Spécifications en logique du premier ordre – TD2

Exercice 1 – *Retour sur le TD 1*

Étant donnée la logique multi-sortes de signature (`Nat`, `Bin`, {`binToNat`}, $\{=, \neq\}$) où `Nat`, `Bin` et `binToNat` sont comme vu lors du TD1. Donner la spécification de la fonction `natToBin` qui est l'inverse de la fonction `binToNat`.

Exercice 2 – *Fusion*

On considère le type des séquences d'entiers $Seq[\texttt{Int}]$ dont les constructeurs sont

$\varepsilon : Seq[\texttt{Int}]$, la séquence vide,

$\bullet : \texttt{Int} \times Seq[\texttt{Int}] \to Seq[\texttt{Int}]$, l'ajout en tête (à gauche) d'un élément à une séquence.

Etant donnée une séquence s, on désigne par $|s|$ la longueur de la séquence s. On considère que $|\varepsilon| = 0$. Pour tout entier naturel $1 \leqslant i \leqslant |s|$, on désigne par $s[i]$ l'élément se trouvant à la position i dans s.

Soit la fonction

$$\texttt{Fusion} : Seq[\texttt{Int}] \times Seq[\texttt{Int}] \to Seq[\texttt{Int}]$$

qui étant données deux séquences triées s et s', fusionne ces séquences. (On suppose qu'il est possible d'avoir plusieurs occurrences d'un même entier dans une séquence.)

1. Ecrire une spécification formelle de la fonction `Fusion` en logique de premier ordre reliant ses données et son résultat.

2. Donner une implémentation récursive de cette spécification.

On considère la fonction

$$\texttt{Partition} : Seq[\texttt{Int}] \to Seq[\texttt{Int}] \times Seq[\texttt{Int}]$$

dont la spécification est la suivante :

$$\texttt{Partition}(s) = (s_1, s_2) \iff (|s| = |s_1| + |s_2|) \wedge (\forall i \cdot 1 \leqslant 2i \leqslant |s| \Rightarrow s[2i] = s_1[i]) \wedge (\forall i \cdot 0 \leqslant 2i < |s| \Rightarrow s[2i+1] = s_2[i])$$

3. Donner une implémentation récursive de tri `FusionSort` basée sur les fonctions `Partition` et `Fusion`.

4. Donner une implémentation récursive de la fonction `Partition`.

Exercice 3 – *Arbres binaires*

On considère le type des arbres binaires d'entiers :

```
Inductive BinTree:=
| Leaf: Int -> BinTree
| Node: BinTree × BinTree -> BinTree
```

On considère la fonction

$$\texttt{Flatten} : \texttt{BinTree} \to \texttt{List}[\texttt{Int}]$$

qui prend un arbre en argument et renvoie la liste des éléments dans les feuilles de l'arbre.

1. Peut on donner la spécification de la fonction `Flatten` en supposant une signature avec les fonctions suivantes :
   - `Find: BinTree x Int -> Bool`
   - `Mem: List[Int] x Int -> Bool`

2. Donner la spécification complète de `Flatten`. (NB : On peut utiliser les `Multiset`).

Exercice 4 – *Arbres binaires de recherche*

On utilise la définition d'arbre binaire de l'Exercice 3 pour définir les Arbres Binaires de Recherche (ABR).

1. Donnez la propriété qu'indique qu'un arbre binaire est bien un ABR. (NB : La spécification même pourrait être récursive.)

2. Donnez l'implémentation d'une fonction pour vérifier si un `BinTree` est un ABR.

3. Donnez la définition inductive des fonctions suivantes pour les ABR :
   `Find: BinTree` $\times$ `Int -> Bool`
   `Insert: BinTree` $\times$ `Int -> BinTree`

4. Donnez la spécification de la fonction `Insert` et prouvez la correction de la définition récursive donnée par rapport à cette spécification.

5. Donnez l'implémentation de la traversée de l'arbre en profondeur d'abord avec le type

   `InOrder: BinTree -> List[Int]`

6. Prouvez que, si l'arbre d'entrée est un ABR, la liste produite par `InOrder` est ordonnée.