Thème 1 - TD2 - Modélisation avec les Types (2/4)

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2021

Exercice 1 : Prélude aux types paramétrés

On manipule dans les questions suivantes (et indépendantes) des types définis dans la bibliothèque standard de Haskell : le **prélude**.

Question 1 : fonction partielle vs. fonction totale

La fonction `head` du prélude est une fonction partielle :

```
>>> head [1, 2, 3]
1
```

```
>>> head []
*** Exception: Prelude.head: empty list
```

L'inconvénient des fonctions partielles qui génèrent une exception (ou bouclent indéfiniment) est qu'elles contredisent une propriété fondamentale de la programmation fonctionnelle **pure** (et **totale**) : la *transparence référentielle*. Cette propriété explique que l'on peut toujours remplacer un appel de fonction par le résultat du calcul sans changer la signification du programme. Mais une fonction partielle justement n'a pas toujours de résultat en fonction de ses entrées.

Considérons par exemple la fonction suivante :

```haskell
headPlus :: [Integer] -> Integer -> Integer
headPlus xs n = (head xs) + n
```

Cette fonction est bien typée et semble (donc) correcte :

```
>>> headPlus [1, 2, 3] 42
43
```

```
>>> headPlus [] 42
*** Exception: Prelude.head: empty list
```

Le problème vient encore du caractère partiel de la fonction `head`, mais ceci se propage maintenant à toute fonction qui l'utilise, ce n'est pas un phénomène local. Il n'est pas toujours possible d'implémenter de façon pure (sans effets de bord) et totale (sans rejeter des entrées bien typées) une fonction partielle, mais dans de nombreuses situations cela est possible grâce aux types `Maybe` et `Either`, ou d'autres types similaires.

Proposer une version totale, appelée `safeHead` telle que par exemple :

```
>>> safeHead [1, 2, 3]
Just 1
```

```
>>> safeHead []
Nothing
```

Compléter ensuite la variante `safeHeadPlus` de la fonction `headPlus` ci-dessous :

```haskell
safeHeadPlus :: [Integer] -> Integer -> Maybe Integer
safeHeadPlus xs n = case safeHead xs of
                      -- <à compléter>
```

Question 2 : variante avec Either

Une différence importante entre la version partielle et la version avec `Maybe` est que dans le premier cas, en cas d'erreur, on a une information associée : `empty list`. Dans le deuxième cas, il faut comprendre que `Nothing` signifie justement ce cas d'erreur.

On peut reproduire une information similaire tout en restant dans le monde des fonctions totales en utilisant le type `Either a b`.

On rappelle la définition de ce type :

```haskell
data Either a b = Left a | Right b
```

Le cas d'erreur est `Left` (mis de côté) et on utilise `Right` quand tout est ok !

On pose le type unitaire suivant :

```haskell
data EmptyList =
     EmptyList
     deriving Show
```

Définir la fonction suivante :

```haskell
eitherHead :: [a] -> Either EmptyList a
```

telle que :

```
>>> eitherHead [1, 2, 3]
Right 1

>>> eitherHead []
Left EmptyList
```

Définir la fonction `eitherHeadPlus`, variante de `headPlus` basée sur la fonction précédente.

Question 3 : variante avec défaut

Dans les questions précédentes, en passant à une version *"safe"*, on a vu que les appelants (comme `safeHeadPlus` ou `eitherHeadPlus`) devaient être modifiés pour prendre en compte l'aspect partiel, maintenant explicite, dans le type de la fonction. Cette propagation du `Maybe` ou `Either` aux appelants peut parfois rendre les choses un peu fastidieuse. Le *style monadique* (et la notation `do`) offre une solution générale et élégante au problème, mais nous ne l'aborderons que dans la deuxième partie du cours. Une autre possibilité qui peut s'appliquer dans certaines situations est de «totaliser» une fonction partielle par la la fourniture de valeurs par défaut.

Proposer une définition de la fonction `defaultHead` qui prend en argument une liste et une valeur par défaut, et retourne la tête de liste si celle-ci existe, ou la valeur par défaut sinon.

Par exemple :

```haskell
>>> defaultHead [1, 2, 3] 4
1

>>> defaultHead [] 4
4
```

En déduire une définition de `defaultHeadPlus` qui elle considérera que la tête d'une liste vide vaut 0 (l'élément neutre pour l'addition).

Exercice 2 : Fold pour les listes

Nous revisitons dans cet exercice un grand classique des combinateurs de listes : `fold`. Nous allons travailler sur le type liste défini en cours, et répété ci-dessous :

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

On rappelle également le foncteur *map* pour ces listes (renommé en `listMap` ici) :

```
listMap :: (a -> b) -> List a -> List b
listMap _ Nil = Nil
listMap f (Cons e l) = Cons (f e) (listMap f l)
```

Question 1

Voici le type de la fonction `foldl` de Haskell

```
>>> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

C'est une fonction polymorphe disponible pour toute structure de donnée *pliable* (*foldable*). Pour le cas particulier des listes, on peut spécialiser ce type de la façon suivante :

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

Par exemple on peut calculer la longueur d'une liste :

```
len :: [a] -> Int
len = foldl step 0
  where step :: Int -> a -> Int
        step k _ = k + 1
```

```
>>> len [1, 2, 3, 4]
4
```

Ou plus simplement :

```
>>> foldl (\x _ -> x + 1) 0 [1,2,3,4]
4
```

Ce qu'en Haskell courant on écrirait finalement :

```
>>> foldl (+1) 0 [1,2,3,4]
4
```

Définir la fonction `listFoldl` qui effectue le même traitement sur les listes *"maison"* (sans utiliser `foldl` bien sûr !)

Par exemple :

```
>>> listFoldl (\x _ -> x + 1) 0 $ Cons 1 $ Cons 2 $ Cons 3 $ Cons 4 Nil
4
```

Question 2

Peut-on définir une variante `listMap2` de `listMap` sans récursion explicite mais en utilisant `listFoldl` ?

(si oui : le faire, si non : expliquer)

Question 3

En utilisant le principe d'induction sur les listes (déduit de la définition du type, comme expliqué en cours), montrer la propriété suivante :

```
prop_lmap :: f -> List a -> Bool
prop_lmap f xs
= listMap f xs == listMap2 f xs
= listMap f xs == listFoldl aux Nil xs
```

Exercice 3 : Arbres binaires/ternaires

Question 1 :

Donner une définition de type `BTTree` des arbres binaires/ternaires avec des constructeurs `Leaf`, `Two` et `Three`. Seuls les noeuds internes portent des étiquettes d'un type arbitraire (on n'utilisera pas les *records* dans cet exercice).

Question 2 :

Définir des fonctions: `nbTwo` pour calculer le nombres de noeuds binaires dans un arbre, `depth` pour la profondeur d'un arbre, et `prefix` pour la liste des étiquettes en parcours préfixe (avec les listes du *prélude* de Haskell).

Question 3 :

Donner le principe d'induction sur les arbres binaires/ternaires, en essayant de le déduire de la définition du type.

Question 4 :

Définir une fonction `treeMap` pour les arbres binaires/ternaires en vous inspirant de `map` pour les listes. Définir de façon similaire une fonction `treeFold` inspirée de `foldl`.

Redéfinir les fonction de la question précédente avec cette dernière fonction.

Question 5

Montrer la propriété suivante :

```
prop_depth:: BTTree a -> Bool
prop_depth t = depth t == depth2 t
```