Série d'exercices #6

IFT-2035

October 7, 2024

6.1 Structure de données infinie

Les listes infinies sont souvent appelées *streams*. En Haskell, l'ordre d'évaluation utilisé permet d'utiliser n'importe quelle structure de donnée infinie sans effort particulier. Prenons par exemple les définitions ci-dessous:

```
zeros = 0 : zeros
uns = 1 : uns
numbers = 0 : zipWith (+) uns numbers
f = 0 : 1 : zipWith (+) f (tail f)
```

De plus, Haskell prédéfini les opérations suivantes:

```
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n - 1)

take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n - 1) xs
```

Avec les définitions ci-dessus, nous avons:

$$\begin{array}{lll} \textit{take}\ 5\ \textit{uns} & \rightsquigarrow^* & [1,1,1,1,1] \\ \textit{take}\ 5\ \textit{numbers} & \rightsquigarrow^* & [0,1,2,3,4] \\ \textit{numbers}\ !!\ 3 & \rightsquigarrow^* & 3 \end{array}$$

1. Montrer les étapes de l'évaluation de:

   ```
   f !! 3
   ```

   Expliquer en détail la notation que vous avez choisi d'utiliser. Attention à l'utiliser de manière cohérente et rigoureuse.

2. Soit la liste infinie de nombres suivante:

   ```
   (1 1/2 1/6 1/24 1/120 ... 1/n! ...
   ```

   Définir cette liste récursivement en utilisant *zipWith*, la liste infinie des entiers *numbers* et la liste circulaire `uns`.

6.2 Une classe de conteneurs

Soit la classe de types suivante:

```
class Conteneur a where
  size    :: a → Int
  subset  :: a → a → Bool
  reverse :: a → a
```

Où `reverse` renvoie un conteneur dont les éléments sont dans l'ordre inverse.

1. Donner une définition pour l'instance `Integer` (on n'utilisera pas les nombres négatifs), où chaque bit représente la présence du nombre correspondant, donc $11 = 2^0 + 2^1 + 2^3$ représente l'ensemble $\{0, 1, 3\}$.
2. Donner une définition pour l'instance `[Int]`[1]. Pour vous aider vous pouvez utiliser la fonction prédéfinie `elem :: Eq a ⇒ [a] → [a] → Bool`.
3. Donner une définition pour l'instance `Maybe a`, qui ne peut contenir qu'un élément au maximum.
   Rappel: `data Maybe a = Nothing | Just a`
4. Donner le type des fonctions suivantes:

```
sizes xs = map size xs
size2 x y z = if x == z
              then let s = size x in (s, s)
              else (size x, size y)
minsize x y = if subset x y then size x else size y
```

6.3 Récursion par les types

Dans le λ-calcul non-typé, le *combinateur* Y peut être utilisé pour définir des fonctions récursives:

$$Y = \lambda f \to (\lambda x \to f\ (x\ x))\ (\lambda x \to f\ (x\ x))$$

propriété: $\quad Y\ f \quad \rightsquigarrow^* \quad f\ (Y\ f)$

On peut alors définir

$$\mathit{fact'}\ \mathit{fact}\ x = \mathsf{if}\ x < 2\ \mathsf{then}\ 1\ \mathsf{else}\ x * (\mathit{fact}\ (x-1))$$
$$\mathit{fact} = Y\ \mathit{fact'}$$

Cependant, il est impossible de définir Y dans le λ-calcul simplement typé, et de même Haskell rejette la définition de Y.

1. Montrer quel problème se pose pour donner un type à Y.
2. Contourner le problème de manière à définir en Haskell une fonction *fact* qui calcule la factorielle d'un nombre, mais sans utiliser de fonction récursive.

[1] Les classes de type de Haskell n'acceptent pas ce genre d'instances à moins d'activer l'option "FlexibleInstances", par exemple en ajoutant {-# LANGUAGE FlexibleInstances #-}