Vérification de programmes et de preuves
Troisième partie : prouver des propriétés de programmes

Yves Bertot

September 2012

1 Introduction et motivation

Pour prouver que des programmes font ce que nous prévoyons, nous avons besoin de couvrir tous les cas possibles dans leur exécution. La plupart du temps, les programmes prennent leurs entrées dans des types avec un nombre infini d'éléments. Il est alors impossible de vérifier tous les cas d'entrée un par un. A la place, nous raisonnons sur des sous ensembles des types qui correspondent aux différents comportements possibles dans les programmes. Cela se fait habituellement en suivant la structure des programmes. Ainsi, nous raisonnons sur les différents cas qui apparaissent pendant l'exécution.

Quand les fonctions sont récursives, cette approche repose sur un outil logique complexe, appelé la *preuve par récurrence* (*proof by induction* en anglais). Dans ce cours, nous étudions aussi cet aspect, mais nous nous restreignons à l'étude de programmes qui calculent avec des nombres et des listes.

Il y a de nombreux moyens pour décrire le comportement attendu d'un programme. Dans ce cours, nous étudions une approche simple, où le comportement attendu est décrit à l'aide de programmes secondaires, qui sont utilisés soit pour produire des entrées spécifiques, soit pour effectuer des tests sur les résultats des calculs.

Par exemple, si nous avons écrit une fonction `evenb` qui calcule une valeur booléenne qui est vraie si et seulement si l'entrée est un nombre pair (le mot anglais pour pair est *even*) nous pouvons chercher à construire une fonction qui construit tous les nombres pairs.

```
Definition mult2 (n : nat) := 2 * n.
```

Ensuite nous voulons simplement montrer que `evenb` retourne la valeur attendue sur tous les résultats de `mult2` :

```
Lemma evenb_complete :
  forall n : nat, evenb (mult2 n) = true.
```

Nous pouvons aussi chercher à montrer que `evenb` n'accepte que des nombres qui sont pairs. Une façon de l'exprimer est l'énoncé suivant:

```
Lemma evenb_sound :
  forall n : nat, evenb n -> exists y : nat, n = mult2 y.
```

2 Raisonner sur les constructions de filtrage

Une construction de filtrage décrit plusieurs cas possibles d'exécution. Lorsque l'on prouve qu'un programme est correct, nous avons besoin de vérifier tous ces cas. Il existe des tactiques pour observer séparément chacun de ces cas. Nous verrons qu'il pourra arriver que certains de ces cas sont incohérents, parce qu'il mettent à jour des hypothèses comme `0 = 1` ou `true = false`. Dans d'autres cas, nous aurons des égalités de la forme `a::b = c::d`, desquelles on peut tirer séparément `a = c` et `b = d`. Nous verrons quelles sont les tactiques qui permettent de traiter ces situations.

2.1 case, destruct, and case_eq

Observons le but suivant :

```
  x : nat
  ======================
   match x with 0 => true | S p => negb (even p) end = false
   -> x <> 0
```

La conclusion de ce but contient une construction de filtrage. Nous ne connaissons pas la valeur de `x`, mais nous savons que si `x` est `0`, la construction de filtrage va s'exécuter pour donner la valeur `true`. Dans l'autre cas, nous ne savons pas dire quel sera le résultat de l'exécution de la construction de filtrage, mais nous savons que le membre droit de l'implication deviendra `S y <> 0` pour un certain `y`.

Il y a trois tactiques qui permettent d'exprimer cette décomposition en deux cas, avec des comportements légèrement différents. Premièrement, observons le comportement de la tactique `destruct`. Elle produit autant de buts qu'il y a de cas dans le type de l'argument. Ici, nous utiliserons la tactique `destruct` et `x` a le type `nat`. Ce type `nat` a deux cas (deux *constructeurs*) et nous avons donc deux cas : soit `x` est 0, soit `x` est `S n` pour un certain nombre naturel `n`.

```
destruct x.
2 subgoals
  ======================
   true = false -> 0 <> 0

Subgoal 2 is
   negb(even n) -> S n <> 0
```

Donc, la tactique produit simplement deux instances du but où `x` est remplacé par des cas issus du type. Dans le premier goal, toutes les occurrences de `x` sont remplacées par `0`, puis la construction de filtrage est calculé.

Dans cet exemple, les deux buts sont prouvables parce qu'ils contiennent des égalités qui ne peuvent pas être satisfaite. Cette situation est traitée par la tactique que nous verrons à la prochaine section.

La tactique `case` effectue approximativement les mêmes opérations que la tactique `destruct`, sauf que son effet n'a lieu que dans la conclusion du but et que la donnée surlaquelle a lieu le traitement par cas n'est pas enlevée du contexte. Si nous reprenons l'exemple au début, nous observons le comportement suivant :

```
  x : nat
  ======================
   match x with 0 => true | S p => negb (even p) end = false
   -> x <> 0
case x.
2 subgoals
  x : nat
  ======================
   true = false -> 0 <> 0

Subgoal 2 is
   forall n : nat, negb(even n) -> S n <> 0
```

Néanmoins la variable `x` qui reste dans le contexte n'est plus connectée avec les valeurs qui la remplacent dans les deux conclusions de buts. Que cette variable deviennent déconnectée est parfois un problème. Montrons maintenant un exemple où un tel problème apparait. Reprenons la même situation initiale, mais utilisons `intros` avant d'appeler `case`.

```
  x : nat
  ======================
   match x with 0 => true | S p => negb (even p) end = false
   -> x <> 0
intros H.
  x : nat
  H : match x with 0 => true | S p => negb (even p) end = false
  ======================
   x <> 0
case x.
 x : nat
  H : match x with 0 => true | S p => negb (even p) end = false
  ======================
   0 <> 0

Subgoal 2 is
   forall n : nat, S n <> 0
```

Le premier but ne peut pas être prouvé, parce que l'occurrence de `x` qui apparait dans l'hypothèse `H` n'a pas été remplacée par 0.

Ce problème peut être résolu par une autre tactique appelée `case_eq`. Cette tactique ajoute dans les buts des égalités qui maintiennent la connection entre `x` et les valeurs qui la remplacent.

```
  x : nat
  H : match x with 0 => true | S p => negb (even p) end = false
 ======================
  x <> 0
case_eq x.
2 subgoals
  x : nat
  H : match x with 0 => true | S p => negb (even p) end = false
  ======================
   x = 0 -> 0 <> 0

Subgoal 2 is
   forall n : nat, x = S n -> S n <> 01
```

Cette fois les occurrences de `x` restent inchangées dans l'hypothèse `H`, mais nous avons maintenant une égalité dans la conclusion qui établit un lien entre `x` et la valeur qui le remplace. Cette égalité pourra être ajoutée dans le contexte à l'aide de la tactique `intros` puis utilisée à l'aide de la tactique `rewrite`.

2.2 the `induction` tactic

Pour les fonctions récursive, il n'est pas adapté de raisonner seulement par cas, parce que nous avons souvent besoin d'hypothèses sur le résultat des appels récursifs. Habituellement, ces hypothèses ont la même forme que l'énoncé que l'on veut prouver. Ceci suit un format déjà observé avec les preuves sur les nombres naturels, connu sous le nom de preuves par récurrence.

Récurrence sur les nombres naturels Si un prédicat sur les nombres naturels est tel que `P O` es satisfait et pour tout nombre `n` on peut déduire `P (S n)` de `P n`, alors ce prédicat est satisfait pour tous les nombres naturels.

induction on lists Si un prédicat sur les listes de nombres naturels `P` est tel que `P nil` est satisfait et pour toute liste `l` et pour tout nombre naturel `a`, si `P l` est satisfait on peut en déduire `P (a::l)`, alors ce prédicat est satisfait pour toutes les listes de nombres naturels.

Lorsque l'on fait une preuve par récurrence sur les nombres naturels, nous avons deux cas à étudier, le premier cas pour la situation où la valeur est `0` et le second pour la valeur a la form `S n` pour un un certain `n`. En ce sens, une preuve par récurrence est très proche d'une preuve par cas. Néanmoins, il y a une différence notable : dans le cas d'une preuve par récurrence, l'un des buts contient une hypothèse de récurrence qui exprime que `n` satisfait déjà la propriété attendu. Pour illustrer, observons une preuve par récurrence qui concerne l'addition d'un nombre avec 0. Etudions d'abord la définition de l'addition des nombres naturels dans le système Coq :

```
Locate "_ + _".
Notation            Scope
"x + y" := sum x y   : type_scope

"n + m" := plus n m  : nat_scope
                      (default interpretation)
Print plus.
fix plus (n m : nat) struct n : nat :=
  match n with
  | 0 => m
  | S p => S (plus p m)
  end
     : nat -> nat -> nat
```

Nous voyons que l'addition est donnée comme une fonction récursive où le premier argument décroit à chaque appel récursif. Par définition `0 + n` se calcul en `n` en une seule étape. En revanche, calculer `n + 0` ne donne rien directement, mais nous pouvons prouver que le résultat est aussi `n`. Voici la preuve en Coq :

```
Lemma example_induction_plus : forall n, n + 0 = n.
induction n.
2 subgoals

  ============================
   0 + 0 = 0

subgoal 2 is:
 S n + 0 = S n
```

Comme prévu, nous avons deux cas où `n` est remplacé soit par `0` soit par `S n`. Pour le premier cas, un calcul immédiat fournit le résultat.

```
reflexivity.
1 subgoal

  n : nat
  IHn : n + 0 = n
  ============================
   S n + 0 = S n
```

Dans le deuxième cas, le contexte du but contient l'hypothèse `IHn` qui exprime que la propriété que nous voulons prouver est déjà satisfaite pour `n`. Donc, le principe de récurrence est bien à l'œuvre ici, et le prédicat `P` de notre explication est instancié avec la fonction suivante :

```
fun x => x + 0 = x
```

2.3 La tactique `discriminate`

Les types de données des valeurs booléennes, des nombres naturels et des listes sont tous décrits dans le système Coq comme des types *inductifs*. Ces descriptions de type expriment à chaque

fois que les données peuvent correspondre à deux schémas. Nous pouvons le voir à l'aide de la commande `Print`.

```
Print bool.
Inductive bool : Set :=  true : bool | false : bool

Print nat.
Inductive nat : Set :=  O : nat | S : nat -> nat

Print list.
Inductive list (A : Type) : Type :=
    nil : list A | cons : A -> list A -> list A
```

La description des nombres naturels signifie que les nombres sont soit de la forme `O` soit de la forme `S n`. Néanmoins, cela signifie également que le nombre `O` n'est pas de la forme `S n`[1]. En conséquence, tout but dont la conclusion a la forme `O <> S n` devrait être prouvable facilement. Le système Coq fournit un tactique pour cela, appelée `discriminate`. Cette tactique traite également les buts avec une conclusion quelconque et une hypothèse de la forme `O = S n`.

En reprenant l'exemple de la section précédente, nous avons deux buts que nous répétons ici :

```
2 subgoals
  ======================
   true = false -> 0 <> 0

Subgoal 2 is
   negb(even n) -> S n <> 0
```

Pour le premier but, nou sutilisons la tactique `intros` et nous obtenons une nouvelle hypothèse :

```
intros Htf.
  ...
  Htf : true = false
  ======================
   0 <> 0
```

Dans ce but, la conclusion serait impossible à prouver dans un contexte vide, mais l'hypothèse `Htf` contient une égalité entre les deux constructeurs du type `bool`, qui sont différents par hypothèse. Ce but peut être prouvé par la tactique `discriminate` ou plus précisément la tactique `discriminate Htf`.

Le second but a la forme suivante :

```
  negb(even n) = true -> S n <> 0
```

Ici la partie droite est la négation d'une égalité entre deux cas du type inductif et la tactique `discriminate` resout ce but directement.

2.4 La tactique `injection`

Nous avons souvent besoin d'exprimer que les constructeurs sont des fonctions *injectives*. Par exemple, si deux listes sont égales, alors les premiers éléments respectifs de ces deux listes doivent être égaux. Ceci est illustré dans l'exemple suivant :

```
Lemma example_injection_list :
  forall (a b : nat) (l1 l2 : list nat),  a::l1 = b::l2 ->
    a = b /\ l1 = l2.
```

[1] C'est aussi une conséquence du fait que nous pouvons définir des expressions par filtrage sur les nombres naturels!

```
intros a b l1 l2 Hq.
  ...
  Hq : a::l1 = b::l2
  ========================
   a = b /\ l1 = l2
```

Dans ce but, l'hypothèse exprime que deux listes sont égales, la conclusion exprime que les deux premières composantes (les premiers éléments `a` et `b`) et les deux deuxièmes composantes (les queues de listes `l1` et `l2`) sont égales. La tactique qui permet d'aller de l'hypothèse à la conclusion est `injection Hq`.

```
injection Hq.
  ...
  ========================
   l1 = l2 -> a = b -> a = b /\ l1 = l2
intros ql qa; rewrite ql qa; split; reflexivity.
Qed.
```

Dans le nouveau but, deux implications sont créées, avec les égalités de composantes qui apparaissent en membre de gauche. Les deux dernières lignes de preuves montrent comment nous pouvons utiliser ces hypothèses.

De façon similaire, si nous savons que deux nombres qui respectent le schéma `S` sont égaux, alors leur sous-composants respectifs sont égaux.

```
Lemma example_injection_nat :
  forall (a b : nat),  Sa = S b -> a = b.
intros a b Hq.
  ...
  Hq : S a = S b
  ========================
   a = b
injection Hq.
  ...
  ========================
   a = b -> a = b
intros q1; exact q1.
Qed.
```

3 Controler l'exécution des fonctions

Dans les buts, le système Coq manipule les expressions sans les exécuter. Nous avons parfois besoin de forcer au moins quelques étapes de calcul. Cette section énumère quelques tactiques qui sont adaptées pour ce besoin.

3.1 La tactique `unfold`

La première approche est tout simplement de demander que le système remplace une fonction par sa définition. Le nom de la tactique est `unfold`.

```
Definition add3 (n : nat) := n + 3.

Lemma example_add3 : forall n, add3 n = 3 + n.
intros n.
  ...
  ====================
   add3 n = 3 + n.
```

Ici, nous voudrions remplacer `add3 n` par l'expression à laquelle cela correspond. Nous utilisons la tactique `unfold`.

```
unfold add3.
  ======================
    n + 3 = 3 + n
```

3.2 La tactique `simpl`

Quand on manipule une fonction récursive, la tactique `unfold` rend souvent des buts illisibles, parce qu'elle expanse la valeur des fonctions récursive vers quelque chose qui répète le texte de la définition récursive plusieurs fois. Pour l'éviter, il existe une tactique qui est étudiée spécialement pour traiter les fonctions récursive. Cette tactique s'appelle `simpl`.

L'exemple que nous avons déjà vu à propos du raisonnement sur la fonction d'addition fournit une illustration. Recommençons cette preuve.

```
Lemma example_induction_plus : forall n, n + 0 = n.
induction n.
reflexivity.
1 subgoal

  n : nat
  IHn : n + 0 = n
  ============================
   S n + 0 = S n
```

Ici, nous pouvons demander que Coq effectue un peu de calcul sur `S n + 0`. Il suffit d'appeler la tactique `simpl` :

```
simpl.
  ...
  IHn : n + 0 = n
  ============================
   S (n + 0) = S n
```

La partie gauche de l'égalité dans la conclusion de ce but est une occurrence du membre gauche de l'hypothèse `H`. Nous pouvons réécrire et conclure cette preuve.

```
rewrite IHn; reflexivity.
Qed.
```

3.3 Valeur explicite : la tactique `change`

Parfois, la tactique `simpl` effectue trop de calculs. Dans ce cas, c'est une bonne idée d'énoncer explicitement le résultat que nous voulons voir après le calcul, du moment que le résultat correspond bien à un calcul. Voici un exemple.

```
Lemma example_change_plus :
  forall n m p, (1 + n) * m = p -> (1 + (1 + n)) * m = m + p.
intros n m p H.
1 subgoal

 H : (1 + n) * m = p
  ============================
   (1 + (1 + n)) * m = m + p
change ((1 + (1 + n)) * m) with (m + (1 + n) * m).
  ...
```

```
  ============================
   m + (1 + n) * m = m + p
rewrite H; reflexivity.
Qed.
```

3.4 Valeur explicite : la tactique `replace`

La tactique `change` effectue un remplacement seulement si les deux expressions sont égales modulo le calcul. Parfois, nous voulons relâcher cette contrainte et effectuer un remplacement à condition que l'on puisse prouver l'égalité entre les deux expressions dans un but séparé. Pour cela, on utilise la tactique `replace`. Cette tactique produit donc un deuxième but.

4 Exercises

1. Définissez une fonction `lo` qui prend un entier naturel `n` en entrée et rtourne la liste contenant les `n` premiers nombres impairs. Par exemple `lo 3 = 5::3::1`.
2. Démontrez `length (lo n) = n`.
3. Définissez une fonction `sl` qui prend en entrée une liste de nombres naturels et qui retourne la somme des éléments de la liste.
4. Démontrez `sl (lo n) = n * n`.
5. Nous définissons une fonction `add` de la façon suivante :

   ```
   Fixpoint add x y := match x with 0 => y | S p => add p (S y) end.
   ```

   Démontrez les lemmes suivants :

   (a) `forall x y, add x (S y) = S (add x y)`
   (b) `forall x, add x 0 = x`
   (c) `forall x y, add (S x) y = S (add x y)`
   (d) `forall x y z, add x (add y z) = add (add x y) z`
   (e) `forall x y, add x y = x + y`