. .

ENSIIE S5 - PFM - Utilisation du système Coq - 2018-2019

Modalités de remise du projet

A faire pour le 6 novembre 2018 minuit - dépot électronique sur `exam.ensiie.fr` (le nom du dépot est `pfm-2018`)
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 les tactiques suivantes : `elim H ; intros 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 `elim (le_lt_dec x y).`

Exercice 1 (**Pour se faire la main sur les listes**)

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

Il est défini de la façon suivante :

```
Inductive list (T : Type) : Type :=
    nil : list T | cons : T -> list T -> list T
```

`T` est le type des éléments de la liste. Il est implicite (vous n'avez en général pas besoin de l'écrire, Coq l'infère quand c'est possible).

Dans la suite on travaille avec un type A quelconque.

```
Variable A : Type.
```

1. Écrire une fonction `repeat` qui prend en argument un élément x de type A et n un entier naturel et qui construit une liste composée de n fois x.

2. Démontrer le théorème suivant :

   ```
   Theorem repeat_length : forall x n,
       length (repeat x n) = n.
   ```

3. 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 le théorème suivant :

```
Theorem repeat_spec : forall n x y,
  mem y (repeat x n) -> y=x.
```

4. Écrire une fonction `alternate` qui prend en argument deux éléments x et y de type A et n un entier naturel et qui construit une liste composée alternativement de x et de y, de n fois.

5. Démontrer le théorème suivant :

```
Theorem alternate_length : forall x n,
    length (alternate x y n) = 2*n.
```

6. Démontrer le théorème suivant :

```
Theorem alternate_spec : forall n x y z,
  mem z (repeat x y n) -> z=x \/ z=y.
```

Exercice 2 (**Plus petit élément et élément minimal**)

Soit `E` un ensemble et `R` une relation binaire sur `E` réflexive, antisymétrique et transitive.

On définit les notions de plus petit élément `smallest` pour `R` et d'élément minimal `minimal` pour `R` .

```
Variable E : Set.
Variable R : E -> E -> Prop.

Hypothesis R_refl : forall (x : E), R x x.
Hypothesis R_antisym : forall (x y : E), R x y -> R y x -> x=y.
Hypothesis R_trans : forall (x y z : E), R x y -> R y z -> R x z.

Definition smallest (x0 : E) := forall x : E, R x0 x.
Definition minimal (x0 : E) := forall x : E, R x x0 -> x=x0.
```

Formaliser et prouver les propriétés suivantes :

1. Si R admet un plus petit élément alors celui-ci est unique.

2. S'il existe un plus petit élément, alors il est un élement minimal

3. Si R admet un plus petit élément alors il n'y a pas d'autre élément minimal que celui ci. Ou dit autrement si R admet un plu petit élément et un élément minimal alors ils sont égaux.

Exercice 3 (**Implantation d'une table**)

On veut définir le type `table` des tables de U dans T, U et T étant deux types abstraits. On appelle clés (de type T) les entrées d'une table et et infos (de type U) les valeurs associées aux clés.

```
Variables T U : 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.

Remarque : dans une fonction, pour distinguer les cas x=y et x $\neq y$, *onécrira* :

```
match (T_eq_dec x y) with
  (left _) => (* on est ici dans le x=y*)
| (right _) =>(* on est ici dans le ~x=y*)
end
```

On veut définir les constantes et fonctions suivantes :

```
empty :  table
find : table -> T -> option U
add : table -> T -> U -> table
```

Remarque : le type `option` est prédéfini en Coq. Tapez `Print option` pour voir sa définition.

Spécification informelle
`empty` est la table vide. Aucune clé n'a de valeur dans la table vide.
`find t x` retourne la valeur `v` associée à `x` dans la table `t` si la clé `x` est dans le domaine de la table. Dans ce cas le résultat sera `Some v`. S'il n'y a pas de valeur pour `x`, alors le résultat sera `None`.
`add t x v` construit une nouvelle table où la nouvelle info associée à la clé `x` est v et les valeurs des autres clés sont inchangées.

Spécification formelle (spécification algébrique)
On spécifie les relations entre `find`, `empty` et `add`.

```
forall x, find empty x = None.                                    (prop1)

forall t x v, find  (add x v t) x = Some v.                       (prop2)

forall t x y v, x <> y ->   find  (add x v t) y = find t y.  (prop3)
```

On va s'intéresser à trois implantations différentes du type `table`. Pour répondre aux questions, il peut être nécessaire d'introduire des fonctions intermédiaires et des lemmes intermédiaires.

1. On représente une table par une liste d'association. Une liste d'association est une liste de couples (clé, info) (de type `list (U*T)`) où le premier composant clé est une clé d'identification et info une ou plusieurs informations. Dans cette implémentation on pourra avoir plusieurs couples concernant la même clé. La fonction `add` ajoutera le nouveau couple en tête alors que la fonction `find` s'arrêtera à la première occurrence de la clé.

   (a) Définir la constante `empty` et les fonctions `find` et `add`.

   (b) Démontrer les 3 propriétés de la spécification.

2. On impose maintenant la contrainte que la liste d'association représentant la table ne contient pas deux couples concernant la même clé. Ou autrement dit le domaine de la table, la liste des clés qui ont une valeur associée, est une liste sans doublons. On dit alors que la liste de couples est bien formée.

   (a) Définir la constante `empty` et les fonctions `find` et `add`.

   (b) Définir un prédicat `wf` (de type `list (U*T) -> Prop`) qui spécifie la bonne formation d'une liste d'association.

   (c) Démontrer que la table `vide` est bien formée.

   (d) Démontrer que la fonction `add` préserve la bonne formation : si une table `t` est bien formée, alors la table `add t x v` est encore bien formée.

   (e) Démontrer les 3 propriétés de la spécification.

3. On se place ici dans le cas où le type `U` est muni d'une relation d'ordre. Reprendre la question précédente en imposant en plus que les couples d'une table sont ordonnés selon la relation d'ordre sur les clés.