Vérification de programmes et de preuves Deuxième partie. décrire des propriétés de programmes

Yves Bertot

September 2012

1 Introduction et motivation

Le système Coq ne fournit pas d'outil de débug. D'un autre coté, il fournit des moyens d'exprimer des relations logiques entre plusieurs données. Par exemple, nous pouvons exprimer logiquement la relation qui devrait exister entre les entrées et les sorties d'une fonction, d'une manière qui est indépendante de la façon dont l'algorithme est décrit dans la fonction. Si nous pensons à l'algorithme de division (tel que nous l'apprenons à l'école), il y a une relation simple entre le dividende `x`, le diviseur `y`, le quotient obtenu `q`, et le reste de la division `r`:

```
x = y * q + r /\ r < y
```

En anglais, cette relation exprime que `x` est égale à la somme du produit de `y` et `q` et de `r`, et que `r` est plus petit que `y`. Cette relation devrait être satisfaite par les résultats, mais ceci n'est possible que si nous évitons de diviser par 0. Dans ce cours, nous concentronrs sur le langage utilisé pour exprimer les relations entre données.

2 Ecrire des formules logiques

2.1 Les prédicats

Dans cette section nous considérons un nouveau type, le type des propositions, and les moyens de construire des objets atomic dans ce type. Le moyen le plus simple d'affirmer une propriété est de dire que deux valeurs sont égales. Voici deux exemples:

```
Check 3 = 2 + 1.
3 = 2 + 1 : Prop

Check 3 = 4.
3 = 4 : Prop
```

Remarquons bien qu'il est toujours possible d'écrire une proposition logique, du moment qu'elle est bien formée, même si cette proposition est fausse.

Dans ce cours, nous définirons de nouveaux prédicats simplement en décrivant de nouvelles fonctions et en montrant que la valeur retournée peut être comparée à une autre valeur. Par exemple, nous pouvons écrire une fonction qui retourne la valeur booléenne `true` exactement quand son argument est pair (*even* anglais) et ensuite transformer cela en un prédicat en exprimant que la valeur retournée par cette fonction est `true`.

```
Fixpoint even (n : nat) :=
  match n with 0 => true | 1 => false | S (S p) => even p end.

Check even 3.
even 3 : bool

Check even 3 = true.
even 3 = true : Prop
```

Nous voyons ici qu'il y a une différence entre les valeurs bouléennes et les propositions. En pratique, les valeurs booléennes sont prévues pour servir de résultat à des tests qui peuvent être testés dans des programmes. Les propositions logiques ne sont pas prévues pour apparaître dans les programmes, au moins pas dans ce cours.

2.2 Connecteurs logiques

Il y a quatre connecteurs logiques simples : *implies*, *and*, *or*, *not*. Ces connecteurs logiques sont donnés avec des notations qui permettent d'écrire des formules logiques de manière concise et lisible.

- Implication is noted `->`.
- Conjunction (*and*) is noted `/\`.
- Disjunction (*or*) is noted `\/`.
- Negation is noted `~`.
- Equivalence is noted `<->`. In fact `A <-> B` is a defined as `A -> B /\ B -> A`.

Par exemple, nous pouvons écrire la proposition logique suivante.

```
Check ~ (even 2) = true -> even 3 = false.
```

2.3 Quantifications

Nous pouvons écrire des formules qui donnent des propriétés sur un type entier. Il y a deux connecteurs principaux pour cela. Le premier, `forall`, décrit

la *quantification universelle*. En d'autres termes, c'est utilisé pour exprimer que tous les éléments du type satisfont une certaine propriété. Le deuxième connecteur, `exists`, exprime la *quantification universelle*. C'est utilisé pour exprimer qu'au moins un élément du type satisfait une certaine propriété. Par exemple, nous pouvonex écrire la proposition suivante.

```
Check forall x, even x = true -> exists y, x = 2 * y
```

Ici, l'exemple choisi pour illustration peut être prouvé. En un sens, ceci veut dire que la fonction `even` calcule exactement ce que nous avons en tête.

2.4 Quelques autres prédicats prédéfinis

Quand les bibliothèques `Arith` et `List` sont chargées, le système Coq fournit auss un prédicat prédéfini pour comparer deux nombres naturels, noté `x <= y`. Plusieurs variantes sont fournis pour des comparaisons strictes (`x < z`), and pour des intervalles (`x <= y < z`).

Pour les listes, il y a aussi le prédicat `In`: `In x l` signifie que `x` apparait dans la liste `l`.

2.5 Playing with logical formulas

Pour faire des progrès vous devriez être capable d'écrire des formules logiques simples qui expriment votre connaissance de la vie réelle, de certains programmes, ou de certains fait mathématiques.

En général, lorsque l'on écrit `A -> B`, pour que cette formule soit vraie, il n'est pas nécessaire que `A` soit vrai. En revanche, si l'on écrit `A /\ B`, pour que cette formule il est au moins nécessaire que `A` soit vrai.

Voici une phrase de la vie de tous les jours: *en hiver, les jours sont toujours plus courts que les nuits, mais certains jours peuvent être ensoleillés*. Nous pourrions exprimer le sens de cette phrase en supposant que nous avons un type pour les dates, un prédicat `winter` qui est satisfait pour toutes les dates en hiver, une fonction `night_length` qui retourne la longueur des nuits sous forme d'un nombre naturel pour chaque date et une fonction similaire `day_length` qui retourne la longueur du jour. Aussi, nous pourrions utiliser une fonction `weather` qui prend une date et une année et retourne le temps pour ce jour-là. Cette fonction devrait retourner ses valeurs dans un type qui contient au moins les valeurs `sunny` et peut-être `rainy`, ou `cloudy`. La phrase pourrait donc être décrit par la formule logique suivante:

```
(forall d: date, winter d -> day_length d <= night_length) /\
(exists y : nat, exists d : date, weather y d = sunny)
```

La distinction entre les conjonctions et les implications est importante. Si vous définissez un prédicat `P` et vous savez qu'un prédicat `A` doit être satisfait pour tous les objets qui satisfontt `P`, alors la définition de `P` devrait certaine contenir une conjonction qui contient une instance de `A`, pas une implication.

Il peut utiliser des objets qui ne satisfont pas `A` (et alors ils ne satisfont pas `P` non plus). Voici un exemple avec les nombres premiers: nous savons que toues les nombres premiers sont supérieurs ou égaul à 2, donc lorsque nous décrivons une définition du prédicat `prime` (l'adjectif anglais pour les nombres premiers) nous écrivons quelque chose comme ça.

```
Definition prime n := 2 <= n /\ ...
```

Si nous remplacions cette conjonction par une implication, cela voudrait dire que 0 satisfait le prédicat `prime`, parce qu'une formule de la forme `2 <= 0 -> ...` peut toujours être prouvée.

3 Performing simple proofs

Pour effectuer une preuve, nous énonçons la proposition que nous voulons prouver, nous décomposons la proposition en des propositions plus simples, jusqu'à ce qu'elle puissent être résolues. Les commandes utilisées pour décomposer les propositions sont appelées des *tactiques*. Pour apprendre comment utiliser des tactiques il est pratique de les ranger en fonction du connecteur sur lequel elles travailles et en fonction de la position où ce connecteur apparaît: dans un fait connu ou dans un fait que nous devons prouver.

3.1 Stating a logical formula

C'est mieux de montrer ça sur un exemple.

```
Lemma ex1 : 2 = 3 -> 3 = 2.
```

Le mot-clef est `Lemma`: nous utiliserons cette commande chaque fois que nous voulons démarrer une nouvelle preuve. Ensuite nous incluons un nom qui doit être unique (ici c'est `ex1`). Ensuite il y a un caractère deux points "`:`" Ensuite il y a une formule logique (ici, une implication entre deux égalitee). La commmande se termine par un point.

3.2 Known facts and facts to prove

A n'importe quel momment pendant la preuve, le système affiche des *buts*, qui décrivent ce que nous devons prouver. Chaque but a deux parties, la première partie est un *contexte* qui contients des faits connus (que nous appellerons aussi des *hypothèses*), la deuxième partie est une *conclusion*, un fait que nous devons prouver. Un cas simple est celui où la conclusion se trouve parmi les faits connus. Dans ce cas, il suffit d'utiliser la tactique élémentaire `assumption` pour résoudre le but. Si nous le faisons, le système affiche le but suivant ou indique que la preuve est finie.

3.3 Pour terminer une preuve

Quand il n'y a plus de buts à résoudre, le système saique la preuve est complète, mais il reste une opération que nous devons faire: nous devons expliquer au système que la preuve complète doit être enregistrée en mémoire. C'est fait à l'aide de la commande `Qed`.

3.4 Manipuler les connecteurs logiques

3.4.1 implication

Quand la conclusion d'un but est une implication, nous pouvons rendre ce but plus simple en utilisant la tactique `intros`. Ceci produit un nouveau but avec un élément supplémentaire dans le contexte. Le nom donné à cet élément est un argument que nous donnons à la tactique `intros`. Ce nouvel élément est la formule qui apparaissait à gauche dans l'implication en conclusion du but, la nouvelle conclusion est la formule qui apparaissait à droite.

```
  H : A
  ==================
  2 = 3 -> 3 = 2

intros H'.
  H : A
  H' : 2 = 3
  ==================
  3 = 2
```

Pour utiliser une hypothèse qui contient une implication, nous utilisons la tactique `apply`. Voici un exemple:

```
  H : A -> 2 = f 3
  ==================
  2 = f 3

apply H.
  H : A -> 2 = f 3
 ==================
  A
```

Ceci ne marche que si la partie droite de l'hypothèse `H` correspond exactement avec la conclusion. La conclusion est alors remplacée par la partie gauche de l'implication. S'il y a plusieurs flèches imbriquées dans l'hypothèse, plusieurs nouveaux buts sont produits. Si l'hypothèse était `A -> B -> 2 = 3`, nous obtiendrions deux buts, un avec `A` comme conclusion et un avec `B`.

3.4.2 Conjunction

Quand la conclusion d'un but est une conjunction nous pouvons simplifier ce but en appliquant la tactique `split`. Ceci produit deux nouveaux buts dont les énoncés sont les parties de la conjonction.

```
  H1 : A -> B
  ======================
  C /\ D
split.
  H1 : A -> B
  ======================
   C

Subgoal 2 is:
   D
```

Quand nous voulons utiliser une hypothésis qui est une conjonction, nous avons souvent besoin de décomposer cette conjonction pour en obtenir les parties. C'est la tactique `destruct` que l'on utilise pour ça. Notez que cette tactique permet de donner le nom des nouvelles hypothèses qui seront ajoutées pour exprimer explicitement que les parties de la conjonction sont des faits connus.

```
  H : A /\ B
  ====================
   A
destruct H as [H1 H2].
  H1 : A
  H2 : B
  ====================
   A
```

3.4.3 Disjunction

Quand la conclusion d'un but est une disjonction, nous pouvons le rendre plus simple en choisissant celle des parties que nous voulons prouver (nous avons le choix, mais nous ne devons pas nous tromper). Pour exprimer notre choix, nous utilisons une tactique `left` ou `right`. Ceci produit un nouveau but qui contient seulement la partie choisie.

```
  H1 : A -> B
  ======================
  B \/ 2 = 3
left.
  H1 : A -> B
  ======================
   B
```

Evidemment, nous devons faire attention de choisir la tactique qui mène réellement à un nouveau but prouvable (dans cet exemple, choisir `right` semble idiot).

Quand nous voulons utiliser une hypothèse qui est une disjonction, nous avons souvent besoin de décomposer cette disjonction pour extraire ses parties comme de nouvelles hypothèses. Il faudt couvrir deux cas: le cas lorsque c'est la partie droite qui est satisfaite et le cas lorsque c'est la partie gauche. Ceci correspond à deux buts.

```
  H : A \/ B
  ======================
   B \/ A
destruct H as [H1 | H2].
  H1 : A
 ======================
   B \/ A

Subgoal 2 is:
   B \/ A
```

Le second but contient une hypothèse nommée `H2` (comme indiqué dans la tactique `destruct`) avec `B` comme énoncé.

3.4.4 Negation

Quand la conclusion d'un but est une négation, nous pouvons le rendre plus simple en applicant la tactique `intros`

```
  H1 : A -> B
  ========================
   ~ A
intros H.
  H1 : A -> B
  H : A
  ========================
   False
```

Puisque nous voulons montrer "not A", l'idée est de montrer que si `A` était vrai, alors nous pourrions prouver une formule qui est normalement impossible.

Si nous voulons utiliser une hypothèse qui est une négation, nous appliquons la tactique `case H`. Ceci remplace la conclusion du but par la formule qui était niée.

```
  H : ~ A
  ======================
   C
case H.
  H : ~ A
 ======================
   A
```

Nous devrions effectuer cette étape de raisonnement exactement quand nous savons que nous pouvons prouver `A`.

L'un des éléments de base de la logique mathématique est que si nous avons une contradiction dans le contexte, nous pouvons en déduire n'importe quoi. La façon la plus pure d'avoir une contradiction est d'avoir une hypothèse dont l'énoncé est `False`. Si ceci vous arrive, alors ce but peut être résolu en appelant la tactique `case H`.

Un autre exemple fréquent de contradiction est lorsque l'on a une hypothèse `H1` qui affirme `~A` et une autre hypothèse `H2` qui affirme `A`. Dans ce cas, on sait que l'on pourra prouver `A`, donc on peut commencer par utiliser l'hypothèse qui est niée, à l'aide de la tactique `case H1`, puis on peut finir la preuve en utilisant la tactique `assumption`.

3.5 Quantifiers

3.5.1 Universal quantification

Quand on veut prouver une formule quantifiée universellement, on utilise souvent la tactique `intros x`, où `x` est le nom choisi pour une variable fixée sur laquelle on va effectuer le raisonnement. C'est une variable représentant une valeur arbitraire, donc si l'on arrive à prouver la propriété pour cette variable, on exprime bien que la propriété est prouvable pour n'importe quelle valeur dans le type considéré. Voici un exemple:

```
  ==================
  forall A:Prop, A -> A
intros A.
  A : Prop
  ==================
   A -> A
```

Cette preuve peut se terminer en utilisant les tactiques `intros H.` puis `assumption.`

Dans une autre leçon, nous verrons que certaines formues quantifiées universellement peuvent être prouvées par d'autres moyens plus élaborés, comme la preuve par récurrence fournie par la tactique `induction`.

Si l'on veut utiliser une hypothèse qui démarre avec une quantification universelle, on devrait utiliser la tactique `apply`. Voici un exemple:

```
  H : forall x: nat, P x -> Q x
  ================
   Q 3
apply H.
  H : forall x: nat, P x -> Q x
  ================
   P 3
```

Notez bien que le système Coq a trouvé une instance de la formule quantifiée universellement qui correspond à `Q 3`, ensuite il applique le même comportement que pour l'implication.

Parfois il est nécessaire d'exprimer explicitement que l'on veut une instance particulière d'une hypothèse quantifiée universellement. Le truc est d'employer l'hypothèse comme si c'était une fonction et de l'appliquer à la valeur sur laquelle on veut l'instancier. Ceci se fait avec une autre tactique que l'on appelle `assert`. Par exemple, voici une autre façon d'utiliser l'hypothèse `H` dans l'exemple précédent:

```
  H : forall x: nat, P x -> Q x
  ================
   Q 3
assert (H1 := H 3).
  H : forall x: nat, P x -> Q x
  H1 : P 3 -> Q 3
  ================
   Q 3
apply H1.
  H : forall x: nat, P x -> Q x
  H1 : P 3 -> Q 3
  ================
   P 3
```

3.5.2 Existential quantification

Lorsque l'on veut prouver une formule quantifiée existentiellement, on doit fournir une valeur candidate et montrer que cette valeur satisfait bien la propriété. La tactique s'appelle `exists` (avec la même orthographe que le connecteur logique). Voici un exemple:

```
  ======================
   exists x, 2 * x = 6
exists 3.
  ======================
   2 * 3 = 6
```

Ensuite, nous n'avons plus qu'à prouver que la valeur fournie satisfait la proposition attendue.

Quand on veut utiliser une hypothèse qui commence par une quantification existentielle, nous devons en fait décomposer cette hypothèse pour obtenir un contexte dans lequel il existe une nouvelle valeur arbitraire et l'hypothèse que cette valeur satisfait la propriété. Voici un exemple:

```
  H : exists x : nat, even x = true
   ======================
    C
destruct H as [w Pw].
  w : nat
  even w = true
  ======================
```

```
  C
```

Dans le but avant l'exécution de la tactique `destruct` il n'y a pas de nombre naturel dans le context, et encore moins de nombre naturel qui satisfait la propriété. Après la tactique, une nouvelle variable `w` est ajoutée, ainsi que l'hypothèse que `even w = true`. Nous pouvons maitenant utiliser cette variable comme un nombre naturel pour des besoins variés.

3.6 Les prédicats usuels

3.6.1 L'égalité

Quand nous voulons prouver une égalité, le plus simple est lorsque les deux membres de l'égalité sont égaux (modulo calcul). On utilise alors la tactique `reflexivity`, comme dans l'exemple suivant:

```
  ======================
  2 = 3 - 1
reflexivity.
  No more subgoals.
```

Cette tactique peut aussi être utilisée si le calcul utilise des fonction que nous avons définies nous-mêmes, comme `even`.

Si nous voulons utiliser une hypothèse qui contient une égalité, nous pouvons utiliser la tactique `rewrite`.

```
  H : 3 = 2
  ====================
   2 = 3
rewrite H.
  H : 3 = 2
  ====================
   2 = 2
```

La tactique `rewrite` peut aussi s'utiliser si l'égalité est enveloppée dans une quantification universelle. Dans ce cas, elle trouve l'instantiation adaptée avant d'effectuer la réécriture.

```
  H : forall x : nat, f (f x) = g (x + x)
  ====================
   g (2 + 2) = E
rewrite <- H.
  H : forall x : nat, f (f x) = g (x + x)
  ====================
   f (f 2) = E
```

Cet exemple illustre aussi le fait que nous pouvons utiliser un modificateur `<-` pour indiquer que la réécriture doit remplacer une instance du membre droit par l'instance correspondante du membre gauche.

4 Exercices

1. Ecrire un prédicat `multiple` de type `nat -> nat -> Prop`, tel que `multiple a b` exprime que `a` est un multiple de `b` (en d'autres termes, il existe un nombre `k` tel que `a = k * b`).

2. En supposant qu'il existe un type `T` utilisé pour représenter les nombres et une fonction `nat_of_T` de type `T -> nat`, qui associe chaque élément de `T` à l'élément de `nat` qu'il représente, et en supposant que `tadd` est une fonction de type `T -> T -> T`, comment exprime-t-on que `tadd` représente l'addition? Faites attention, plusieurs élément de `T` peuvent représenter le même élément de `nat`.

3. Ecrivez le script qui prouve la formule suivante:

```
forall P Q : nat -> Prop,
forall x y:nat, (forall z, P z -> Q z) -> x = y -> P x ->
  P x /\ Q y
```

4. Ecrivez le script qui prouve la formule suivante:

```
forall A B C : Prop, (A /\ B) \/ C -> A \/ C
```

5. Ecrivez le script qui prouve la formule suivante:

```
forall P : nat -> Prop, (forall x, P x) ->
 exists y:nat, P y /\ y = 0
```

5 Plus d'information

Vous pouvez utiliser le livre [1] (disponible en français sur internet) et le manuel de référence [4]. Il existe aussi un tutoriel en français par un autre professeur [7]. Il y aussi des tutoriels en anglais [5, 3].

References

[1] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[2] B. Pierce et al. *Software Foundations* `http://www.cis.upenn.edu/~bcpierce/sf/`

[3] Y. Bertot *Coq in a Hurry* Archive ouverte "cours en ligne", 2008. `http://cel.archives-ouvertes.fr/inria-00001173`

[4] The Coq development team. *The Coq proof Assistant Reference Manual*, Ecole Polytechnique, INRIA, Université de Paris-Sud, 2004. `http://coq.inria.fr/doc/main.html`

[5] G. Huet, G. Kahn, C. Paulin-Mohring, *The Coq proof Assistant, A Tutorial*, Ecole Polytechnique, INRIA, Université de Paris-Sud, 2004. `http://coq.inria.fr/V8.1/tutorial.html`

[6] E. Giménez, P. Castéran, *A Tutorial on Recursive Types in Coq*, INRIA, Université de Bordeaux, 2006. `http://www.labri.fr/Perso/~casteran/RecTutorial.pdf.gz`

[7] A. Miquel, *Petit guide de survie en Coq*, Université de Paris VII. `http://www.pps.jussieu.fr/~miquel/enseignement/mpri/guide.html`