TP Logique n° 4. Preuve de programmes fonctionnels avec Coq.

Soit un ensemble A donné, l'ensemble $liste(A)$ des listes contenant des éléments de A est défini par les opérateurs `Nil` et `Cons` comme le **plus petit** ensemble de termes tels que :

1. $\texttt{Nil} \in \texttt{liste}(A)$
2. $\forall t \in A. \forall q \in \texttt{liste}(A).\ \texttt{Cons}(t, q) \in \texttt{liste}(A)$

Cette définition est équivalente au plus petit point fixe de l'équation : $\texttt{liste}(A) = \{\texttt{Nil}\} \cup \{\texttt{Cons}(t, q) \mid t \in A, q \in \texttt{liste}(A)\}$.

La concaténation de deux listes est définie sous la forme d'équations entre termes :

$$\text{(a) } \forall y \in \texttt{liste}(A).\ \ \text{append}(\texttt{Nil},\, y) = y$$
$$\text{(b) } \forall x \in A. \forall y, z \in \texttt{liste}(A).\ \ \text{append}(\texttt{Cons}(x,\, z),\, y) = \texttt{Cons}(x,\, \text{append}(z,\, y))$$

Soit la définition suivante de la fonction rev :

$$\text{(c) } \text{rev}(\texttt{Nil}) = \texttt{Nil}$$
$$\text{(d) } \forall x \in A. \forall y \in \texttt{liste}(A).\ \ \text{rev}(\texttt{Cons}(x,\, y)) = \text{append}(\text{rev}(y),\, \texttt{Cons}(x,\, \texttt{Nil}))$$

L'objectif de cette séance est de prouver des propriétés des fonctions `append` et `rev`.

Vous disposez sur MOODLE du fichier `induction-etu.v` contenant les éléments nécessaires pour la séance commentés par la suite.

1 Préliminaires

Pour utiliser Coq pour prouver des propriétés sur ces fonctions, il est nécessaire d'effectuer les commandes :

```
(* Bibliothèque pour l'extraction de programme. *)
Require Import Extraction.
(* Ouverture d'une section *)
Section Induction.
(* Déclaration d'un domaine pour les éléments des listes *)
Variable A : Set.
```

2 Types inductifs

Soit la définition inductive du type `liste` d'éléments de `A` :

```
Inductive liste : Set :=
   Nil    : liste
   | Cons : A -> liste -> liste.
```

Cette définition engendre 3 définitions utilisées par Coq pour les preuves par induction structurelle (`liste_rect` (itérateur sur la structure), `liste_ind` (principe de preuve par induction) et `liste_rec` (principe de calcul par induction)) en plus de la définition du type inductif `liste` comportant les deux constructeurs `Nil` et `Cons`. Ces définitions spécialisées sont exploitées par les tactiques `elim` et `induction` pour l'hypothèse `l : liste`.

$$\frac{\Gamma \vdash [\texttt{Nil}/\texttt{l}]\varphi \quad \Gamma \vdash \forall t\ t \in \texttt{A} \to \forall q\ q \in \texttt{liste} \to [q/\texttt{l}]\varphi \to [(\texttt{Cons}\,t\,q)/\texttt{l}]\varphi}{\Gamma,\, \texttt{l} : \texttt{liste} \vdash \varphi} \ (\texttt{elim l.})$$

$$\frac{\Gamma \vdash [\texttt{Nil}/\texttt{l}]\varphi \quad \Gamma,\, \texttt{t} : \texttt{A},\, \texttt{q} : \texttt{liste},\, \texttt{Hq} : [\texttt{q}/\texttt{l}]\varphi \vdash [(\texttt{Cons}\,\texttt{t}\,\texttt{q})/\texttt{l}]\varphi}{\Gamma \vdash \forall \texttt{l}\ \texttt{l} \in \texttt{liste} \to \varphi} \ (\texttt{induction l.})$$

3 Spécification de fonctions

La fonction `append` sera spécifiée en Coq sous le nom `append_spec` par les deux axiomes suivants :

```
(* Déclaration du nom de la fonction *)
Variable append_spec : liste -> liste -> liste.

(* Spécification du comportement pour Nil *)
Axiom append_Nil : forall (l : liste), append_spec Nil l = l.

(* Spécification du comportement pour Cons *)
Axiom append_Cons : forall (t : A), forall (q l : liste),
                    append_spec (Cons t q) l = Cons t (append_spec q l).
```

Les preuves sur la relation d'équivalence = peuvent être manipulées en Coq par les tactiques `reflexivity`, `symmetry` et `transitivity t`.

$$\Gamma \vdash t = t \quad (\texttt{reflexivity.}) \qquad \frac{\Gamma \vdash d = g}{\Gamma \vdash g = d} \quad (\texttt{symmetry.}) \qquad \frac{\Gamma \vdash g = \texttt{t} \quad \Gamma \vdash \texttt{t} = d}{\Gamma \vdash g = d} \quad (\texttt{transitivity t.})$$

Les équations de la forme `H` : $G = D$ peuvent être appliquées dans un but par les tactiques `rewrite H.` qui applique l'équation de gauche à droite (identique à `rewrite -> H.`) ; et `rewrite <- H.` qui applique l'équation de droite à gauche.

Nous allons d'abord prouver des propriétés de la spécification de la fonction `append` qui seront satisfaites par toutes les implantations.

1. Démontrez :

   ```
   Theorem append_Nil_right : forall (l : liste), (append_spec l Nil) = l.
   ```

2. Démontrez :

   ```
   Theorem append_associative : forall (l1 l2 l3 : liste),
            (append_spec l1 (append_spec l2 l3)) = (append_spec (append_spec l1 l2) l3).
   ```

4 Implantation de fonctions

Coq contient également un langage de spécification de fonctions semblable aux langages fonctionnels tels OCaML, $F^\sharp$ ou Haskell, l'implantation `append_impl` de la fonction `append` est définie par le point fixe suivant :

```
Fixpoint append_impl (l1 l2 : liste) {struct l1} : liste :=
   match l1 with
      Nil            => l2
      | (Cons t1 q1) => (Cons t1 (append_impl q1 l2))
   end.
```

L'annotation `{struct l1}` indique que la terminaison du calcul de la fonction `append_impl` quelles que soient les valeurs de ses paramètres `l1` et `l2` se prouve par induction structurelle sur `t1`. Coq génère alors automatiquement la preuve de terminaison.

La tactique `simpl` en Coq permet de calculer les parties d'un but qui sont calculables (par exemple, l'application de la fonction `append_impl` sur des paramètres dont la structure est partiellement connue (par exemple, `Cons t q` a une structure de liste contenant la tête `t` et la queue `q`, le calcul de `append_impl (Const t q) l` renverra `Cons t (append_impl q l)`)).

Nous allons maintenant prouver la correction de cette implantation par rapport à la spécification manipulée dans la section précédente.

1. Démontrez :

   ```
   Theorem append_correctness : forall (l1 l2 : liste),
            (append_spec l1 l2) = (append_impl l1 l2).
   ```

Nous disposons donc d'une implantation en Coq correcte vis-à-vis de la spécification. Nous pouvons extraire de celle-ci une version en OCaML, Scheme et Haskell. avec la commande `Recursive Extraction append_impl.`. Attention, il faut auparavant fermer la section `End Induction.`. Vous poursuivrez les exercices avant la fin de section.

Voici les commandes qui permettent d'extraire l'implantation vers les langages OCaML, Scheme et Haskell.

```
End Induction.

Extraction Language Ocaml.
Extraction "/tmp/induction" append_impl rev_impl.
Extraction Language Haskell.
Extraction "/tmp/induction" append_impl rev_impl.
Extraction Language Scheme.
Extraction "/tmp/induction" append_impl rev_impl.
```

5 Propriétés de la fonction rev

Nous nous focaliserons uniquement sur l'implantation `rev_impl` de la fonction qui renverse le contenu d'une liste. Dans le cas idéal, il faudrait spécifier celle-ci, étudier les propriétés sur la spécification, l'implanter et prouver la correction de l'implantation comme nous l'avons fait pour la fonction `append`.

1. Définissez en Coq la fonction `rev_impl`. Attention, faites ceci avant `End Induction.`.
2. Démontrez le lemme suivant :

   ```
   Lemma rev_append : forall (l1 l2 : liste),
               (rev_impl (append_impl l1 l2)) = (append_impl (rev_impl l2) (rev_impl l1)).
   ```

3. Démontrez :

   ```
   Theorem rev_rev : forall (l : liste), (rev_impl (rev_impl l)) = l.
   ```

TP Logique n° 5. Atelier de vérification Why3

6 Préliminaires

Ce sujet consiste à utiliser l'atelier de vérification WHY3 pour démontrer la correction de programmes fonctionnels comme nous l'avons fait avec COQ précédemment. WHY3 repose sur un langage de modélisation combinant logique du premier ordre pour la spécification, programmation fonctionnelle et impérative pour l'implantation. WHY3 est une passerelle vers de nombreux outils de vérification partiellement (assistants de preuve) ou totalement (techniques SAT et SMT) automatisés.

1. Si vous ne l'avez pas fait dans la séance précédente, lancer la commande `why3 config detect` depuis la même fenêtre de commande pour configurer *Why3* en fonction des outils de preuve disponible dans votre environnement. Si vous avez déjà utilisé cet outil les années précédentes, il est préférable de supprimer le fichier de configuration avec la commande `rm $HOME/.why3.conf`.

Vous disposez sur MOODLE du fichier `induction-etu.mlw` contenant les éléments nécessaires pour la séance commentés par la suite.

Attention, si vous modifiez le fichier `induction-etu.mlw`, que ce soit dans l'éditeur de `why3` ou dans un autre éditeur, vous devez le recharger dans `why3` soit en quittant l'outil et en le relançant, soit en utilisant la commande `Save all and Refresh session` du menu `File`.

7 Preuve de correction de la fonction append

Nous allons d'abord créer une théorie contenant la définition du type liste et l'implantation de la fonction `append`.

```
theory Induction

type liste 'a = Nil | Cons 'a (liste 'a)

function append (l1 l2 : liste 'a) : liste 'a =
   match l1 with
      | Nil -> l2
      | Cons t q -> Cons t (append q l2)
   end

lemma append_Nil_left : forall l : liste 'a. append Nil l = l

lemma append_Cons_left : forall e : 'a. forall l1 l2 : liste 'a.
   append (Cons e l1) l2 = Cons e (append l1 l2)

lemma append_Nil_right: forall l : liste 'a. append l Nil = l

lemma append_associative : forall l1 l2 l3 : liste 'a.
   append l1 (append l2 l3) = append (append l1 l2) l3

end
```

1. Étudiez le fichier `induction-etu.mlw` disponible sur MOODLE. Celui-ci comporte des :
   — déclaration du type algébrique générique `liste` d'éléments de sorte `'a`;
   — déclaration de la fonction `append`;
   — déclarations de lemme préfixé par le mot clé `lemma` avec un nom commençant par une minuscule. Ce sont des propriétés qui doivent être satisfaites par la fonction `append`.
2. Démarrer WHY3 avec : `why3 ide induction-etu.mlw`. L'outil présente :
   — les différents objectifs de preuve (`goal` en Why3) de manière arborescente;
   — les outils de preuve automatique ou assistée qui peuvent être utilisés pour effectuer les preuves;

— effectuer les preuves automatiques des lemmes `append_Nil_left` et `append_Cons_left` en utilisant un des outils SAT/SMT ;
— tenter d'effectuer la preuve automatique des lemmes `append_Nil_right` et `append_associative`. Les outils de preuve automatique échouent ;

3. Sélectionner le but `append_Nil_right`. Sélectionner l'assistant de preuve `Coq`, soit dans le menu `Tools/Provers`, soit par un clic droit sur le but. Une tentative de preuve apparaît dans l'onglet associé au but. Sélectionner cette tentative puis sélectionner la commande `Edit` soit dans le menu `Tools`, soit par un clic droit sur la tentative de preuve, qui lancera `coqide`. Vous pouvez réaliser la preuve en `Coq` rapidement en combinant l'induction (élimination des variables de type `liste`) avec la tactique `tauto` et sauvegarder la preuve depuis `coqide`. Après avoir quitté `coqide`, vous pouvez jouer cette preuve en sélectionnant la commande `Replay` dans le menu `Tools` ou par un clic droit sur le but.
4. Pour automatiser la vérification de cette preuve dans l'atelier WHY3, il faut utiliser la transformation `induction_ty_lex` (menu `Tools/transformations (e-r)`) qui explicite la preuve par induction sur un terme. Sélectionner le lemme, appliquer la transformation qui introduit un sous-objectif de preuve. Sélectionner ce sous-objectif puis le consulter dans l'onglet `Task` à droite de l'atelier. Appliquer les outils de preuve automatique.
5. Prouver de la même manière le lemme `append_associative`. Si le lemme comporte plusieurs variables de terme, il faut préciser sur quelle variable l'induction sera faite.

```
lemma append_associative : forall l1 [@induction] l2 l3 : liste 'a.
```

8 Preuve de correction de la fonction reverse

1. Ajouter dans le fichier `induction-etu.mlw` la fonction `rev` et les lemmes nécessaires à la preuve que :

$$\forall x \in \texttt{liste}(A).\ \ \mathrm{rev}(\mathrm{rev}(x)) = x$$

2. Construire la preuve de la propriété précédente en utilisant les outils de preuve automatique.