Thème 2 - TD4 - Évaluation non-stricte et structures infinies

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2021

Exercice 1 : Listes infinies et corécursion

Les générateurs paresseux apparaissent depuis peu dans les langages de programmation *«mainstream»* (par exemple en Python, Ruby ou même Java et javascript). Comme la plupart de ces langages sont stricts, il s'agit d'introduire des concepts supplémentaires : `yield` en Python, Ruby ou Javascript, `delay`/`force` du module *Lazy* en Ocaml, etc. En Haskell, la stratégie non-stricte permet de coder des générateurs paresseux sans utiliser de constructions dédiées.

Par exemple, en cours on a vu comment construire la liste infinie de 1's

```
ones :: [Integer]
ones = 1 : ones
```

Cette définition se traduit littéralement :

La liste infinie des 1's est construite avec comme premier élément 1 et comme reste la suite infinie des 1's.

Question 1.1

Définir une fonction `genInteger` telle que `genInteger n` génère la liste infinie `[n, n , n, ...]`

Remarque : dans le module `Data.List`, une généralisation (pour tout type `a`) de cette fonction existe et se nomme `repeat`.

Par exemple :

```
>>> take 10 $ genInteger 42
[42,42,42,42,42,42,42,42,42,42]

>>> take 10 $ repeat 42
[42,42,42,42,42,42,42,42,42,42]
```

Contrairement à ce que l'on pourrait penser, ces définitions ne sont *pas* récursives car les objets construits ne sont pas finis. En particulier ces fonctions n'ont pas forcément de cas de base. Il s'agit de définitions dites *corécursives*. La théorie sous-jacente est très intéressante mais on restera à un niveau de compréhension informel du concept (d'ailleurs connaissez vos la théorie de la récursion ? Connaissez-vous les théorèmes de point fixe de Knaster-Tarski ? Les algèbres et coalgèbres ?).

Puisque l'objet `ones` est infini (de même que tout object `repeat v` pour tout `v`), il ne faut jamais se trouver dans une situation où son évaluation complète est nécessaire. Par exemple, l'expression suivante boucle dans ghci :

```
>>> ones
... boucle infinie ...
```

Il est cependant possible d'*observer* un *préfixe fini* de la structure. Une fonction très utile pour cela est la fonction `take` du prélude.

```
>>> take 10 ones
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
>>> take 10 (repeat 'z')
"zzzzzzzzzz"
```

Question 1.2.

Donner une définition de la fonction `myTake` telle que `myTake n xs` (avec `n` de type `Int`) retourne la même valeur que `take n xs` (sans utiliser cette dernière bien sûr).

(Remarque : cette fonction a été vue en cours mais il est utile de la rappeler)

Cette fonction est-elle :

- récursive ou corécursive ?
- stricte ou non-stricte ?
- totale ou partielle ?

Question 1.3.

Définir corécursivement la liste ordonnée `nats` des entiers naturels (de type `Integer`) , donc *sans* utiliser la notation `[1 ..]` (qui fait la même chose).

Par exemple :

```
>>> take 10 nats
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

(Remarque : cette fonction aussi a été définie plusieurs fois en cours. . .)

Question 1.4.

La fonction `map` sur les listes est telle que `map f [`e_1 , e_2, . . . , e_i, . . .`] = [f` e_1, `f` e_2, . . . , `f` e_i, . . .`]`.

En déduire une nouvelle définition pour `nats` en utilisant `map`.

Question 1.5.

Définir corécursivement la liste infinie `fibo` des nombres dits de *Fibonacci* :

```
[1, 1, 2, 3, 5, 8, 13, ...]
```

Question 1.6.

La fonction zipWith du prélude possède la signature suivante :

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Par exemple :

```
>>> take 10 $ zipWith (*) nats (tail nats)
[2,6,12,20,30,42,56,72,90,110]
```

En déduire une (potentiellement) nouvelle définition de `fibo` obtenue en remplaçant les `???` ci-dessous :

```
fibo :: [Integer]
fibo = 1 : 1 : ??? ??? ??? (tail ???)
```

Exercice 2 : Compréhensions de listes (infinies ou non)

Dans cet exercice, nous illustrons le concept de *compréhension* de liste, concept que l'on peut trouver par exemple en Python mais aussi en Haskell.

Voici la syntaxe générale des compréhensions de listes en Haskell :

```
[ <yield_expr> | <var1> <- <list1>, <var2> <- <list2>, ..., <condition>, ... ]
```

La sémantique informelle d'une telle expression est la construction d'une liste dont chaque élément est une évaluation de `<yield_expr>` dans laquelle des variables `<var1>`, `<var2>`, ... apparaissent. Chaque variable est associée à une liste `<list1>`, `<list2>` ... qui seront parcourues dans l'ordre séquentiel, selon un produit cartésien. Il est de plus possible d'indiquer des conditions permettant de filtrer la liste résultat.

Par exemple, on peut construire la liste des carrés d'une liste d'entiers :

```
>>> [x * x | x <- [1..5]]
[1,4,9,16,25]
```

Avec deux variables, on obtient un produit cartésien :

```
>>> [(x, y) | x <- [1..3], y <- [1..3]]
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
```

On peut de plus ajouter des conditions :

```
>>> [x * x | x <- [1..10], even x]
[4,16,36,64,100]
```

```
>>> [(x, y) | x <- [1..4], y <- [1..4], x >= y]
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3),(4,1),(4,2),(4,3),(4,4)]
```

Les compréhensions de listes sont bien paresseuses comme le montre les exemples suivants :

```
>>> take 10 [x | x <- [1..], even x]
[2,4,6,8,10,12,14,16,18,20]
```

```
>>> take 10 [(x, y) | x <- [1..], y <- [1..4], x >= y]
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3),(4,1),(4,2),(4,3),(4,4)]
```

Nous allons voir, dans cet exercice, les fondements *fonctionnels* de ces compréhensions.

Question 2.1. : Tout commence avec flatMap

Définir une fonction `flatMap` (parfois également appelée `concatMap` ou `mappend`) avec le type suivant :

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

telle que par exemple :

```
flatMap (\x -> [x, 10 * x, 100 * x]) [1, 2, 3, 4, 5]
-- => [1,10,100,2,20,200,3,30,300,4,40,400,5,50,500]
```

Question 2.2. : compréhensions simples avec flatMap

Par soucis de lisibilité, on modifie légèrement la fonction `flatMap` en utilisant la fonction suivante :

```
compr = flip flatMap
-- avec :
flip :: (a -> b -> c) -> b -> a -> c
flip f y x = f x y
```

1) Quel est le type de `compr` ?

Avec cette fonction, on peut exprimer des compréhensions simples (non-conditionnelles) «simplement».

On peut par exemple réécrire `[x * x | x <- [1..5]]` de la façon suivante :

```
>>> [1..5] `compr` (\x -> [x * x])
[1,4,9,16,25]
```

C'est moins lisible, mais nous sommes désormais en mode «purement fonctionnel», sans sucre syntaxique.

2) Donner les valeurs des expressions suivantes :

```
a> take 10 ([1..] `compr` (\x -> [x *x]))
b> [1..3] compr (\x -> [x, 10 * x, 100 * x])
c> take 10 ([1..] `compr` (\x -> [1..4] `compr` (\y -> [(x, y)])))
```

Question 2.3 : Compréhensions conditionnelles

Pour pouvoir prendre en compte des conditions dans les compréhensions «purement fonctionnelles», on introduit les définitions suivantes :

```
success :: a -> [a]
success x = [x]

failure :: [a]
failure = []
```

1) Donner les résultats des évaluations suivantes :

```
a> [1..8] `compr` (\x -> if even x then (success x) else failure)
b> [1..8] `compr` (\x -> if odd x then (success x) else failure)
c> take 10 ([1..] `compr` (\x -> if even x then (success x) else failure))
d> [1..8] `compr` (\x -> if even x then success (x, x*x) else failure)
```

2) Définir la fonction `select` telle que les expressions ci-dessus puissent être réécrites de la façon suivante :

```
a> [1..8] `compr` (select even id)  -- avec id :: a -> a  la fonction identité
b> [1..8] `compr` (select odd id)
c> take 10 ([1..] `compr` (select even id))
d> [1..8] `compr` (\x -> (select even (\x -> (x, x*x))))
```

3) Traduire la compréhension suivante en mode «purement fonctionnel»

```
take 10 [(x, y) | x <- [1..], y <- [1..4], x >= y]
```

Exercice 3 : Manipulations paresseuses d'arbres

Dans cet exercice, nous travaillons sur la structure d'arbres binaires suivante :

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

leaf :: a -> Tree a
leaf v = Node v Tip Tip

exTree :: Tree Int
exTree = Node 5 (Node 7 (leaf 6) (leaf 4))
                (Node 2 (leaf 1) (leaf 3))
```

Question 3.1.

Définir la fonction `maxTree` qui retourne la valeur maximale d'un `Tree a` avec la constrainte `Ord a`.

Par exemple :

```
>>> maxTree exTree
7
```

Question 3.2.

Définir la fonction `addTree` telle que `addTree n t` ajoute `n` à toutes les feuilles de l'arbre `t` de type `Tree a` avec la contrainte `Num a`.

Par exemple :

```
>>> addTree 42 exTree
Node (Node (Leaf 47)
           (Node (Leaf 49) (Leaf 45)))
     (Node (Leaf 43) (Leaf 48))
```

Question 3.3. (assez difficile)

Considérons l'expression suivante :

```
>>> addTree (maxTree exTree) exTree
Node 12 (Node 14 (Node 13 Tip Tip) (Node 11 Tip Tip))
         (Node 9 (Node 8 Tip Tip) (Node 10 Tip Tip))
```

Il s'agit d'ajouter le maximum des éléments trouvé dans l'arbre de départ ... à toutes les feuilles de l'arbre. Ce calcul (strict) se produit en deux passes : une passe pour récupérer le maximum, et une passe pour effectuer les additions.

En exploitant la stratégie non-stricte, définir une fonction `addMaxTree` qui réalise le même calcul en une seule passe dans la structure d'arbre.

Remarque : on pourra penser à une fonction auxiliaire qui retourne un couple `(t', mx)` avec `mx` le maximum et `t'` l'arbre résultat.

Question 3.4. (assez difficile)

Définir la liste `allTrees` énumérant tous les arbres binaires de type `Tree ()`.

Remarque : on pourra penser à décomposer le problème en générant tous les arbres de taille `n` (en nombre de noeuds internes) à partir des énumération des arbres de tailles `1`, `2`, ,..., `n-1`