Verifying programs and proofs
Quatrième partie : preuves sur les programmes arithmétiques

Yves Bertot

September 2012

1 introduction

Avec les tactiques que nous avons apprises depuis le début nous pouvons prouver beaucoup de faits sur les expressions arithmétiques, mais nous devons le faire pas-à-pas, en combinant des théorèmes élémentaires. Dans ce chapitre, nous apprendrons comment utiliser des tactiques de preuve automatique qui traite de grosses formules en un seul coup.

Dans une première section, nous observerons plusieurs types de nombres qui sont disponibles dans le système Coq. Premièrement, le type des nombres naturels n'est pas adapté pour de grands calculs; deuxièmement, un type de données avec des nombres négatifs permet de traiter la soustraction de façon plus régulière. Nous pouvons aussi considérer un type de nombres réels, mais ce type n'est pas vraiment un type de données, au sens où de nombreux calculs sur ces nombres ne peuvent pas être décrits par des algorithmes.

2 Plusieurs types de nombres

Du point de vue de la complexité, le type des nombres naturels est très inefficace. La quantité de mémoire nécessaire pour stocker un nombre est proportionnel à la valeur du nombre. Ainsi, pour stocker le nombre 10 en mémoire, on utilise approximativement 20 mots de mémoire, pour stocker le nombre 20, on utilise approximativement 40 mots de mémoire et ainsi de suite.

Evidemment, il existe des façons plus évidentes de stocker les ombres. Par exemple, la représentation décimale que nous apprenons tous à l'école primaire utilise une quantité de mémoire qui est proportionnelle au logarithme du nombre représenté. Il faut $3 + 1$ chiffres pour écrire le nombre 1000 et $6 + 1$ pour écrire le carré, 1000000. Le système Coq fournit aussi un type de données qui à cette caractéristique de consommation mémoire : le type des entiers `Z`.

Le type des entiers est chargé dans Coq si nous exécutons la commande suivante (il faut bien noter l'usage exact des lettres capitales).

```
Require Import ZArith.
```

Ce type de données est décrit en deux étapes. La première étape décrit seulement les nombres positifs en forme binaire.

```
Print positive.
```

```
Inductive positive : Set :=
  xI : positive -> positive
| xO : positive -> positive
| xH : positive
```

Il y a trois cas dans ce type de données. Le cas `xH` représente le nombre 1, le cas `xI` représente la fonction $x \mapsto 2 \times x + 1$, et le cas `xO` représente la fonction $x \mapsto 2 \times x$. Par exemple, 1 est `xH`, 2 est `xO xH`, 3 est `xI xH`, 4 est `xO (xO xH)`, et 10 est `xO (xI (xO xH))`.

Pour construire des entiers, nous avons un second type de données qui choisit à nouveau entre trois cas, mais cette fois-ci pour exprimer que l'on est en présence de zéro, d'un nombre positif ou d'un nombre négatif.

```
Print Z.
Inductive Z : Set :=
  Z0 : Z | Zpos : positive -> Z | Zneg : positive -> Z
```

Ainsi le nombre 10 est effectivement représenté par `Zpos (xO (xI (xO xH)))`. Il faut s'habituer au fait que le même nombre est représenté différement suivant le type dans lequel il apparaît.

```
Check Zpos (xO (xI (xO xH))).
10%Z : Z

Check S (S (S (S (S (S (S (S (S (S 0))))))))).
10 : nat
```

Pour utiliser des nombres entiers par défaut, nous devons indiquer à l'analyseur syntaxique que nous travaillons dans un contexte d'analyse particulier (en anglais, ce genre de contexte est appelé un *scope*).

```
Open Scope Z_scope.
```

Si nous voulons retourné à l'usage des nombres naturels par défaut, nous pouvons annuler la directive précedente en tapant `Close Scope Z_scope`, mais cette n'est pas commande n'est pas utilisée dans ce qui suit.

```
Check Zpos (xO (xI (xO xH))).
10 : Z

Check S (S (S (S (S (S (S (S (S (S 0))))))))).
10%nat : nat
```

Donc le mécanisme des scopes indique à l'afficheur de convertir un nombre dans sa représentation dans un type ou un autre. Nous pouvons changer le scope localement en utilisant un signe pourcent, comme dans `%Z`.

Le méchanisme des scopes est aussi en charge de comprendre les opérations arithmétiques comme `+`, `*`, ou `-`.

La représentation des données pour les entiers relatifs est assez efficace pour que nous calculions des nombres très grands dans un temps raisonnable, une propriét qui n'est pas satisfaite par les nombres naturels.

```
Compute 1001 * 1001 * 1001 * 1001.
     = 1004006004001 : Z
```

3 Prouver des égalités numériques

Si vous exécutez la commande `Require Import Arith.`, le système Coq charge des théorèmes supplémentaires pour raisonner sur les nombres naturels. Si vous exécutez la commande `Require Import ZArith.`, il charge des théorèmes pour raisonner sur les nombres entiers.

Parmi les capacité de raisonnement disponibles, il y a la tactique `ring`. Pour les nombres naturels, elle traite les égalité entre formules contenant des constantes numériques, des additions et des multiplications. Voici un exemple :

```
Close Scope Z_scope.
Lemma ex1 :
  forall x y : nat, (3 * x + 2) * y = 2 * (x * y + y) + y * x.
```

```
intros x y.
  x : nat
  y : nat
  ======================
   (3 * x + 2) * y = 2 * (x * y + y) + y * x
ring.
No more subgoals.
```

L'égalité doit être auto-suffisante : la validité ne doit pas dépendre d'une égalité dans le contexte. Par exemple, le but suivant est prouvable, mais pas directement par la tactique `ring`.

```
  x : nat
  y : nat
  H : 2 * (y * y) = x * x
  ========================
   3 * x * x = 2 * (y * y) + 2 * x * x
ring.
Error: Tactic failure: not a valid ring equation.
```

Pour les entiers naturels, la tactique `ring` résout la même famille de problèmes, mais avec une opération supplémentaire : la soustraction.

```
Open Scope Z_scope.

Lemma ex2 : forall x y, 2 * (y * y) = x * x ->
  2 * (x - y) * (x - y) = (2 * y - x) * (2 * y - x).
intros x y H.
replace (2 * (x - y) * (x - y)) with
   (2 * (x * x) - 4 * (x * y) + 2 * (y * y)).
2 subgoals
  ...
  H : 2 * (y * y) = x * x
  ============================
   2 * (x * x) - 4 * (x * y) + 2 * (y * y) =
   (2 * y - x) * (2 * y - x)

subgoal 2 is:
 2 * (x * x) - 4 * (x * y) + 2 * (y * y) = 2 * (x - y) * (x - y)

replace ((2 * y - x) * (2 * y - x)) with
  (4 * (y * y)- 4 * (x * y) + x * x).
3 subgoals
  ...
  H : 2 * (y * y) = x * x
  ============================
   2 * (x * x) - 4 * (x * y) + 2 * (y * y) =
   4 * (y * y) - 4 * (x * y) + x * x

   ...
rewrite <- H.
3 subgoals
  ...
  ============================
   2 * (2 * (y * y)) - 4 * (x * y) + 2 * (y * y) =
   4 * (y * y) - 4 * (x * y) + 2 * (y * y)
```

```
  ...
```

Maintenant, les formules des deux cotés de l'égalité sont plutôt complexes, mais un calcul algébrique habituel permet de voir qu'elles sont égales. La tactique `ring` fait exactement cela. Les deux autres buts qui étaient engendrés par la tactique `replace` peuvent aussi être vérifiées par des calculs algébriques simples. De nouveau, la tactique `ring` suffit pour cette tâche. Donc cette preuve est effectuée complètement avec quelques usages de la tactique `ring`. Ainsi l'utilisation combinée de `replace` et `ring` permet de travailler rapidement sur des formules numériques, même des formules qui utilisent des variables.

4 Prouver des comparaisons

Les problèmes de comparaison sont des buts où les hypothèses et la conclusion sont de la forme `A < B`, `A = B`, `A <= B`, et les formules `A`, `B` sont des formules numériques reposant sur des additions, des multiplications, et des soustraction. Si les formules contiennent des des multiplications arbitraires et des nombres sont des nombres naturels ou des entiers, on sait que les problèmes de comparaison sont indécidables. En d'autres termes, il est impossible de construire une tactique qui résoudra tous les problèmes de comparaisons vrais.

Mais si `A` et `B` ne contiennent que des additions et des soustractions, alors les problèmes de comparaison deviennent décidables. C'est équivalent à accepter des multiplications par des constantes numériques. Il y a une tactique qui effectue ce genre de preuves pour les nombres naturels et les entiers, elle est appelé `omega`. Les problèmes acceptés par cette tactique sont appelés des problèmes d'arithmétique linéaire entière.

Cette tactique est incluse dans la bibliothèque `ZArith`, mais pas dans la bibliothèque `Arith`. Pour la charger, vous pouvez avoir à la charger explicitement, comme dans l'exemple suivant :

```
Require Import Omega.

Lemma omega_ex : forall x y, x > y -> 2 * y > 3 * x -> False.
1 subgoal

  ============================
   forall x y : nat, x > y -> 2 * y > 3 * x -> False

intros; omega.
No more subgoals.
Qed.
```

5 Advanced proof by induction for numbers

Pour les preuves par récurrence, il est important de s'assurer que l'hypothèse de récurrence est assez forte pour la preuve que nous voulons faire. La technique la plus simple pour rendre l'hypothèse de recurrence plus forte est de démontrer par récurrence un énoncé plus fort. Une autre approche est d'utiliser des théorèmes génériques qui fournissent systématiquement des hypothèses de récurrences plus faibles. Les deux section suivantes illustrent cette approche.

5.1 Renforcer la récurrence

Quand on fait des preuves par récurrence, il peut être plus facile de démontrer des énoncés plus forts. Cela peut sembler paradoxal, parce que des énoncés plus forts sont plus durs à démontrer. Néanmoins, dans les preuves par récurrence l'énoncé à démontrer par récurrence est répété dans l'hypothèse de récurrence qui devient également plus forte.

Voici un exemple qui illustre cette idée. Les premières lignes de la preuve sont utilisées pour montrer qu'une récurrence simple échoue.

```
Require Import Arith Omega.

Fixpoint mod2 (n : nat) :=
  match n with S (S p) => mod2 p | a => a end.

Lemma mod2_div2_eq : forall n, n = mod2 n + 2 * div2 n.
Proof.
induction; simpl; [omega | ]
  n : nat
  IHn : mod2 n < 2
  ============================
   match n with
   | 0 => S n
   | S p => mod2 p
   end < 2
destruct n;[ omega | ].

  n : nat
  IHn : mod2 (S n) < 2
  ============================
   mod2 n < 2
```

Il peut sembler naturel d'invoquer `destruct` parce que la conclusion du but contient une construction de filtrage. Mais le but que nous obtenons par la suite n'est pas facile à prouver. Il y a une trop grande différence entre l'énoncé qui concerne `n` dans la conclusion et `S n` dans l'hypothèse. Nous allons interrompre cet essai et repartir du début avec un énoncé plus fort.

L'énoncé que nous choisissons de prouver est non seulement que `mod2 n` est plus petit que 2, mais aussi que `mod2 (S n)` est plus petit que 2. Nous utilisons la tactique `assert` pour introduire cet énoncé plus fort.

```
Lemma mod2_lt : forall n, mod2 n < 2.
intros n; assert (H : mod2 n < 2 /\ mod2 (S n) < 2);
 [ | destruct H; assumption].
  n : nat
  ============================
   mod2 n < 2 /\ mod2 (S n) < 2
induction n;[simpl; omega | ].
  n : nat
  IHn : mod2 n < 2 /\ mod2 (S n) < 2
  ============================
   mod2 (S n) < 2 /\ mod2 (S (S n)) < 2
```

Le nouveau but est plus complexe : nous devons maintenant prouver une conjonction plutôt qu'une égalité. Mais l'hypothèse de récurrence est aussi une conjonction et le membre droit de l'hypothèse coincide avec le membre gauche de la conclusion. Ainsi, nous pouvons décomposer l'hypothèse et résoudre la première partie de la preuve rapidement.

```
destruct IHn as [H1 H2]; split;[assumption | ].
  n : nat
  H1 : mod2 n < 2
  H2 : mod2 (S n) < 2
  ============================
   mod2 (S (S n)) < 2
```

Par calcul, `mod2 (S (S n))` est la même chose que `mod2 n`. Ainsi, ce but est facilement résolu par la tactique `assumption`.

5.2 Récurrence forte

L'exemple donné dans la section précédente montre que nous pouvons modifier l'hypothèse de récurrence pour qu'elle mentionne non seulement le prédécesseur du nombre mentionné dans la conclusion, mais aussi le prédécesseur du prédécesseur. Parfois, nous pouvons avoir besoin d'une hypothèse de récurrence qui soit valide pour tous les nombres plus petits que celui qui apparait dans la conclusion.

Il existe un théorème qui couvre exactement ce genre de besoin, mais il s'applique dans un contexte plus général. Le nom de ce théorème est `well_founded_ind` et il s'applique pour une grande quantité de relation binaires, dès que ces relations sont "bien fondées". La relation `lt` (l'ordre strict entre deux nombres naturels) satisfait cette property. Ainsi, nous pouvons combiner deux théorèmes ensemble pour obtenir un principe de récurrence plus fort que le principe de base sur les nombres naturels.

```
Check well_founded_ind lt_wf.
well_founded_ind lt_wf
 : forall P : nat -> Prop,
   (forall x : nat, (forall y : nat, y < x -> P y) -> P x) ->
   forall a : nat, P a
```

Lisons attentivement l'énoncé de ce théorème. Il contient une quantification universelle sur un predicat `P`. Il n'y a qu'un cas à couvrir (au lieu de deux pour le principe de récurrence de base), mais ce cas fournit une hypothèse de récurrence pour tout nombre plus petit que celui pour leque le prédicat doit être prouvé. Si nous arrivons à prouver ce cas unique, le prédicat est vrai universellement sur tout le type des entiers naturels.

Pour le nombre 0, il n'y a pas de nombre naturel plus peit, donc il n'y a pas de nombre pour lequel l'hypothèse de récurrence est satisfaite. Donc c'est similaire au cas de base du principe de récurrence conventionnel.

C'est illustré dans un exemple où l'on raisonne sur un algorithme de division sur les nombres naturels. Cette fonction retourne un couple contenant le quotient et le reste de la division. Il faut noter que cette fonction est réglée pour traiter les divisions par 0 avec indulgence. Quand on divise par 0, le quotient est mis à 0 et le reste est le nombre que l'on est en train de diviser.

```
Fixpoint div (n m : nat) :=
 match m, n with
   S m', S n' =>
   if leb m n then
      let (q, r) := div (n' - m') m in (S q, r)
    else
     (0, n)
 | _, _ => (0, n)
 end.
```

Quand cette fonction calcul, elle a des appels récursifs, mais le second argument ne décroît jamais (la valeur initiale est `m` et la valeur du second argument dans l'appel récursif est aussi `m`). Le premier argument décroit d'une quantité variable, mais il décroit bien strictement, parce que `n'` est strictement plus petit que `n` et que l'on enlève `m'` de `n'`. Par exemple, si nous calculons `div 7 1`, les appels récursifs sont `div 6 1`, `div 5 1`, et cetera. Si `m| est 3, alors m' est 2, et si l'on calcule div 7 3, alors les appels récursifs sont div 4 3 et div 1 3.`

`Nous pouvons prouver une égalité simple à propos de cette fonction de division :`

```
Lemma div_eq :
   forall n m, n = m * fst (div n m)  + snd (div n m).
```

```
intros n; induction n  as [n IHn] using (well_founded_ind lt_wf).
  n : nat
  IHn : forall y : nat,
        y < n -> forall m : nat, y = m * fst (div y m) + snd (div y m)
  ============================
   forall m : nat, n = m * fst (div n m) + snd (div n m)
```

Comme nous le voyons, l'hypothèse de récurrence exprime que n'importe quel y plus petit que n satisfait la propriété que nous voulons prouver pour n.

Le reste de cette preuve est laissé comme exercice. Un autre exercice consiste à prouver que le deuxième argument du couple retourné par la division est inférieur au diviseur lorsque ce diviseur est non nul.

6 Techniques avancées de preuves sur les listes

Pour les listes, la technique de renforcement de l'hypothèse de récurrence s'applique comme précédemment. Pour l'illustrer, observons une preuve qui concerne les listes retournées.

Il y a deux façons de retourner des listes, mais seulement une est efficace en temps.

```
Require Import List.

Fixpoint slow_rev_nat (l : list nat) : list nat :=
  match l with
    nil => nil
  | a::l' => slow_rev_nat l' ++ (a::nil)
  end.

Fixpoint pre_rev_nat (l l' : list nat) : list nat :=
  match l with
    nil => l'
  | a :: l' => pre_rev_nat l (a::l')
  end.

Definition rev_nat (l : list nat) : list nat :=
   pre_rev_nat l nil.
```

Dans la définition de slow_rev_nat, nous utilisons une fonction app, avec une notation infixe ++, qui concatène deux listes. Il est facile de se convaincre que slow_rev_nat retourne une liste : lorsque l'on travail avec une liste dont le premier élément est a, cet élément se retrouve à la fin du résultat et le reste de l'entrée se trouve devant, après avoir été retourné. Nous pouvons utiliser cette algorithme pour référence, mais il est lent à l'exécution parce la concaténation prend un temps proportionnel à la longueur du premier argument. En fin de compte, la complexité de la fonction slow_rev_nat est quadratique.

L'autre fonction est plus efficace : le retournement se fait en temps linéaire. Nous pouvons le vérifier en testant ces fonctions sur des listes de taille croissante. Donc, il est utile d'avoir les deux fonction; la première peut être utilisée pour les spécification, la deuxième pour les implémentations.

Si nous voulons exprimer la correction de la fonction efficace, nous pouvons simplement écrire l'énoncé suivant :

```
Lemma rev_nat_correct : forall l, rev_nat l = slow_rev_nat l.
```

Si nous commençons directement cette preuve par une récurrence, nous découvrons que l'énoncé perd sa belle forme et la preuve se bloque dans le cas de récurrence.

```
induction l as [ | a l' IHl'].

  ======================
   rev_nat nil = slow_rev_nat nil
reflexivity.
simpl.

  IHl' : rev_nat l' = slow_rev_nat l'
  ======================
   rev_nat (a::l') = slow_rev_nat l'::(a::nil)
```

Dans cet état, il n'y a pas de moyen simple pour montrer une correspondance entre rev_nat a::l' et rev_nat l' principalement parce que le deuxième terme n'est pas un sous-terme du premier. L'essai de preuve est bloqué.

Pour réparer cette preuve, nous pouvons utiliser un énoncé plus fort qui concerne prev_rev_nat et prouve une formule quantifiée universellement par rapport aux deux arguments de cette fonction.

```
Lemma rev_nat_correct : forall l, rev_nat l = slow_rev_nat l.
assert (forall l l', pre_rev_nat l l' = slow_rev_nat l ++ l').
 induction l.
  intros; reflexivity.
 intros l'; simpl.
  rewrite <- app_assoc.
 simpl; apply IHl.
intros l; unfold rev_nat; rewrite H, <- app_nil_end; reflexivity.
Qed.
```