Vérification (HAI603I)

Licence Informatique
Département Informatique
Faculté des Sciences de Montpellier
Université de Montpellier

TD/TP N°5 : Preuve de correction du tri par insertion

Dans ce TD/TP, l'objectif est de démontrer la correction de l'algorithme de tri par insertion d'une liste d'entiers naturels. Cette démonstration sera progressive. On va d'abord écrire la spécification, puis la fonction de tri elle-même, pour finir par la preuve de correction. La preuve sera découpée en plusieurs lemmes intermédiaires et il faudra la mécaniser. L'intégralité du TD/TP se fera en utilisant l'outil Coq. En prélude de votre fichier Coq, mettez les commandes suivantes qui vous permettront de charger les bibliothèques dont nous allons avoir besoin dans notre développement :

```
1 Require Import Arith.
2 Require Import Omega.
3 Require Export List.
4 Open Scope list_scope.
5 Import ListNotations.
```

Exercice 1 (Spécification)

Cet exercice correspond à l'exercice 4 du TD/TP N°3. Si vous l'avez déjà réalisé en Coq, vous pouvez passer à l'exercice suivant. Les deux relations « être triée » et « être une permutation de » que nous allons définir permettent de spécifier ce que va faire notre fonction de tri. En effet, une fonction va trier une liste l si elle rend une liste l' telle que l' est triée et l' est une permutation de l.

1. Spécifier la relation « être une permutation de » pour deux listes.
2. Démontrer que la liste $[1; 2; 3]$ est une permutation de $[3; 2; 1]$.
3. Spécifier la relation « être triée » pour une liste.
4. Démontrer que la liste $[1; 2; 3]$ est triée.

Exercice 2 (Fonction de tri par insertion)

Dans cet exercice, nous allons écrire la fonction de tri par insertion proprement dite. Cette fonction s'écrit en deux temps. Nous allons d'abord écrire la fonction d'insertion dans une liste triée, puis la fonction de tri elle-même. Les deux fonctions sont récursives (structurelles) et s'écrivent par récursion sur la liste prise en entrée.

1. Avant d'écrire les fonctions, on doit d'abord s'intéresser à comment on fait des tests (conditionnelles) dans les fonctions en Coq. Le problème n'est pas si simple car il est possible de définir des relations non décidables en Coq. Par exemple, l'égalité sur les nombres réels (pas les flottants) n'est pas décidable. Ainsi, si on met un test d'égalité entre deux nombres réels en Coq, toute exécution se bloquera sur ce test qui ne pourra pas se faire. Donc, tout test doit se faire sur une relation décidable et cela se fait en utilisant un lemme qui exprime que la relation est décidable.

   Par exemple, la relation $\leq$ est décidable sur les entiers naturels et c'est le lemme `le_dec` qui le dit. Vérifier le type du lemme `le_dec` avec la commande `Check`.

   Ce lemme utilise le type `sumbool` qui est un type inductif qui prend deux propositions `A` et `B` en paramètres. Pour ce type, Coq utilise le sucre syntaxique suivant `{A} + {B}`. Afficher la définition de `sumbool` avec la commande `Print`.

   Le type `sumbool` correspond à la définition d'un "ou" constructif en ce sens que l'on ne peut le démontrer qu'en utilisant l'un de ses deux constructeurs : soit `left` avec une preuve de `A`, soit `right` avec une preuve de `B`.

   Ainsi, si on revient sur `(le_dec n m)`, ce lemme peut être vu comme une fonction qui rend soit `left` si `n` $\leq$ `m`, soit `right` si `n` $\not\leq$ `m`.

   Afin de jouer un peu avec `le_dec`, on peut définir la fonction `le_10` qui prend un entier n en paramètre et retourne `true` si $n \leq 10$ ou `false` sinon (on utilise ici le type des booléens de Coq) :

```
Definition le_10 (n : nat) : bool :=
  match (le_dec n 10) with
  | left _ ⇒ true
  | right _ ⇒ false
  end.
```

   Tester la fonction `le_10` avec les entiers 5 et 15. Pour ce faire, on utilisera la commande « `Eval compute in` » suivie de l'application de fonction proprement dite.

2. Écrire la fonction *insert* qui insère de manière triée un entier x dans une liste triée l (la liste rendue reste triée). Cette fonction est récursive. La récursion est structurelle et se fait sur la liste l prise en entrée. La signature de la fonction en Coq est la suivante :

```
Fixpoint insert (x : nat) (l : list nat) {struct l} : list nat := ...
```

   Le test à effectuer dans cette fonction devra utiliser le lemme de décidabilité `le_dec` vu à la question précédente.

3. Tester la fonction *insert* avec l'entier 3 et la liste $[1; 2; 4; 5]$. Pour ce faire, on utilisera la commande « `Eval compute in` » comme précédemment.

4. Écrire la fonction *isort* de tri par insertion qui prend une liste l et rend une liste l' triée qui est une permutation de l. Elle devra utiliser la fonction d'insertion *insert* écrite précédemment. Cette fonction est également récursive. La récursion est structurelle et se fait sur la liste l prise en entrée. La signature de la fonction en Coq est la suivante :

```
Fixpoint isort (l : list nat) : list nat := ...
```

5. Tester la fonction *isort* avec la liste $[5; 4; 3; 2; 1]$ (le pire des cas en termes de complexité). On utilisera la commande « `Eval compute in` » comme précédemment.

Exercice 3 (Preuve de correction)

Nous nous proposons maintenant de faire la preuve de correction de la fonction *isort* de tri par insertion en Coq. Nous allons d'abord exprimer le théorème de correction, puis nous détaillerons la preuve progressivement. La preuve nécessite de poser un certain nombre de lemmes intermédiaires et vous serez guidés pas à pas vers la preuve du théorème final. La plupart des preuves se réalisent par induction et nous vous indiquerons quelle induction réaliser à chaque fois. Dans ce qui suit, les relations `is_perm` et `is_sorted` correspondent respectivement aux relations « être une permutation de » et « être triée » définies dans l'exercice 1.

1. Démontrer le lemme suivant :

```
Lemma head_is_perm : forall (x1 x2 : nat) (l : list nat),
  is_perm (x1 :: x2 :: l) (x2 :: x1 :: l).
```

   La preuve de ce lemme est directe (aucune induction n'est nécessaire). Il suffit d'appliquer les constructeurs de la relation `is_perm`.

2. Démontrer le lemme suivant :

```
Lemma insert_is_perm : forall (x : nat) (l : list nat),
  is_perm (x::l) (insert x l).
```

   La preuve de ce lemme se fait par induction structurelle sur la liste `l`. Cette preuve fera appel (dans le cas inductif) au lemme `head_is_perm` démontré à la question 1. Dans cette preuve, il y aura également une expression de la forme suivante qui va apparaître :

```
if (le_dec x a) then e1 else e2
```

   où `e1` et `e2` sont des termes. Cette expression provient du `match` effectué sur le type `sumbool` dans la fonction `insert`. Cette expression est complètement équivalente à :

```
match (le_dec x a) with
| left _ ⇒ e1
| right _ ⇒ e2
end
```

   Ce type d'expressions peut se simplifier en faisant une preuve par cas. En effet, si $\texttt{x} \leq \texttt{a}$ alors elle se réduit sur `e1`, sinon elle se réduit sur `e2`. Pour réaliser cette preuve par cas en Coq, il suffit d'utiliser la tactique « `elim (le_dec x a)` », qui va générer deux buts suivant les deux cas dans lesquels l'expression aura été simplifiée soit par `e1`, soit par `e2`.

3. Démontrer le lemme suivant :

```
Lemma insert_is_sorted : forall (x : nat) (l : list nat),
  is_sorted l → is_sorted (insert x l).
```

   La preuve de ce lemme se fait par induction structurelle sur la relation `(is_sorted l)`. Cette preuve est la plus longue (mais pas forcément difficile) car il y a une petite combinatoire entre 3 entiers induite par des termes de la forme `(le_dec a b)`, qu'il faudra simplifier avec la tactique `elim` comme vu à la question précédente.

4. Démontrer le théorème de correction, qui est exprimé comme suit :

```
Lemma isort_correct : forall (l l' : list nat),
  l' = isort l → is_perm l l' ∧ is_sorted l'.
```

   La preuve de ce lemme se fait par induction structurelle sur la liste `l`. Cette preuve fera appel au lemme `insert_is_perm` démontré à la question 2, ainsi qu'au lemme `insert_is_sorted` démontré à la question 3. Dans cette preuve, il y aura également une hypothèse (l'hypothèse d'induction pour être plus précis) de la forme `H : A → (B ∧ C)` à éliminer. C'est quelque chose que nous avons déjà vu avec le « ∧ » mais pas avec une « → » devant. Ici, ce qui va changer, c'est que l'élimination de `H` va produire les buts habituels avec en plus l'obligation de démontrer `A`. Ainsi, si le but à démontrer est `D` avec `H` en hypothèse, la tactique « `elim H` » va produire les buts `B → D`, `C → D` et `A`.

Exercice 3 (Preuve de complétude)

Nous nous proposons maintenant de faire la preuve de complétude de la fonction *isort* de tri par insertion en Coq. Nous allons d'abord exprimer le théorème de complétude, puis nous détaillerons la preuve progressivement. La preuve nécessite de poser un certain nombre de lemmes intermédiaires et vous serez guidés pas à pas vers la preuve du théorème final. La plupart des preuves se réalisent par induction et nous vous indiquerons quelle induction réaliser à chaque fois. Dans ce qui suit, les relations `is_perm` et `is_sorted` correspondent respectivement aux relations « être une permutation de » et « être triée » définies dans l'exercice 1.

Par chacune des preuves demandées ci-après, opérez d'abord une réflexion sur papier, tablette ou autres supports d'écriture avant de mécaniser la preuve avec Coq.

1. Démontrer le lemme suivant :

```
Lemma is_sorted_cons_inv :
  forall (l : list nat) (a : nat), is_sorted (a :: l) → is_sorted l.
```

   La preuve de ce lemme se fait par analyse de cas sur la structure de la liste `l`. Pour ce faire, nous utiliserons la tactique `destruct`. Appliquée à `l`, cette tactique va générer une branche de preuve où `l` sera la liste vide et une branche de preuve où `l` sera de la forme élément de tête suivie d'une queue de liste.

   Vous aurez également besoin de raisonner sur la définition de la relation `is_sorted` via la tactique `inversion` (pendant de `destruct` pour les propositions). Si votre contexte

contient une hypothèse `H : is_sorted l`, le résultat de `inversion H` sera de générer trois branches de preuve, c.-à-d. une par cas de construction de `is_sorted l`. Vous pouvez tester le comportement de la tactique `inversion` sur le but suivant :

```
Goal forall (l : list nat), is_sorted l → True.
```

2. Démontrer le lemme suivant :

```
Lemma is_sorted_app :
  forall (l : list nat) (a : nat), is_sorted (l ++ [a]) → is_sorted l.
```

La preuve se fait par induction structurelle sur la liste `l`. Pour la mener à bien, vous aurez besoin :

- des lemmes `app_comm_cons` et `app_cons_not_nil`, fournis par la librairie standard. Utilisez la commande `Check` pour vous renseignez sur leur contenu.
- de la tactique `injection` qui permet de générer des égalités en se basant sur l'injectivité des constructeurs de types inductifs. Par exemple, si votre contexte contient l'égalité `EQ : a :: l = b :: m`, c.-à-d. la liste `a :: m` est égale à la liste `b :: m`, le résultat de `injection EQ` sera de générer une hypothèse d'égalité entre les deux têtes de listes, c.-à-d. `a = b`, et une hypothèse d'égalité entre les deux queues de listes, c.-à-d. `l = m`.
- de la tactique `generalize`. Si le but à prouver est `C`, le résultat de `generalize H`, où `H : A` est une hypothèse du contexte, sera de transformer le but en `A → C`. Cette tactique est utile pour faire correspondre exactement le but à prouver avec la conclusion d'un lemme ou d'un théorème à appliquer.

3. Démontrer le lemme suivant :

```
Lemma is_sorted_id : forall l, is_sorted l → isort l = l.
```

La preuve se fait par induction structurelle sur la relation `is_sorted`.

4. Démontrer le lemme suivant :

```
Lemma is_sorted_insert_id :
  forall l a, is_sorted (a :: l) → insert a l = a :: l.
```

Le preuve se fait par induction structurelle sur la liste `l`.

5. Démontrer le lemme suivant :

```
Lemma is_sorted_not_lt_tl_hd :
  forall l a x, is_sorted (a :: l ++ [x]) → x < a → False.
```

Le preuve se fait par induction structurelle sur la liste `l`. Attention, comme la conclusion du lemme est `False`, il faudra générer une contradiction en raisonnant sur la définition de la relation `is_sorted` (avec la tactique `inversion`). Utilisez le solveur `lia` pour fermer une branche de preuve lorsqu'une contradiction, basée sur un système d'équations linéaires, apparaît dans le contexte.

6. Démontrer le lemme suivant :

```
Lemma is_sorted_app_insert :
  forall l a, is_sorted (l ++ [a]) → insert a l = l ++ [a].
```

Le preuve se fait par induction structurelle sur la liste `l`. Pour la mener à bien, vous aurez besoin d'utiliser les lemmes `is_sorted_cons_inv`, `is_sorted_not_lt_tl_hd`, `is_sorted_insert_id`, et `is_sorted_app` que nous venons de prouver.

7. Démontrer le lemme suivant :

```
  Lemma isort_eq_cons_app :
  forall l a, isort (a :: l) = isort (l ++ [a]).
```

La preuve se fait par induction structurelle sur la liste `l`. Pour mener à bien la preuve, vous aurez besoin du lemme suivant, que vous admettrez pour l'instant.

```
  Lemma insert_comm_assoc :
  forall l x y, insert x (insert y l) = insert y (insert x l).
  Admitted.
```

En question bonus, vous pouvez prouver le lemme `insert_comm_assoc` par induction sur `l`. Essayez d'obtenir un script de preuve de moins de 10 lignes, sans définir de tactique avec `Ltac`.

8. Démontrer le lemme suivant :

```
  Lemma is_perm_eq_isort :
  forall l m, is_perm l m → isort l = isort m.
```

La preuve se fait par induction sur la relation `is_perm`, et vous aurez bien sûr recours au lemme `isort_eq_cons_app` qui vient d'être prouvé.

9. Nous voilà maintenant prêt à prouver le théorème de complétude pour la fonction `isort`. Ce théorème s'exprime comme suit :

```
Lemma is_sort_compl : forall (l l' : list nat),
    is_perm l l' → is_sorted l' → isort l = l'.
```

La preuve se fait en combinant un raisonnement par induction sur la relation `is_perm`, un raisonnement sur la définition de la relation `is_sorted`, et en utilisant les lemmes prouvés jusqu'à présent. Opérez d'abord une réflexion sur papier, tablette ou autres supports d'écriture avant de mécaniser la preuve avec Coq.