Devoir Maison : Inverser deux fois une liste

Logique et Fondements de l'Informatique

Modalités de rendu.

Le fichier à compléter est `dm.v`. C'est un fichier texte contenant du code Rocq. Voici la marche à suivre :

1. **Ouvrez jsCoq** dans votre navigateur (`https://jscoq.github.io/scratchpad.html`).
2. **Copiez-collez** le contenu du fichier `dm.v` dans l'éditeur jsCoq.
3. **Travaillez dans jsCoq** : complétez les preuves et les réponses aux questions directement dans l'éditeur. Vous pouvez exécuter le code pas à pas pour vérifier vos preuves au fur et à mesure.
4. **Enregistrez** votre travail en cliquant sur le bouton de sauvegarde (💾) dans jsCoq. Cela téléchargera un fichier `.v` sur votre ordinateur.
5. **Rendez** le fichier `.v` téléchargé par email à `pablo.donato@tuta.io`.

Tout votre travail (preuves *et* réponses aux questions de compréhension) doit se trouver dans ce fichier unique. Les réponses en français se rédigent dans des commentaires Coq, entre `(*` et `*)`.

Ce DM vous guide vers la preuve d'un résultat classique de vérification formelle :

$$\forall A,\ \forall l : \text{list } A, \quad \texttt{rev}\ (\texttt{rev}\ l) = l$$

c'est-à-dire qu'inverser deux fois une liste redonne la liste de départ.

Cette preuve ne peut pas se faire directement : elle nécessite des résultats intermédiaires, appelés **lemmes auxiliaires**. En mathématiques comme en programmation, on décompose un problème complexe en sous-problèmes plus simples. En Rocq, cela se traduit par des lemmes qu'on prouve séparément, puis qu'on réutilise avec la tactique `rewrite` [1].

Les exercices 1 à 3 établissent les lemmes dont vous aurez besoin pour l'exercice 4. Remplacez `admit` par votre preuve. Quand vous avez terminé un exercice, remplacez `Admitted` par `Qed` — Rocq vérifiera que votre preuve est complète.

Rappel des tactiques utiles :

- `intros` : introduire les hypothèses
- `induction l as [| h t IH]` : raisonner par induction sur une liste
- `simpl` : simplifier (appliquer les définitions)
- `rewrite lemme` : réécrire le but en utilisant un lemme ou une hypothèse
- `rewrite <- lemme` : réécrire de droite à gauche
- `reflexivity` : conclure quand les deux côtés sont égaux

1. On utilise plus généralement la tactique `apply` si la conclusion du lemme n'est pas une égalité. Dans ce DM on prouve uniquement des égalités.

Définitions de référence. On utilise `rev` et `app` (noté `++`) de la bibliothèque standard :

```
Fixpoint app {A : Type} (l1 l2 : list A) : list A :=
  match l1 with
  | []     => l2
  | h :: t => h :: (app t l2)
  end.

Fixpoint rev {A : Type} (l : list A) : list A :=
  match l with
  | []     => []
  | h :: t => rev t ++ [h]
  end.
```

Exemple : utiliser `rewrite` avec un lemme.

Supposons qu'on ait déjà prouvé le lemme suivant :

```
Lemma add_0_r : forall n : nat, n + 0 = n.
```

Dans une preuve ultérieure, si le but contient `n + 0`, on peut utiliser `rewrite add_0_r` pour le remplacer par `n` :

```
(* But courant :  f (n + 0) = f n *)
rewrite add_0_r.
(* Nouveau but :  f n = f n *)
reflexivity.
```

De même, `rewrite <- add_0_r` ferait le remplacement inverse : de `n` vers `n + 0`.

1. Concaténer avec la liste vide ne fait rien

Indice : `app` est défini par récursion sur son *premier* argument. Donc `[] ++ l` se réduit par définition, mais `l ++ []` ne se réduit pas si `l` est une variable. Procédez par induction sur `l`.

```
Lemma app_nil_r : forall (A : Type) (l : list A),
    l ++ [] = l.
```

2. La concaténation est associative

Indice : même structure que l'exercice 1, par induction sur `l1`.

```
Lemma app_assoc : forall (A : Type) (l1 l2 l3 : list A),
    (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
```

3. Inverser une concaténation

Indice : induction sur `l1`. Vous aurez besoin des exercices 1 et 2.

```
Lemma rev_app : forall (A : Type) (l1 l2 : list A),
    rev (l1 ++ l2) = rev l2 ++ rev l1.
```

4. Inverser deux fois redonne la liste de départ

C'est le résultat principal.

Indice : induction sur `l`. Vous aurez besoin de l'exercice 3.

```
Theorem rev_involutive : forall (A : Type) (l : list A),
    rev (rev l) = l.
```

Questions de compréhension

Répondez aux questions suivantes en quelques phrases, directement dans le fichier `dm.v`, en commentaires Coq (entre `(*` et `*)`).

Question 1. Pourquoi ne peut-on pas prouver `rev (rev l) = l` simplement avec `reflexivity`, sans induction ?

Question 2. Quel est le rôle du lemme `rev_app` dans la preuve de `rev_involutive` ? Pourquoi est-il nécessaire ?

Question 3. Les exercices 1 à 3 sont des lemmes auxiliaires. Aurait-on pu s'en passer et tout prouver d'un seul coup dans l'exercice 4 ? Quel est l'intérêt de décomposer la preuve en lemmes ?