérateur permettant de *réifier* les fonctions pures dans le contexte, ce qui est le rôle des *arrows* dont une définition possible est la suivante :

```
class Cat x => Arr x where
    arr : (a -> b) -> x a b
```

Compléter les instances suivantes :

```
instance Arr (->) where
    arr = _A_COMPLETER_

instance Monad m => Arr (ContextFun m) where
    arr f = CF _A_COMPLETER_
```

Et on pourra finalement écrire :

```
compCFGood = readFileCF >>> arr (split ' ') >>> arr (filter (==w)) >>> arr length >>> printCF
```

Ce n'est pas parfait, mais on obtient ainsi un style *point free* assez homogène.

Remarque : en pratique, les catégories et les *arrows* sont définies dans la bibliothèque de base, sous les noms respectifs `Category` et `Arrow`. Ce style de programmation est en particulier utilisé dans le cadre de la *programmation réactive fonctionnelle* (FRP).

Exercice 3 : Les Rose Trees

Les *rose trees* sont une généralisation des arbres généraux. En Haskell une définition possible est la suivante :

```
data Rose a = Node a [Rose a]
  deriving (Show, Eq, Ord)

leaf :: a -> Rose a
leaf x = Node x []
```

On peut par exemple encoder des arbres généraux, comme le suivant :

```
rose1 :: Rose Integer
rose1 = Node 15 [leaf 3, Node 45 [leaf 27, leaf 9, leaf 81]]
```

Les *rose trees* correspondent également à une généralisation des *streams*, avec par exemple l'arbre «infini» suivant :

```
rose2 :: Rose (Integer, Integer)
rose2 = buildN 0 1
  where buildN lvl num = Node (lvl, num) (buildF (lvl+1) 1)
        buildF lvl num = buildN lvl num : buildF lvl (num+1)
```

(en pratique, on utilise ce genre de structure pour explorer des espaces d'états infinis de façon paresseuse)

Question 3.1

Définir la fonction `preorder` qui implémente le parcours préfixe d'un *rose tree*.

Par exemple :

```
>>> preorder rose1
[15,3,45,27,9,81]
>>> take 10 (preorder rose2)
[(0,1),(1,1),(2,1),(3,1),(4,1),(5,1),(6,1),(7,1),(8,1),(9,1)]
```

Question 3.2

Pour un arbre de profondeur infinie, le parcours préfixe ne donne pas beaucoup d'information. Un parcours en largeur semble plus appropriée. Pour cela nous allons définir une fonction permettant de retourner la liste des sous-arbres à une profondeur donnée, en partant de 0. La signature proposée est la suivante :

```
level :: Integer -> Rose a -> [Rose a]
```

On aura par exemple :

```
>>> level 0 rose1
[Node 15 [Node 3 [],Node 45 [Node 27 [],Node 9 [],Node 81 []]]]
>>> level 1 rose1
[Node 3 [],Node 45 [Node 27 [],Node 9 [],Node 81 []]]
>>> level 2 rose1
[Node 27 [],Node 9 [],Node 81 []]
>>> level 3 rose1
[]
```

Question 3.3

On souhaite maintenant utiliser la fonction suivante :

```
valuesAt :: Integer -> Rose a -> [a]
valuesAt lvl r = fmap (\(Node x _) -> x) (level lvl r)
```

Pour cela, il faut pouvoir utiliser `fmap`.

Proposer une instanciation de la typeclass `Functor` pour les *rose trees*.

Rappel :

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

On aura par exemple :

```
>>> fmap (*2) rose1
Node 30 [Node 6 [],Node 90 [Node 54 [],Node 18 [],Node 162 []]]
```

Que valent les expressions suivantes ?

```
>>> valuesAt 2 rose1
???
>>> take 5 (valuesAt 0 rose2)
???
>>> take 5 (valuesAt 1 rose2)
???
>>> take 5 (valuesAt 2 rose2)
???
>>> take 5 (valuesAt 3 rose2)
???
```

Question 3.4

Proposer une instance de `Foldable` pour les *rose trees*.

Rappel :

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

class Monoid m where  -- version abrégée
  mempty :: m
  (<>) :: m -> m -> m
```

Par exemple :

```
>>> foldMap Sum rose1
Sum {getSum = 180}
>>> foldMap Product rose1
Product {getProduct = 39858075}
```

Question 3.5

Définir une fonction de recherche d'une valeur dans un *rose tree* :

```
roseFind :: (a -> Bool) -> Rose a -> Maybe a
```

Par exemple :

```
>>> roseFind even rose1
Nothing
>>> roseFind odd rose1
Just 15
>>> roseFind (==3) rose1
Just 3
```

Question 3.6

On peut généraliser ce principe de recherche au délà des *rose trees*. Pour cela, on introduit le type suivant :

```
newtype First a = First { getFirst :: Maybe a }
```

Compléter les instances suivantes :

```
instance Semigroup (First a) where
    (First _?) <> b = _?
    a <> _ = _?

instance Monoid (First a) where
    mempty = _?
```

... de sorte qu'avec la définition suivante :

```
find ok = getFirst . foldMap (\x -> First (if (ok x) then Just x else Nothing))
```

on obtienne :

```
>>> find even rose1
Nothing
>>> find odd rose1
Just 15
>>> find (==3) rose1
Just 3
```

Que valent les expressions suivantes ?

```
>>> find even [1,3,5,7,9] :: Maybe Integer
???
>>> find odd [2,4,3,5,8,10] :: Maybe Integer
???
>>> find (==3) [2, 5, 3, 8, 10] :: Maybe Integer
???
```

En déduire la signature la plus générale possible pour la fonction `find`.

Question 3.7

Proposer une instance de la typeclass `traversable` pour les *rose trees*

Rappel :

```
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)
    -- etc ...
```

Remarque : on pourra définir `traverse` ou `sequenceA` au choix.

Donner l'affichage produit par l'expression suivante (qui est de type `IO ()`) :

```
>>> printAll 5 rose1
```

avec :

```
printAll y tr = do traverse (printAux y) tr
                   putStrLn ""

printAux y x = do { aux ; putStr " " ; return () }
  where aux = putStr $ if x `mod` y == 0 then show x else "-"
```