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

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2021

Exercice 1 : Bool'n co

Petit échauffement avec les booléens et les entiers.

Question 1.1

On rappelle la définition de l'implication logique en Haskell :

```haskell
(==>) :: Bool -> Bool -> Bool
(==>) True False = False
(==>) _ _ = True
infixr 1 ==>
```

L'implication logique est transitive, on a par exemple :

```haskell
True ==> False
```

```haskell
False ==> True
```

```haskell
True ==> True
```

```haskell
True ==> False ==> True
```

Décrire en Haskell la propriété de transitivité de l'implication.

Question 1.2

Donner, en écriture mathématiques usuelles, les principes d'induction pour les entiers naturels et pour les entiers relatifs.

Question 1.3

Donner le code Haskell d'une fonction `sum` qui, à partir d'un entier naturel `n`, calcule la somme des entiers de 1 à `n`, sans utiliser de formule directe.

```haskell
-- Hypothèse : n est un entier naturel
sum :: Integer -> Integer
sum 0 = 0
sum n = n + sum (n-1)
```

Question 1.4

Soit la propriété suivante pour tout n entier naturel :

$P(n) \stackrel{def}{=} \texttt{sum n} = \frac{n(n+1)}{2}$

Décrire la propriété en Haskell, proposer ensuite une démonstration.

Exercice 2 : Pipemania

Dans cet exercice, nous modélisons directement en Haskell un petit système de *"Cuves et tuyaux"* inspirés par le célèbre jeu Pipemania mais aussi l'industrie du Pétrole et la vigne.

Question 2.1 : Modélisation des cuves

Une cuve est un conteneur pouvant être rempli d'une certaine *quantité* de liquide, dans la limite d'une *capacité* fixée une fois pour toute. Une cuve peut se trouver dans 3 états possibles :

— la cuve est vide, elle ne contient pas de liquide
— la cuve est pleine, la quantité de liquide contenu est égal à la capacité de la cuve
— la cuve n'est ni vide, ni pleine

Décrire en Haskell un type `Tank` (cuve en anglais) permettant de représenter au mieux, par un type "somme de produit(s)" la description ci-dessus. On ajoutera sous forme de propriété `prop_tankInvariant` un *invariant* fondamental de la cuve:

La capacité est strictement positive et la quantité est inférieure ou égale à la capacité. De plus, une cuve non-vide et non-pleine possède une quantité non-nulle de liquide.

Finalement, écrire en Haskell une fonction `initTank` qui prend en paramètre une capacité et qui construit une cuve de cette capacité et ne contenant pas de liquide. On fera attention à ne pas construire la cuve si cela risque de contredire `prop_tankInvariant`. Faire une démonstration de la propriété que si la cuve est construite alors l'invariant est respecté.

Finalement, définir des observateurs `quantity` et `capacity` pour les cuves.

Question 2.2. : remplissage et pompage

Décrire, sous la forme de deux fonctions Haskell les deux opérations possible sur les cuves :

— le remplissage d'une cuve avec `fillTank :: Tank -> Integer -> Maybe Tank`
— le pompage d'une cuve avec `pumpTank :: Tank -> Integer -> Maybe Tank`

Le type `Maybe Tank` permet de reflêter le fait que les opérations de remplissage ou de pompage peuvent échouer, en fonction de la quantité de liquide considérée.

On pourra introduire une fonction auxiliaire (par ex. : `changeTank`) permettant de changer l'état d'une cuve sans contrôle supplémentaire.

Quelles propriétés faut il garantir pour justifier la correction de ces deux opérations.

Bonus : démontrer ces propriétés.

Question 2.3. : Modélisation des Tuyaux

Un tuyau (`Pipe` en anglais) relie une cuve entrante à une cuve sortante. Le tuyau lui même contient une certaine quantité de liquide. Nous proposons de modéliser les tuyaux par le type suivant :

```
data Pipe = Pipe {
  ptankIn :: Tank
  , ptankOut :: Tank
  , pdoorIn :: Bool
  , pdoorOut :: Bool
```

```
  , pcontent :: Tank
  } deriving (Show)
```

Les invariants du type Pipe sont les suivants :

- Invariant 1 : les cuves entrante, sortante et le contenu du tuyau doivent être corrects
- Invariant 2 : les deux portes entraintes et sortantes ne peuvent être simultanément ouvertes.

Définir en Haskell ces deux invariants ainsi que l'invariant `prop_pipeInvariant` qui les combine.

L'initialisation d'un tuyau se fait de la manière suivante :

```
initPipe :: Tank -> Tank -> Integer -> Maybe Pipe
initPipe tkIn tkOut cap =
  case initTank cap of
    Nothing -> Nothing
    Just tk -> Just $ Pipe tkIn tkOut False False tk
```

Donner les hypothèses nécessaires et montrer que si un tuyau est correctement initialisé, alors il respecte son invariant.

Question 2.4 : Ouverture/Fermeture des portes

Donner une définition de l'opération d'ouverture ou de fermeture des portes, selon le type suivant.

```
data DoorSide = IN | OUT
  deriving (Show)

switchDoor :: Pipe -> DoorSide -> Pipe
```

On considéra les cas suivants :

- ouverture de la porte entrante initialement ouverte
- ouverture de la porte entrante initialement fermée
- ouverture de la porte sortante initialement ouverte
- ouverture de la porte sortante initialement fermée

Justifier la correction de l'opération du point de vue de l'invariant du tuyau.

Question 2.5 : version sûre

Définir une variante sûre de l'opération d'ouverture/fermeture, selon le type suivant :

```
safeSwitchDoor :: Pipe -> DoorSide -> Maybe Pipe
```

...