. .

ENSIIE S5 - PRFM - Utilisation du système Rocq - 2025-2026

Modalités de remise du projet

A faire en binôme pour le 17 novembre 2025 minuit - dépôt électronique sur `exam.ensiie.fr` (le nom du dépôt est `prfm-2025`)

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 (`Admitted` ou lieu de `Qed`) 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.

Partie 1 : Examen du contenu d'une liste

On utilise le type des listes polymorphes défini dans la bibliothèque standard : `Require Import List`. La commande `Import ListNotations` vous permet d'utiliser les notations *à la OCaml* (`[]` et `I::`). On utilise également les entiers naturels prédéfinis dans `Arith`.

Le prédicat `Forall` défini dans la bibliothèque standard permet de spécifier qu'une liste l contient des éléments qui satisfont tous un prédicat P. Voir la définition de `Forall` et du théorème `Forall_forall`, ci-dessous.

```
Inductive Forall (A : Type) (P : A -> Prop) : list A -> Prop :=
    Forall_nil : Forall P []
  | Forall_cons : forall (x : A) (l : list A),
                  P x -> Forall P l -> Forall P (x :: l).

Lemma Forall_forall:   forall (A : Type) (P : A -> Prop) (l : list A),
  Forall P l <-> (forall x : A, In x l -> P x).
```

1. Définir une fonction `repeat` qui prend en argument un élément `a` d'un type `A` quelconque et un entier naturel `n` et qui produit une liste contenant `n` occurrences de `a`.

   Démontrer que la fonction `repeat` est correcte, i.e. que tous les éléments de la liste résultat sont égaux à `a` et que sa longueur est `n`.

   ```
   Lemma repeat_sound1 :  forall (A : Type) (a : A) n,
        Forall (fun x => x=a) (repeat A a n).
   Lemma repeat_sound2 :  forall (A : Type) (a : A) n,
       length (repeat A a n) = n.
   ```

2. (a) Ecrire une fonction `split_p` qui éclate une liste l (de type list A) en deux sous-listes l1 et l2. La séparation se fait sur le premier élément v de l qui satisfait une fonction booléenne P (de type `A -> bool`) donnée. La liste l1 contient tous les éléments de l qui précèdent v alors que la liste l2 contient v puis tous les éléments de l qui suivent. Par exemple si l est la liste [1;3;5;4;1;6;8;9], alors `split_p is_even l` retourne le couple de listes `([1;3;5], [4;1;6;8;9]` . Si la liste est vide, le résultat est `([], []`. Si aucun élément de l ne satisfait le prédicat alors le résultat est `(l, []`. L'ordre des éléments dans l doit être respecté dans le résultat.

```
Eval compute in (split_p (fun x => x =? 0) [1;2;0;4;0;5]).
(*     = ([1; 2], [0; 4; 0; 5])
       : list nat * list nat *)

Eval compute in (split_p (fun x => x =? 0) [1;2;1;4;1;5]).
(*      = ([1; 2; 1; 4; 1; 5], [])
        : list nat * list nat    *)

Eval compute in (split_p (fun x => x =? 0) (repeat nat 1 5)).
(*    = ([1; 1; 1; 1; 1], [])
      : list nat * list nat  *)
```

Remarque : `=?` désigne une fonction de type `nat -> nat -> bool`, elle teste si ses deux arguments sont égaux. L'égalité `=` est un prédicat logique binaire. Donc si `x` est de type `nat`, `x=0` est une proposition logique (de type `Prop`) alors que `x =?  0` est une expression de type `bool`.

(b) Démontrer les lemmes suivants :

```
Lemma split_p_first : forall (A : Type) (p : A -> bool) (l  : list A),
 Forall (fun x => p x = false) (fst (split_p p l)) .

Lemma split_p_snd : forall (A : Type) (p : A -> bool) (l l1 l2  : list A) x,
 split_p p l = (l1, l2) -> head l2 = Some x -> p x = true.

Lemma split_p_forall : forall (A : Type) (p : A -> bool) (l  : list A),
 Forall (fun x => p x = true) l -> split_p p l = ([], l)  .
```

(c) De la même façon démontrer que si aucun des éléments de l ne vérifient p alors `split_p p l = (l, [])`.

(d) Démontrer que la concaténation des listes du couple résultat de `split_p` donne la liste initiale. Vous utiliserez la fonction `append` de la bibliothèque standard.

(e) Démontrer que la longueur des listes du couple résultat de `split_p` est inférieure ou égale à la longueur de la liste initiale.

3. En utilisant ce qui précède, définir une fonction `split_occ` qui éclate une liste (de type list A) en deux sous-listes l1 et l2. La séparation se fait sur la première occurrence d'un élément v de l donné. La liste l1 contient tous les éléments de l qui précèdent v (et qui sont donc différents de v) alors que la liste l2 contient v puis tous les éléments de l qui suivent. On aura besoin ici d'un type `A` décidable.

   Démontrer la correction de `split_occ` : l1 ne contient pas v, l2 commence par v et l1 ++ l2 = l. Là aussi vous réutiliserez les lemmes démontrés à la question 2.

4. **(Difficile - Ne faire qu'après avoir fait la deuxième partie)**

   (a) Ecrire la fonction `split_p_all` qui généralise la fonction `split_p` de manière à retourner un couple `(l1, L)`. `l1)` est comme précédemment la liste des premiers éléments de `l` qui ne vérifient pas `p`. `L` est une liste de listes. Chacune des listes commence par un élément

qui satisfait `p` et contient ceux qui suivent jusqu'au prochain élément qui satisfait `p`. Ainsi `split_p_all is_even [1; 3; 2; 5;7;8;10;11]` a pour résultat `([1;3] ; [[2;5;7] ; [8] ; [10;11]]`.

(b) Démontrer que la première liste de couple résultat (`l1`) ne contient que des éléments qui ne satisfont pas `p`.

(c) Démontrer que tous les premiers éléments des listes de `L` vérifient `p` et les suivants de chaque liste ne vérifient pas `p`.

(d) Démontrer que la concaténation de `l1` et de toutes les listes de `L` forment exactement `l`. Vous pourrez utiliser la fonction `concat` de la bibliothèque standard ainsi que les lemmes qui s'y rapportent. Celle-ci permet de concaténer toutes les listes d'une liste.

Implantation des multi-ensembles

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

```
Parameter
Variable T : Type.
```

On supposera que l'égalité est décidable sur T :

```
Parameter
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
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.
`multiplicity x s` retourne le nombre d'occurrences de x dans s.
`removeOne x s` retourne le multi-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 multi-ensemble s où x n'apparait plus. Si s ne contient pas x, le multi-ensemble résultat est s.

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

~~Un module implémentant les listes d'association sera fourni ultérieurement. Il devra être utilisé pour implémenter les multi-ensembles.~~

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.

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`~~ `Parameter` `T : Type.` par `Definition T := nat.` et la ligne ~~`Hypothesis`~~ `Parameter` `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` (de manière directe ou de manière inductive).

   (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 résultat bien formé et que les fonctions `add`, `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.
```

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.