Chapter 1

theme08logicprogstudent

```
Require Import Arith.
Require Import List.
```

Dans ce thème, nous allons exploiter l'expressivité de la construction `Inductive` de Coq pour notamment dépasser certaines limitations du modèle de calcul "total" sous-jacent.

Du point de vue de la programmation fonctionnelle, `Inductive` peut être vu comme une version "sur-vitaminée" des déclarations de types en Ocaml ou Haskell. C'était le point de vue du thème précédent, dans lequel nous avons défini des *types de données* : listes, arbres, etc.

Il est également possible de spécifier des types plus "calculatoires" qui ne sont pas exprimables dans les langages de type usuels. Ces types sont de nature *relationelle*. Plutôt que d'exprimer une *fonction* avec `Definition` ou `Fixpoint`, on va exprimer une *relation* avec `Inductive`.

1.1 Fonctions calculables vs. relations logiques

Les fonctions sont bien sûr des cas particuliers de relations, donc commençons par une fonction simple.

```
Fixpoint sum_fun (n:nat) : nat :=
  match n with
    | 0 => 0
    | S m => n + sum_fun m
  end.
```

La fonction *sum_fun* est définie ici dans le langage fonctionnel encodé dans la logique de Coq. Il y a de nombreux avantages à utiliser ce langage. Tout d'abord, on peut facilement l'utiliser pour effectuer des calculs.

```
Example sum_fun_5:
  sum_fun 5 = 15.
Proof.
  compute. reflexivity.
Qed.

Example sum_fun_10:
  sum_fun 10 = 55.
Proof.
  compute. reflexivity.
Qed.
```

D'autre part, la possibilité de calculer avec les fonctions permet de simplifier certaines étapes de preuve, en utilisant notamment `simpl`.

```
Lemma two_times:
  ∀ n : nat, 2 × n = n + n.
Proof.
  intro n. ring.
Qed.

Theorem sum_fun_Gauss:
  ∀ n : nat, 2 × (sum_fun n) = n × (n + 1).
Proof.
  intro n.
  rewrite two_times.
  induction n as [| n'].
  - (*  cas n=0 *)
    simpl.
    reflexivity.
  - (*  cas n=S n' *)
    simpl.
    (*  SearchPattern (S _ = S _).
       eq_S: forall x y : nat, x = y -> S x = S y *)
    apply eq_S.
    assert (H: n' + sum_fun n' + S (n' + sum_fun n')
                  = sum_fun n' + sum_fun n' + n' + S n').
    { ring. }
    rewrite H. clear H.
    rewrite IHn'.
    ring.
Qed.
```

Comme l'illustre l'exemple ci-dessus, les simplifications par calcul aident les preuves mais ne les résolvent pas. On doit toujours décider "manuellement" du schéma de preuve. De plus, l'utilisation de procédures de décision spécifiques (comme `ring`) est souvent nécessaire.

La fonction qui, à un entier naturel n, associe la somme $\sum_{i=0}^{n} i$ peut être vue de façon alternative comme une relation.

Considérons le type inductif suivant :

```
Inductive sum_rel : nat → nat → Set :=
| sum_O: sum_rel 0 0
| sum_S: ∀ n s:nat,
           sum_rel n s → sum_rel (S n) ( (S n) + s).
```

Ce type définit une relation qui est comparable, mais pas identique, à la fonction *sum_fun*. Pour comprendre cette relation, le mieux est sans doute de l'interpréter comme un système de preuve, i.e. le plus petit ensemble satisfaisant les règles suivantes :

$$\frac{}{sum\_rel\ 0\ 0}\ (sum\_O) \quad \frac{sum\_rel\ n\ s}{sum\_rel\ (Sn)\ ((Sn)+s)}\ (sum\_S)$$

Par exemple, pour montrer que $\sum_{i=0}^{3} = 6$ on construit l'arbre de dérivation suivant :

$$\cfrac{\cfrac{\cfrac{\cfrac{}{sum\_rel\ 0\ 0}\ (sum\_O)}{sum\_rel\ (S0)\ ((S0)+0) = sum\_rel\ 1\ 1}\ (sum\_S)}{sum\_rel\ (S1)\ ((S1)+1) = sum\_rel\ 2\ 3}\ (sum\_S)}{sum\_rel\ (S2)\ ((S2)+3) = sum\_rel\ 3\ 6}\ (sum\_S)$$

On peut reproduire ce raisonnement en Coq :

```
Example sum_rel_3:
  sum_rel 3 6.
Proof.
  change (sum_rel 3 (3 + 3)).
  apply sum_S.
  change (sum_rel 2 (2 + 1)).
  apply sum_S.
  change (sum_rel 1 (1 + 0)).
  apply sum_S.
  apply sum_O.
Qed.
```

Si on compare à la version fonctionnelle, le "calcul" avec la version relationelle est assez fastidieux car manuel. Notons qu'il existe des possibilités d'automatisation, notamment en utilisant la tactique `auto` (cf. manuel de référence).

Pour ce qui concerne les preuves, on perd les simplifications par calcul mais les étapes intéressantes sont souvent les mêmes.

```
Theorem sum_rel_Gauss:
  ∀ n s : nat, (sum_rel n s) → 2 × s = n × (n + 1).
Proof.
  induction n as [|n'].
  - (*  cas n=0 *)
    intros s Hsum.
    inversion Hsum.
    simpl.
    reflexivity.
  - (*  cas n=S n' *)
    intros s Hsum.
    inversion Hsum. clear Hsum n H H1.
    apply IHn' in H0. clear IHn'.
    rewrite two_times in *.
    assert (H1: S n' + s0 + (S n' + s0) = s0 + s0 + S n' + S n').
    { ring. }
    rewrite H1. clear H1.
    rewrite H0.
    ring.
Qed.
```

Il n'est pas du tout évident que *sum_fun_Gauss* soit plus facile que *sum_rel_Gauss*. On peut même dire que le schéma de preuve est un peu plus clair dans ce deuxième cas. Mais on retiendra surtout que la relative difficulté de l'exercice provient de la propriété que l'on essaye de prouver sur la fonction/relation plutôt que sur la manière dont elle a été encodée.

Il existe une troisième stratégie pour démontrer notre propriété : utiliser le principe d'induction directement associé au type inductif.

Voici une troisième version de notre lemme démontré cette fois-ci en utilisant effectivement les règles d'inférences *sum_O* et *sum_S* effectivement comme un système de preuve. On parle d'une *induction basée sur les règles* (*rule induction* en anglais), qui est une forme d'induction structurelle.

```
Theorem sum_rel_Gauss':
  ∀ n s : nat, (sum_rel n s) → 2 × s = n × (n + 1).
Proof.
  intros n s H.
  induction H. (*  rule induction *)
  - (*  cas sum_O *)
    simpl.
    reflexivity.
```

```
  - (*  cas sum_S *)
    rewrite two_times in *.
    assert (H1: S n + s + (S n + s) = s + s + S n + S n).
    { ring. }
    rewrite H1. clear H1.
    rewrite IHsum_rel.
    ring.
Qed.
```

On ne peut pas vraiment dire que cette dernière preuve est plus simple, car la difficulté arithmétique sous-jacente est toujours la même. Cependant, le schéma de preuve semble un peu plus adapté que le schéma inductif des entiers naturels de *Peano*.

1.1.1 Exercice

Dans cet exercice, nous allons redéfinir la concaténation sur les listes, selon le point de vue fonctionnel et le point de vue relationnel.

Question 1

Définir la fonction *concat_fun* réalisant la concaténation de deux listes *l1* et *l2*.

Question 2

Définir la relation *concat_rel* sous la forme d'un type inductif.

Remarque : il sera utile, dans un premier temps, de décrire cette relation sur papier sous la forme de deux règles d'inférence *concat_nil* et *concat_cons*.

Question 3

Démontrer (par cas) :

```
Lemma concat_fun_cons:
  ∀ A : Set, ∀ e : A, ∀ l1 l2 : list A,
    concat_fun (e::l1) l2 = e::(concat_fun l1 l2).
```

Question 4

Démontrer :

```
Lemma concat_rel_cons:
  ∀ A : Set, ∀ e : A, ∀ l1 l2 l3 : list A,
    concat_rel A l1 l2 l3 → concat_rel A (e::l1) l2 (e::l3).
```

Question 5

Démontrer (par induction sur *l1*) :

```
Lemma concat_fun_length:
  ∀ A : Set, ∀ l1 l2 : list A,
    length (concat_fun l1 l2) = (length l1) + (length l2).
```

Question 6

En utilisant la définition relationnelle, on "perd" la possibilité des simplications par calcul (avec la tactique `simpl`). En revanche, on peut naturellement exploiter le type inductif comme un schéma de preuve par induction.
Compléter la preuve suivante :

```
Lemma concat_rel_length:
  ∀ A : Set, ∀ l1 l2 l3 : list A,
    concat_rel A l1 l2 l3
    → length l3 = (length l1) + (length l2).
Proof.
  intros A l1 l2 l3 Hrel.
  induction Hrel. (*  rule induction *)
  - (*  cas concat_nil *)
    (*  «< COMPLETER ICI »> *)
  - (*  cas concat_cons *)
    (*  «< COMPLETER ICI »> *)
```

1.1.2 Exercice : appartenance à une liste

Dans cet exercice, nous revisitons la formalisation du prédicat d'appartenance d'un élément à une liste.
D'un point de vue fonctionnel, la définition du prédicat *is_in_fun* est paramétrée par :

- un type paramétrique A (dans `Set`)
- une hypothèse de décidabilité de l'égalité sur A
- un élément e de A

- une liste l de type *list A*

Le résultat est un booléen de type *bool*.
Voici la définition complète :

```
Section is_in.

Variable A : Set.
Variable A_eq_dec : ∀ a b : A, {a = b} + {a ≠ b}.

Fixpoint is_in_fun (e : A) (l : list A) : bool :=
  match l with
    | nil ⇒ false
    | e'::l' ⇒ match A_eq_dec e e' with
                  | left _ ⇒ true
                  | right _ ⇒ is_in_fun e l'
                end
  end.

End is_in.
```

Si du point de vue de la programmation “classique” la fonction *is_in_fun* possède deux paramètres “principaux”, pour être précis il en faut bien quatre comme le confirme la commande suivante :

```
Check is_in_fun.
```

```
is_in_fun :
  ∀ A : Set, (∀ a b : A, {a = b} + {a ≠ b}) → A → list A → bool
```

Voici un exemple d'utilisation du prédicat :

```
Example is_in_fun_ex1:
  is_in_fun nat (eq_nat_dec) 3 (1::2::3::4::5::nil) = true.
Proof.
  compute. reflexivity.
Qed.
```

Le paramètre de décidabilité de l'égalité sur A est nécessaire car les fonctions Coq doivent être *totales*. Dans le cas de *is_in_fun*, il faut donc toujours pouvoir décider par calcul si le résultat est *true* ou *false*. Ce résultat dépend de la valeur de l'expression *A_eq_dec e e'* dans la fonction, c'est-à-dire de la possibilité de décider par calcul si e et e' sont égaux ou non. Il existe en effet de nombreuses structures mathématiques pour lesquelles, dans le cas général, l'égalité n'est pas décidable, à commencer par les fonctions elles-mêmes. L'hypothèse de décidabilité de l'égalité est donc primordiale.
En revanche, dans l'approche relationnelle, il n'y a plus de contrainte de totalité, qui est la contrepartie principale de la “perte” de la simplification par calcul.

Question 1

Soit la relation inductive *is_in_rel* définie de la façon suivante :

$$\frac{A : Set \quad a : A \quad l : list\ A}{is\_in\_rel\ A\ a\ (a :: l)}\ (is\_in\_head)$$

$$\frac{A : Set \quad a\ b : A \quad l : list\ A \quad a \neq b \quad is\_in\_rel\ A\ a\ l}{is\_in\_rel\ A\ a\ (b :: l)}\ (is\_in\_tail)$$

Remarque : dans les types inductifs, il est en général préférable de rendre les différents constructeurs (i.e. règles) indépendantes donc si l'une est vraie, l'autre est fausse et *vice_versa*. Compléter la spécification suivante :

```
Inductive is_in_rel (A:Set) : A → list A → Prop :=
  is_in_head: (*  «A COMPLETER» *)
| is_in_tail: (*  «A COMPLETER» *)
```

Question 2

Montrer :

```
Example is_in_rel_ex1:
  is_in_rel nat 3 (1::2::3::4::5::nil).
```

Remarque : on pourra utiliser `auto` pour "résoudre" les inégalités.
Dessiner l'arbre de preuve correspondant.

1.1.3 Exercice : liste préfixe

Soit le prédicat fonctionnel défini de la façon suivante :

```
Section is_prefix.

Variable A : Set.
Variable A_eq_dec : ∀ a b : A, {a = b} + {a ≠ b}.

Fixpoint is_prefix_fun (l1 l2 : list A) : bool :=
  match l1 with
    | nil ⇒ true
    | e1::l1' ⇒ match l2 with
                  | nil ⇒ false
                  | e2::l2' ⇒ match A_eq_dec e1 e2 with
                                | left _ ⇒ is_prefix_fun l1' l2'
                                | right _ ⇒ false
                              end
```

```
                    end
  end.
End is_prefix.
```

Question 1

Définir la relation inductive *is_prefix_rel* correspondante.
Remarque : on pourra dans un premier temps définir un système de preuves avec deux règles *is_prefix_nil* et *is_prefix_cons*.

Question 2

En supposant une liste *l1* préfixe d'une liste *l2*, on souhaite montrer que tous les éléments de *l1* appartiennent également à *l2*.
La preuve utilisant la définition fonctionnelle est relativement complexe.
Compléter le schéma de preuve suivant :

```
Lemma is_prefix_is_in_fun:
  ∀ A : Set, ∀ A_eq_dec : ∀ a b : A, {a = b} + {a ≠ b},
    ∀ l2 l1 : list A,
      (is_prefix_fun A A_eq_dec l1 l2) = true
      → (∀ e : A, is_in_fun A A_eq_dec e l1 = true → is_in_fun A A_eq_dec e l2 = true).
Proof.
  intros A A_eq_dec.
  induction l2 as [| e2 l2'].
  - (*  cas l2 = nil *)
    (*  «< A COMPLETER »» *)
  - (*  cas l2=e2::l2' *)
    intros l1 H1 e H2.
    simpl.
    destruct (A_eq_dec e e2).
    + (*  cas e=e2 *)
      reflexivity.
    + (*  cas e<>e2 *)
      { destruct l1 as [| e1 l1'].
        - (*  cas l1=nil *)
          (*  «< A COMPLETER »> *)
        - (*  cas l1=e1::l1' *)
          { apply IHl2' with (l1:=l1').
              (*  «< A COMPLETER »> *)
          }
      }
Qed.
```

Question 3

Compléter maintenant la preuve relationnelle :

```
Lemma is_prefix_is_in_rel:
  ∀ A : Set,
    ∀ l1 l2 : list A,
      (is_prefix_rel A l1 l2)
      → (∀ e : A, is_in_rel A e l1 → is_in_rel A e l2).
Proof.
  intros A l1 l2 Hprefix.
  induction Hprefix.
  (*  «<A COMPLETER»> *)
```

1.2 Etude de cas : sémantiques opérationnelles

Les types inductifs de Coq permettent, nous l'avons vu, de définir des systèmes de preuve (axiomes et règles d'inférences). Les sémantiques opérationnelles de langages informatiques (programmation, DSL, etc.) sont souvent formalisées sour la forme de tels systèmes de preuve. On parle de *sémantiques opérationnelles structurées* (ou SOS). il s'agit donc d'un bon exemple d'exploitation des types inductifs dépendants.
Nous allons définir une toute petite partie d'un langage de programmation : les expressions arithmétiques dans les entiers relatifs.
Remarque : le langage formalisé ici est d'une très (trop) grande simplicité. Cependant, les propriétés que nous discutons ainsi et schémas de preuves associés se généralisent naturellement à la plupart des constructions de langages.
Nous commençons par importer la bibliothèque *ZArith* de Coq.

```
Require Import ZArith.

Open Scope Z_scope. (*  expressions arithmétiques dans les entiers relatifs
                          par défaut, plutôt que nat. *)
```

La syntaxe des expressions arithmétiques est donnée par le type inductif suivant :

```
Inductive ArithExpr : Set :=
| aval : Z → ArithExpr
| aplus : ArithExpr → ArithExpr → ArithExpr
| atimes: ArithExpr → ArithExpr → ArithExpr.
```

Remarque : on a omis les opérateurs aritmétiques : soustraction, division (entière), etc. Il est tout a fait intéressant d'ajouter ces extensions après un premier passage par l'intégralité du sujet.

1.2.1 Exercice : sémantique à grands pas

La *sémantique à grands pas* (ou *big step semantics*) d'un langage valué (comme notre mini-langage d'expressions arithmétiques) correspond à une relation entre un terme syntaxique et la valeur qui lui correspond.
Les règles de la sémantique opérationnelle à grands pas pour notre langage arithmétique sont données ci-dessous :

$$\frac{n \in \mathbb{Z}}{BigSem\ (aval\ n)\ n}\ (aval\_big) \qquad \frac{BigSem\ a_1\ v_1 \quad BigSem\ a_2\ v_2}{BigSem\ (aplus\ a_1\ a_2)\ (v_1 + v_2)}\ (aplus\_big)$$

$$\frac{BigSem\ a_1\ v_1 \quad BigSem\ a_2\ v_2}{BigSem\ (atimes\ a_1\ a_2)\ (v_1 * v_2)}\ (atimes\_big)$$

Question 1

Formaliser la sémantique opérationnelle à grand pas du langage *ArithExpr* sous la forme d'un type inductif *BigSem* implémentant les règles définies ci-dessus.

Question 2

Montrer :

```
Example big_ex1:
  BigSem (atimes (aplus (aval 2) (aval 3)) (aval 5))
          (*  ==> *) 25.
```

Question 3

On dispose de plusieurs techniques de preuve pour démontrer des propriétés sur notre langage. On peut notamment effectuer des preuves par induction sur la syntaxe.
Utiliser ce principe pour démontrer que la sémantique est déterministe, c'est-à-dire :

```
Theorem BigSem_determinist:
  ∀ a : ArithExpr, ∀ v1 v2 : Z,
    BigStepSem a v1
    → BigStepSem a v2
    → v1 = v2.
```

1.2.2 Exercice : sémantique à petits pas

Dans une sémantique à petits pas on s'intéresse non pas à la valeur finale d'un calcul mais aux étapes intermédiaires permettant d'obtenir cette valeur finale.
Voici un extrait de la sémantique à petits pas encodée sous la forme d'un type inductif :

```
Inductive SmallSem: ArithExpr -> ArithExpr -> Prop :=
| aplusl_small: forall a1 a1' a2: ArithExpr,
                      SmallSem a1 (* --> *) a1'
               (*------------------------------------------*)
                -> SmallSem (aplus a1 a2) (aplus a1' a2)
| aplusr_small: forall a1 a2 a2' : ArithExpr,
                      SmallSem a2 (* --> *) a2'
               (*------------------------------------------*)
                -> SmallSem (aplus a1 a2) (aplus a1 a2')
| aplusv_small: forall v1 v2 : Z,
                 (*------------------------------------------------*)
                SmallSem (aplus (aval v1) (aval v2)) (aval (v1 + v2))
```

Question 1

Représenter le système de preuve correspondant à cette sémantique.

Question 2

Ajouter les règles pour la multiplication :

- au système de preuve de la Question 1
- au type inductif *SmallSem*.

Question 3

Montrer par induction sur la syntaxe :

`Theorem` *Strong_Progress*:
  $\forall$ *a*: *ArithExpr*,
    value a $\vee$ $\exists$ *a'* : *ArithExpr*, *SmallSem a a'*.

Ce théorème dit de *progression forte* explique que les expressions arithmétique “non-triviales” peuvent toujours être “simplifiées”.
Remarque : la propriété *value* indique que l'expression est une valeur. Elle est définie de la façon suivante :

`Inductive` *value*: *ArithExpr* $\to$ `Prop` :=
| *const_value*: $\forall$ *n*, *value* (*aval n*).

1.2.3 Exercice : formes normales

La *forme normale* d'une expression arithmétique est définie de la façon suivante :

```
Definition NormalForm (a : ArithExpr) : Prop :=
  not (∃ a' : ArithExpr, SmallSem a a').
```

Question 1

Montrer qu'une valeur est en forme normale, c'est-à-dire :

```
Lemma value_is_NF:
  ∀ a : ArithExpr,
    value a → NormalForm a.
```

et

```
Lemma NF_is_value:
  ∀ a : ArithExpr,
    NormalForm a → value a.
```

Question 2

Les règles de *réduction* d'un terme sont les suivantes :

$$\frac{a : ArithExpr}{Reduce\ a\ a}\ (red\_refl) \quad \frac{a\ a'\ a'' : ArithExpr \quad SmallSem\ a\ a' \quad Reduce\ a'\ a''}{Reduce\ a\ a''}\ (red\_step)$$

En appliquant ces règles, on peut "trouver" la forme normale d'une expression arithmétique. Traduire les règles de réduction en un type inductif *Reduce*.

Question 3

Montrer (sans induction) :

```
Lemma Reduce_refl: ∀ a : ArithExpr,
  Reduce a a.
```

et

```
Lemma Reduce_Small: ∀ a a' : ArithExpr,
  SmallSem a a' → Reduce a a'.
```

Question 4

On souhaite montrer que la relation de réduction est transitive. Pour cela, on ne peut simplement utiliser une induction basée sur la syntaxe.

L'idée est d'exploiter une *induction basée sur les règles* de *Reduce*.
Démontrer selon ce principe :

`Lemma` *Reduce_trans*: $\forall$ *a a' a'' : ArithExpr*,
  Reduce a a' $\rightarrow$ *Reduce a' a''* $\rightarrow$ *Reduce a a''*.

1.2.4 Exercice : correspondance des sémantiques

Dans ce dernier exerice, nous souhaitons mettre en correspondance les sémantiques opérationnelles à grands et petits pas.
Le schéma de preuve est décomposé selon les questions suivantes.

Question 1

Montrer que la réduction est congruente pour l'addition, c'est-à-dire :

`Lemma` *Reduce_plus_congl*: $\forall$ *a1 a2 a1' : ArithExpr*,
  Reduce a1 a1'
  $\rightarrow$ *Reduce* (*aplus a1 a2*) (*aplus a1' a2*).

et

`Lemma` *Reduce_plus_congr*: $\forall$ *a1 a2 a2' : ArithExpr*,
  Reduce a2 a2'
  $\rightarrow$ *Reduce* (*aplus a1 a2*) (*aplus a1 a2'*).

Puis, en utilisant ces deux lemmes :

`Lemma` *Reduce_plus_congv*:
  $\forall$ *n1 n2 : Z*, $\forall$ *a1 a2 : ArithExpr*,
    Reduce a1 (*aval n1*)
    $\rightarrow$ *Reduce a2* (*aval n2*)
    $\rightarrow$ *Reduce* (*aplus a1 a2*) (*aval* (*n1*+*n2*)).

1.2.5 Question 2

Même question pour la multiplication.
Remarque : les détails de preuve sont quasiment du copier/coller.

1.2.6 Question 3

Montrer par induction sur la sémantique à grands pas :

`Lemma` *Big_implies_Small*:
  $\forall\ a : ArithExpr, \forall\ v : Z,$
    $BigSem\ a\ v \to Reduce\ a\ (aval\ v).$

Question 4

Pour passer de la sémantique à petits pas vers la sémantique à grands pas, c'est un peu plus complexe. On commence par le cas d'un petit pas unique.
Montrer :

`Lemma` *Small1_Big_implies_Big_plus*:
  $\forall\ a\ a' : ArithExpr, \forall\ v : Z,$
    $SmallSem\ a\ a'$
    $\to BigSem\ a'\ v$
    $\to BigSem\ a\ v.$

Remarque : on effectuera une induction sur la sémantique à petit pas, et on prendra soin de généraliser sur la variable v (avec la tactique `generalize`) avant d'appliquer le principe d'induction.

Question 5

Démontrer par induction sur les règles de réduction :

`Lemma` *SmallNF_implies_Big*:
  $\forall\ a\ a': ArithExpr,$
    $Reduce\ a\ a'$
    $\to NormalForm\ a'$
    $\to (\exists\ v : Z, (a' = aval\ v) \land (BigSem\ a\ v)).$

Question 6

En utilisant le lemme de la question précédente, ainsi que *value_is_NF* et *value*, démontrer :

`Lemma` *Small_implies_Big*:
  $\forall\ a : ArithExpr, \forall\ v : Z,$
    $Reduce\ a\ (aval\ v)$
    $\to BigSem\ a\ v.$

Question 7

Conclure :

```
Theorem Small_equiv_Big:
  ∀ a : ArithExpr, ∀ v : Z,
    Reduce a (aval v) ↔ BigSem a v.
```