Série d'exercices #7

IFT-2035

November 6, 2023

7.1 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
```

[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 #-}

7.2 Types et classes de types

Donner le types des expressions ci-dessous.

1. $(+)$
2. $(+)\ (3 :: Int)$
3. $\lambda x \rightarrow fst\ x + snd\ x$
4. $\lambda x\ y \rightarrow \mathsf{if}\ fst\ x = snd\ x\ \mathsf{then}\ y\ \mathsf{else}\ y + 1$
5. $map\ (\lambda x \rightarrow \mathsf{if}\ fst\ x < 0\ \mathsf{then}\ snd\ x\ \mathsf{else}\ snd\ x + 1)$

7.3 Ceci, puis cela

Soit le code ci-dessous qui définit une opération *cpcMaybe*. Cette opération fait une sorte de composition de fonction similaire à une sorte de séquencement.

$$
\begin{array}{l}
\mathsf{data}\ Maybe\ \alpha\ =\ Nothing \mid Just\ \alpha \\
baseMaybe\ ::\ \alpha \rightarrow Maybe\ \alpha \\
baseMaybe\ x\ =\ Just\ x \\
cpcMaybe\ ::\ Maybe\ \alpha \rightarrow (\alpha \rightarrow Maybe\ \beta) \rightarrow Maybe\ \beta \\
cpcMaybe\ x\ f\ =\ \mathsf{case}\ x\ \mathsf{of} \\
\quad Nothing\ \rightarrow\ Nothing \\
\quad Just\ x' \rightarrow\ f\ x'
\end{array}
$$

Cette opération peut être utilisée par exemple si on veut composer non pas deux fonctions $f\ ::\ \alpha \rightarrow \beta$ et $g\ ::\ \beta \rightarrow \gamma$ mais deux fonctions $f\ ::\ \alpha \rightarrow Maybe\ \beta$ et $g\ ::\ \beta \rightarrow Maybe\ \gamma$.

Définir des fonctions similaires pour les types suivants:

```
-- Renvoie une valeur ou une erreur.
```
$$
\begin{array}{l}
\mathsf{data}\ Err\ \alpha\ =\ Err\ String \mid Suc\ \alpha \\
baseErr\ ::\ \alpha \rightarrow Err\ \alpha \\
cpcErr\ ::\ Err\ \alpha \rightarrow (\alpha \rightarrow Err\ \beta) \rightarrow Err\ \beta
\end{array}
$$
```
-- Une valeur avec un compteur du coût pour la calculer.
```
$$
\begin{array}{l}
\mathsf{data}\ countSteps\ a\ =\ CountSteps\ Int\ \alpha \\
baseCS\ ::\ \alpha \rightarrow CountSteps\ \alpha \\
cpcCS\ ::\ CountSteps\ a \rightarrow (\alpha \rightarrow CountSteps\ \beta) \rightarrow CountSteps\ \beta
\end{array}
$$

Finalement généraliser ce "design pattern" en définissant une *classe de type* (ou plus précisément une classe de *constructeurs de types*) *CPC f* avec des instances *CPC Maybe*, *CPC Err*, et *CPC CountSteps*.