. .

ENSIIE S5 - PFM - Utilisation du système Coq - 2019-2020

Modalités de remise du projet

A faire pour le 24 octobre 2019 minuit - dépot électronique sur `exam.ensiie.fr` (le nom du dépot est `pfm-2019`)
A faire en binôme

Pour la remise du devoir, il est attendu une archive contenant un fichier .v (le numéro de chaque exercice et question sera indiqué en commentaire) ainsi qu'un document pdf, tout deux comportant dans leur nom le binôme. Cette archive sera déposée sur le site habituel. Le code sera accompagné d'un document au format pdf contenant les définitions et les énoncés des théorèmes ainsi que quelques commentaires (précisez en particulier si une preuve est complète ou non). Des outils existent pour les produire automatiquement, coqdoc par exemple.

Remarque : si vous ne parvenez pas à démontrer un lemme, admettez le (commande `Admitted`) et continuez.
N'hésitez pas à me poser des questions !

Quelques indications générales

- On travaille sur les entiers naturels de Coq de type `nat`. Ne les redéfinissez pas.
- Il se peut que pour démontrer un lemme, on ait besoin d'un lemme démontré précédemment. Dans ce cas, utilisez `apply` suivi du nom du lemme ou `rewrite` suivi du nom du lemme.
- Quelques tactiques supplémentaires pour manipuler des égalités.
  - Pour passer d'un but `S x = S y` à un but `x = y`, appliquer le lemme `f_equal`.

  ```
  Check f_equal.
       : forall (A B : Type) (f : A -> B) (x y : A),
         x = y -> f x = f y
  ```

  - Si on a une hypothèse `H : S x = S y` (resp. `x::l = y::r`), `injection H` déduira la nouvelle hypothèse `x = y` (resp les hypothèses `x = y` et `l=r`). Ceci fonctionne avec n'importe quel type inductif.
  - Si l'hypothèse `H` concerne deux constructeurs différents (par exemple `0 = S x` ou `nil = x::l`) alors `discriminate H` terminera la preuve du but en question.
- Pour *éliminer* une hypothèse de la forme H : exists x:T, P, on utilisera la tactique suivante :`destruct H as [t Ht.` Ceci a pour effet de rajouter dans le contexte des hypothèses un témoin `t` de type `T` (`t:T`) et la preuve que `t` vérifie `P` (`Ht : P[x <- t]`). Bien entendu le choix des identificateurs `t` et `Ht` est à votre convenance.
- Pour revenir à la définition d'une fonction ou d'un prédicat `f`, on utilise la tactique `unfold f`.
- Comment écrire dans une fonction une expression de la forme `si t1=t2 alors .. sinon..`

  Soit A un type muni d'une égalité décidable, ie d'un lemme `A_eq_dec` (ou axiome si A est abstrait) de la forme :

  ```
  forall (x y : A), ({x=y}+{~x=y}.
  ```

Cela veut dire que pour tout couple `(x, y)`, on est capable de donner une preuve de cette propriété, c'est soit une preuve de `x=y` (elle sera de la forme `left H` où `H` est une preuve de `x=y`) ou une preuve de `~x=y` (elle sera de la forme `right H'` où `H'` est une preuve de `~x=y`). `A_eq_dec x y` est donc de la forme `left H` ou `right H'`. Donc pour écrire `if x=y then e1 else e2`, il suffit de faire un filtrage sur `A_eq_dec x y` :

```
 match A_eq_dec x y with
  left _ => e1
| right _ => e2
end
```

On utilisera le même principe par exemple pour coder une expression de la forme `if x<=y then e1 else e2`. Si `x` et `y` sont des valeurs de type `nat`, on utilisera le lemme de décidabilité `le_lt_dec n m` qui exprime que soit on peut prouver `n<=m`, soit on peut prouver `m<n`).

```
Definition le_lt_dec n m : {n <= m} + {m < n}.
```

il suffit de faire un filtrage sur `le_lt_dec x y` :

```
 match le_lt_dec x y with
  left _ => e1
| right _ => e2
end
```

La première clause du filtrage correspond au cas où x est inférieur ou égal à y (le _ représente la preuve de $x \leq y$), la seconde clause correspond au cas où $x < y$.

Dans une preuve, pour faire une démonstration par cas : 1er cas `x <= y` 2ème cas `y < x`. On procèdera également en utilisant le lemme `le_lt_dec`. On examine les deux formes possibles de la preuve de `le_lt_dec x y` : avec la tactique `destruct (le_lt_dec x y).` Idem pour tout lemme de décidabilité.

Exercice 1 (**Listes et comptage**)

On utilisera le type des listes polymorphes défini dans la bibliothèque standard : `Require Import List`.

Dans la suite on travaille avec un type A quelconque. mais muni d'une égalité décidable.

```
Variable A : Type.

Hypothesis dec_A : forall (x y : A), ({x=y}+{~x=y}.
```

1. Écrire une fonction `occ` qui prend en argument un élément x de type A et une liste l de type (list A) et qui retourne le nombre (un nat) d'occurrences de x dans l.

2. La fonction `app` de la bibliothèque standard rélaise la concaténation de deux listes. Démontrer le théorème suivant :

   ```
   Theorem occ_app : forall (x : A) l1 l2,
   occ x (app l1 l2) = (occ x l1) + (occ x l2).
   ```

3. La fonction `filter` est elle-aussi définie dans la bibliothèque standard. Démontrer le théorème suivant : :

   ```
   Theorem occ_filter : forall (P : A -> bool) (a : A) l,
   occ a (filter P l) = if (P a) then occ a l else 0.
   ```

4. Ecrire la fonction `map` qui applique une fonction f de type `A -> A` sur tous les éléments d'une liste. Que pensez-vous de l'énoncé suivant ?

   ```
   Theorem occ_map ; forall (f : A -> A) (x : A) l,
     occ (f x) (map f l) = occ x l.
   ```

   Essayez de le prouver ou prouvez-en une variante.

5. Le prédicat inductif `mem` ci-dessous définit l'appartenance d'un élément à une liste.

   ```
   Inductive mem : A -> list A -> Prop :=
   | mem_cons : forall x l, mem x (cons x l)
   | mem_tail : forall x y l, mem x l -> mem x (cons y l).
   ```

   Démontrer que si le nombre d'occurrences d'un élement dans une liste est nul alors cet élément n'appartient pas à la liste, et réciproquement.

6. ```
   Inductive nodup : list A -> Prop :=
   | nodup_nil : nodup nil
   | nodup_tail : forall x l, ~mem x l -> nodup l ->  nodup (cons x l)
   .
   ```

   Le prédicat `(nodup l)` spécifie que la liste l n'a pas de doublons. Démontrer qu'une liste est sans doublons si et seulement si pour tout x le nombre d'occurrences de x dans l est inférieur ou égal à 1.

Exercice 2 (**Implantation des multi-ensembles**)

On veut définir le type `multiset` des multi-ensembles contenant des éléments de type T, T étant un type abstrait.

```
Variable T : Type.
```

On supposera que l'égalité est décidable sur T :

```
Hypothesis T_eq_dec : forall (x y : T), {x=y} + {~x=y}.
```

Ceci se lit : pour tout x et y de type T, on a soit x=y soit ¬(x=y). Plus précisément on a une preuve de x=y ou une preuve du contraire.

On veut définir les constantes et fonctions suivantes :

```
empty :  multiset
singleton :  T -> multiset
member :  T -> multiset -> bool
add : T -> nat -> multiset -> multiset
union : multiset -> multiset -> multiset
multiplicity : T -> multiset -> nat
removeOne : T -> multiset -> multiset
removeAll : T -> multiset -> multiset
```

Spécification informelle

`empty` est le multiset vide.
`member x s` retourne la valeur `true` si x a au moins une occurrence dans s, `false` sinon.
`singleton x` crée le multi-ensemble qui ne contient que x en un seul exemplaire.
`add x n s` ajoute, au multi-ensemble s, n occurrences de l'élément x dans s.
`union` fait l'union de deux multi-ensembles.
`muliplicity x s` retroune le nombre d'occurrences de x dans s.
`removeOne x s` retourne le mult-ensemble s avec une occurrence de moins pour x. Si s ne contient pas x, le multi-ensemble résultat est s.
`removeAll x s` retourne le mult-ensemble s où x n'apparait plus. Si s ne contient pas x, le multi-ensemble résultat est s.
```

2.1 Implantation des multi-ensembles à l'aide de listes d'association

1. Implanter les fonctions précédentes en représentant un multi-ensemble par une liste d'association formée de couples de la forme (x,n) où n est le nombre d'occurrences de l'élément x dans le multi-ensemble ainsi représenté.

   ```
   Definition multiset := list (T*nat).
   ```

   On fera l'hypothèse qu'il n'y a qu'un seul couple par élément dans le multi-ensemble. Tous les couples de la liste comportent un nombre d'occurrences strictement positif. Ainsi si T est Z, le multi-ensemble contenant 3 fois 1 et 2 fois -1 est représenté par la liste [(1, 3) ; (-1, 2). Les couples peuvent être dans n'importe quel ordre.

   Vous utiliserez les listes et les entiers naturels de la bibilothèque standard. Réutilisez `add` pour définir `union`.

   Pour tester vos fonctions dans Coq, il vous suffit d'instancier (donner une valeur à) `T` et son lemme de décidabilité, par exemple remplacer au début de votre fichier `Variable T : Type.` par `Definition T := nat.` et la ligne `Hypothesis T_eq_dec : forall (x y : t), {x=y} + {~x=y }.` par `Definition T_eq_dec := Nat.eq_dec.`

2. On s'intéresse maintenant à la correction des fonctions précédentes. On spécifie ici les propriétés attendues par les fonctions précédentes.

   (a) Cette spécification s'appuie sur le prédicat `InMultiset` de type `T -> multiset -> Prop` qui spécifie qu'un élément appartient à un multi-ensemble dès lors qu'il en existe une occurrence.

   Définir le prédicat `InMultiset`.

   (b) Définir le prédicat `wf` qui spécifie qu'une liste qui représente un multi-ensemble est bien formée, i.e que tout élément de `T` apparaît dans au plus un seul couple et que tous les nombres d'occurrences sont des entiers naturels non nuls.

   (c) Démontrer que les fonctions `empty` et `singleton` produisent un résutlat bien formé et que les fonctions `add`, `union`, `removeOne` et `removeAll` préservent la propriété de bonne formation.

3. Démontrer ensuite les propriétés ci-dessous :

   ```
   forall x, ~ InMultiset x empty.

   forall x y , InMultiset y (singleton x) <-> x = y.

   forall x, multiplicity x (singleton x) = 1.

   forall x s, wf s -> (member x s = true  <-> InMultiset x s).

   forall x n s, n > 0 -> InMultiset x (add x n s).

   forall x y s, x <> y -> (InMultiset y (add x n t) <-> InMultiset y s).

   forall x s, wf s -> (multiplicity x s = 0 <-> ~InMultiset x s).

   forall x n s, multiplicity x (add x n s) = n + (multiplicity x s).

   forall x n y s, x <> y -> wf s ->
      multiplicity y (add x n s t) = multiplicity y s.
   ```

```
forall s t x, wf s -> wf t ->
  (InMultiset x (union s t) <-> InMultiset x s \/ InMultiset x t).
```

N'hésitez pas à introduire des lemmes intermédiaires si besoin. Vous pouvez décomposer les équivalences en deux théorèmes démontrés séparément.

4. Enoncez des propriétés similaires pour `removeOne` et `removeAll` et démontrez-les.

2.2 Implantation Fonctionnelle des multi-ensembles

On reprend ici l'implantation des multi-ensembles en représentant un multi-ensemble par une fonction de type `T -> nat` qui encode les multiplicités. Si un élément n'est pas présent dans le multi-ensemble, la fonction l'associe à 0.

1. Redéfinir les fonctions précédentes.
2. Redéfinir le prédicat `InMultiset`.
3. Démontrer les propriétés précédentes (il n'est plus besoin ici de prédicat de bonne formation).