Examen 1ère Session

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2021

Durée : 2 heures

Documents autorisés : tous documents papier (notes de cours, corrections de TD, etc.)

Le sujet est composé de deux exercices indépendants : un exercice de questions de cours et un exercice de modélisation. Ilst *vivement* conseillé de consulter rapidement l'ensemble des questions de chaque exercice avant de commencer à répondre.

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

Exercice 1 : Les triples monadiques

Soit le type suivant :

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

Avec par exemple :

```
>>> :t Triple 42 True "hello"
Triple 42 True "hello" :: Num a => Triple a Bool [Char]
```

Question 1.1

On rappelle quelques informations sur la *typeclass* `Functor` :

```
>>> :info Functor
type Functor :: (* -> *) -> Constraint
class Functor f where
   fmap :: (a -> b) -> f a -> f b
-- etc ...
```

Proposer une instance pour les `Triples`, en vous inspirant de l'exemple suivant :

```
>>> fmap (+1) (Triple 1 2 3)
Triple 1 2 4
```

Question 1.2

On rappelle quelques informations sur la *typeclass* `Applicative` :

```
>>> :info Applicative
type Applicative :: (* -> *) -> Constraint
class Functor f => Applicative f where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
-- etc ...
```

Proposer une instance pour les `Triples`, en vous inspirant des exemples suivants :

```
>>> pure [1, 2, 3] :: Triple [Int] (Max Int) [Int]
Triple [] (Max {getMax = -9223372036854775808}) [1,2,3]
```

```
>>> Triple [1,2] (Max (12 :: Int)) (+1) <*> Triple [3,4] (Max 15) 41
Triple [1,2,3,4] (Max {getMax = 15}) 42
```

Remarque : `Max` est un constructeur de monoïdes pour l'opération de *maximum*.

Par exemple :

```
>>> Max (12 :: Int) <> Max 15
Max {getMax = 15}
```

```
>>> mempty :: Max Int
-- Max {getMax = -9223372036854775808}
```

On rappelle, au cas où, quelques informations sur les *typeclasses* `Semigroup` et `Monoid` :

```
>>> :info Semigroup
-- type Semigroup :: * -> Constraint
-- class Semigroup a where
--   (<>) :: a -> a -> a
-- etc ...

>>> :info Monoid
-- type Monoid :: * -> Constraint
-- class Semigroup a => Monoid a where
--   mempty :: a
-- etc ...
```

Question 1.3

Voici une instance proposée de la *typeclass* `Monad` pour les `Triple`s :

```
instance (Monoid a, Monoid b) => Monad (Triple a b) where
  (Triple x y z) >>= g = let (Triple x' y' z') = g z
                         in Triple (x <> x') (y <> y') z'
```

Soit la définition suivante :

```
exTripleM :: Triple [Int] (Max Int) Int
exTripleM = do
  x <- Triple [1,2] (Max 12) 19
  y <- pure 13
  z <- Triple [3,4] (Max 15) 9
  return $ x + y + z + 1
```

Quelle est la valeur de l'expression suivante :

```
>>> exTripleM
?????
```

Question 1.4

En posant `m` = `Triple x y z` pour des `x`, `y`, `z` «arbitraires» mais bien typés, montrer par raisonnement équationnel (en justifiant chaque étape), que la loi monadique suivante est bien respectée par l'instance de `Monad` pour les `Triple`s.

`m >>= pure` = `m`

(le symbole = signifiant que les deux programmes sont équivalents, i.e. possèdent le même comportement)

Exercice 2 : Spécification des Arbres Rouge-Noirs

Dans cet exercice, l'objectif est de programmer en Haskell une *spécification* de la structure de données des arbres rouge noir (*red black (RB) trees*).

Important : il ne s'agit pas de coder l'algorithmique (relativement complexe) de cette structure, mais d'en spécifier les propriétés les plus importantes. Il n'est bien sûr pas non plus nécessaire de déjà connaître cette structure pour répondre aux questions.

Nous utiliserons les définitions suivantes :

```haskell
data RBColor = RED | BLACK
  deriving (Show, Eq, Ord)

data RBTree a =
  Node { color :: RBColor, val :: a, left :: RBTree a, right :: RBTree a}
  | Tip
  deriving (Show, Eq, Ord)

leaf :: RBColor -> a -> RBTree a
leaf col v = Node col v Tip Tip

exTree :: RBTree Int
exTree =
 Node BLACK 13
   (Node RED 8 (Node BLACK 1 Tip (leaf RED 6))
               (leaf BLACK 11))
   (Node RED 17 (leaf BLACK 15)
                (Node BLACK 25 (leaf RED 22)
                               (leaf RED 27)))
```

La représentation choisie sépare les nœuds terminaux (constructeur `Tip`) vides, et les nœuds internes (constructeur `Node`) avec une couleur (rouge ou noire), une valeur portée (paramètre de type `a`), un sous-arbre gauche (`left`) et un sous-arbre droit (`right`).

Question 2.1

Proposer une implémentation récursive de la fonction dont la signature est la suivante :

```haskell
search :: Ord a => a -> RBTree a -> Maybe (RBTree a)
```

Avec par exemple :

```
>>> search 22 exTree
Just (Node {color = RED, val = 22, left = Tip, right = Tip})

>>> search 33 exTree
Nothing
```

Question 2.2

On souhaite définir un *fold* adapté aux `RBTree`s, avec la signature suivante :

```haskell
foldRB :: (RBColor -> a -> b -> b) -> b -> RBTree a -> b
```

Proposer une définition récursive pour cette fonction, en vous inspirant des exemples suivants :

```
-- Nombre de nœuds
>>> foldRB (\_ _ acc -> acc + 1) 0 exTree
10
```

```
-- Somme des valeurs
>>> foldRB (\_ v acc -> acc + v) 0 exTree
145

-- Liste des contenus
>>> foldRB (\c v acc -> (c,v):acc) [] exTree
[(BLACK,1),(RED,6),(RED,8),(BLACK,11),(BLACK,13),(BLACK,15),(RED,17),(RED,22),(BLACK,25),(RED,27)]
```

On remarque que la liste générée ci-dessus est ordonnée sur les valeurs des nœuds, on a donc effectué un parcours *infixe*, en prenant pour chaque nœud interne: (1) le sous-arbre gauche, puis (2) le nœud lui-même, et enfin (3) le sous-arbre droit.

Question 2.3

On rappelle quelques informations sur la *typeclass* `Foldable` :

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

(cf. Exercice 1 / Question 1.2 pour la *typeclass* `Monoid`)

Proposer une instance de `Foldable` pour `RBTree` en vous inspirant des exemples suivants :

```
>>> foldMap (\v -> [v]) exTree
[1,6,8,11,13,15,17,22,25,27]

>>> foldMap Max exTree
Max {getMax = 27}

>>> foldMap Sum exTree
Sum {getSum = 145}

>>> foldMap Product exTree
Product {getProduct = 25992252000}
```

Question 2.4

On s'intéresse maintenant à définir les propriétés fondamentales des `RBTree`s.

Pour cette question, l'objectif est d'implémenter la propriété suivante :

```
prop_bst :: Ord a => RBTree a -> Bool
```

> Tout nœud interne d'un arbre rouge-noir possède la propriétés des arbres binaires de recherche (*binary search trees*, bst), telle que :
>
> - le sous-arbre gauche et le sous-arbre droit possèdent la même propriété, et :
> - la valeur portée par le nœud est strictement supérieure aux valeurs du sous-arbre gauche, et strictement inférieure à celles du sous-arbre droit (on suppose que toutes les valeurs sont distinctes)

(on supposera la propriété vraie pour les terminaux)

On a ainsi :

```
>>> prop_bst exTree
True
```

Remarque: pour cette question et les suivantes, il ne faut pas hésiter à définir des fonctions auxiliaires - avec leur signature - pour faciliter l'expression des propriétés.

Question 2.5

Proposer une implémentation de la propriété suivante :

```
prop_red :: RBTree a -> Bool
```

> Les racines des sous-arbres gauche et droit d'un nœud interne **rouge** sont de couleur noire. Et les deux sous-arbres eux-mêmes respectent la propriété.

Remarque : les terminaux sont considérés de couleur noire.

On a donc :

```
>>> prop_red exTree
True
```

Question 2.6

Un chemin d'un `RBTree a` est une suite de nœuds (on prendra une liste de type `[(RBColor,a)]`) depuis la racine jusqu'à un terminal.

Définir la fonction de signature suivante :

```
allPaths :: RBTree a -> [[(RBColor, a)]]
```

qui retourne la liste des chemins non-redondants d'un arbre rouge-noir.

Par exemple :

```
>>> allPaths Tip
[]

>>> allPaths (leaf BLACK 42)
[[(BLACK,42)]]

>>> allPaths (left (left exTree))
[[(BLACK,1)],[(BLACK,1),(RED,6)]]

>>> allPaths (right (left exTree))
[[(BLACK,11)]]

>>> allPaths (left exTree)
[[(RED,8),(BLACK,1)],[(RED,8),(BLACK,1),(RED,6)],[(RED,8),(BLACK,11)]]
```

Question 2.7

La fonction `allPaths` nous permet de programmer la propriété des chemins d'arbres rouge-noirs:

```
prop_pathRB :: RBTree a -> Bool
```

> Tous les chemins d'un arbre rouge-noir possèdent le même nombre de nœuds de couleur noire.

Proposer une définition de cette propriété.

Par exemple :

```
>>> prop_pathRB exTree
True
```