Vérification de programmes et de preuves
Première partie. décrire des algorithmes

Yves Bertot

September 2012

1 Motivating introduction

A partir des années 1940, il a été compris que l'on pouvait utiliser les ordinateurs pour faire des mathématiques. On peut penser que l'objectif principal est de calcul, et bien sûr les ordinateurs sont très utiles pour calculer. Mais les mathématiques c'est plus que du calcul, il y a aussi du raisonnement.

Le raisonnement respect des règles, les règles de la logique. Si vous voulez convaincre quelqu'un, vous devez suivre ces règles, sinon votre interlocuteur peut toujours prétendre qu'il n'a pas confiance en vos arguments. Les règles de la logique doivent être plutôt très simples, voire simplistes, de telle sorte que votre interlocuteur ne peut pas les rejeter, même s'il est très têtu. Les règles que nous utilisons maintenant sont si simple qu'un ordinateur peut vérifier si elles ont été appliquées correctement.

Si Alice veut vérifier que Paul fait bien son travail, c'est très difficle si ce que Paul doit faire est long et ennuyeux, et si la vérification repose sur des règles trop simples. Mais un ordinateur est utile dans ce cas, dès qu'il peut reconnaître les règles à appliquer, parce qu'un ordinateur ne fatigue jamais.

C'est ça le but du jeu: we tâchons de décrire les mathématiques jusqu'au niveau le plus élémentaire pour qu'un ordinateur puisse le vérifier et ensuite on peut s'en remettre à l'ordinateur pour la vérification.

La programmation et le génie logiciel peuvent également bénéficier de cette approche. Pour s'assurer qu'un programme fait ce qui est prévu, nous avons besoin de vérifier tous les cas. Nous pouvons le faire par des campagnes de test massives, mais les tests ne peuvent attraper les erreurs qu'au hasard. En revanche, nous pouvons raisonner sur ce que les programmes font et vérifier qu'ils se comportent bien dans les cas possibles.

La seule limite de cette approche est que nous utilisons un modèle simplifié de ce que les programmes font et de ce qu'il doivent faire. La plupart du temps, le modèle est différent de la vie réelle. Donc, même si nous prouvons que le programme fait ce que nous voulons, il reste des possibilité pour des erreurs: les erreurs qui ne peuvent pas être décrites par le modèle. Malgré cela, il y a toujours un bénéfice. Nous augmentons la compréhension du programme dès que nous faisons l'effort d'en donner une description plus abstraite.

L'industrie exprime un intérêt grandissant pour cette approche, surtout pour les entreprises qui sont très vulnérables aux fautes de conception logicielles et matérielles. Par exemple, Intel a perdu des millions de dollars parce que l'algorithme de division qui était imprimé dans les Pentium (vers l'année 1998) était erroné. Intel emploie maintenant des spécialiste en démonstration sur ordinateur. Les entreprises de l'aéronautique remplacent de plus en plus souvent des pièces physiques (des tiges et des cables) par des réseaux et du logiciel pour la commande des avions, mais elle doivent convaincre les autorités de certifications que le logiciel est aussi sûr ou plus sûr que les dispositifs physiques qu'il remplace. Ces entreprises sont intéressées à employer des ingénieurs qui savent comment garantir la correction de leur logiciel. En France, le Commissariat à l'Energie Atomique (le CEA) investit également beaucoup d'efforts sur ces approches.

2 Starting the Coq system

Pour travailler avec Coq, vous pouvez utiliser trois commandes. La première, appelée `coqc`, est utilisée comme un compilateur; la deuxième, appelée `coqtop`, est initialement prévue pour fonctionner directement dans une fenêtre de terminal; la troisième, appelée `coqide`, est un environnement interactif de développement: elle maintient une fenêtre où vos commandes sont enregistrées et d'autres fenêtres où les résultats les plus récents de l'outil sont affichés. Il est recommandé d'utilisé `coqide`, mais j'utilise personnellement une approche alternative, basée sur l'éditeur de text `Emacs` avec un mode spécialisé (appelé `Proof General`) qui fournit approximativement les mêmes possibilités que `coqide`.

Dans l'utilisation de `coqide`, il faut savoir que les commandes que vous tapez sont traitées seulement si vous demandez explicitement que ce traitement ait lieu, soit en cliquant sur une flêche qui apparait dans une rangée au dessus de la fenêtre, soit en tapant la combinaison appropriée de caractère spéciaux. L'outil `coqide` essaie de maintenir un historique précis de ce que vous avez fait et comment l'état de l'outil a été modifié. Ainsi, vous n'avez pas le droit de modifier les commandes qui sont enregistrées comme "déjà traitées" (elles apparaissent sur un fond de couleur différente). Pour modifier une commande déjà traitée, vous devez d'abord demander que l'outil annule le traitement. C'est également possible avec l'une des flêches qui apparaissent dans la rangée au dessus de la fenêtre.

Ces notes décriront seulement un sous-ensemble des possibilités offerte par l'outil Coq. Si vous voulez en savoir plus, ou si quelque chose n'est pas clair. Vous devriez suivre le pointeur suivant pour accéder à de la documentation:

`http://coq.inria.fr`

3 Premiers pas avec le langage de programmation

Le système Coq contient un langage de programmation très simple, mais aussi peu conventionnel.

Pour commencer, il est intéressant de demander que le système charge des objets qui sont déjà définis. Dans nos premières expériences nous utiliserons des valeurs booléennes, des nombres naturels et des listes. Donc nous devrions commencer avec la commande suivante.

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

Les nombres naturels ne ressemblent pas aux nombres que l'on trouve habituellement dans les langages de programmation: il sont tous positifs ou nuls.

Dans un langage de programmation, on a besoin de définir de nouvelles choses, on a aussi besoin de tester des données pour prendre des décisions en fonction de ces données et enfin on a besoin de répéter des opérations. C'est ce que nous allons illustrer dans les exemples suivants.

Pour définir de nouvelles choses, on utilise le mot-clef `Definition`.

```
Definition four := 4.

Definition polynom1 (x:nat) := x * x + 3 * x + 1.
```

Les types jouent un rôle important dans la programmation en Coq: La plupart des choses ont un type. Par exemple, le type des nombres 0, 1, 2 ... ont le type `nat`. Quand on écrit une fonction, comme la fonction `polynom1` ci-dessus, on donne habituellement le type des arguments attendus par cette fonction. C'est similaire à ce qui se passe habituellement dans les langages de programmation conventionnels.

A n'importe quel moment, vous pouvez utiliser une commande pour vérifier si une formule est bien formée. La commande est appelée `Check`.

```
Check polynom1 (polynom1 (polynom1 (polynom1 (polynom1 4)))).
```

Cette commande ne fait aucun calcul. Elle se contente de vérifier que les fonctions sont appliquées à des arguments du bon type.

Pour évaluer une expression, on peut utiliser une commande appelée `Compute`.

```
Compute polynom1 (polynom1 4).
```

Aver la représentation des nombres utilisée dans nos premiers essais, les calculs numériques sont très lents et utilisent beaucoup de mémoire. Il n'est pas recommandé d'essayer de calculer la grosse expression imbriquée qui était utilisée pour illustrer la commande `Check`. Mais cette représentation a d'autres avantages, que nous verrons au moment de raisonner sur nos programmes.

Dans ce langage, nous pouvons aussi construire des fonctions sans leur donner un nom. De telles fonctions utilisent existent seulement le temps qu'on les considère.

```
Check (fun x => x * x, fun x => x * polynome1 x).
```

Nous pouvons aussi définir des choses localement, de telle manière qu'elles sont oubliées en dehors la commande où elles sont utilisées. La construction s'appelle `let`.

```
Check let x := 3 * 4 in x * x * x.
```

3.1 Simple data structures and pattern-matching

Dans ces premiers pas, nous utiliserons 5 familles différentes de types de données.

3.1.1 Natural numbers

Le premier type de données sera le type des nombres. Pour définir un type de données, on décrit tous les cas possibles dans ce type. Pour les nombres naturels, la définition indique qu'il n'y a que deux cas: un nombre naturel est soit le nombre 0 soit le successeur d'un autre nombre naturel. Le nom interne pour le nombre 0 est écrit `O` (un o majuscule). Le nom interne pour l'autre cas s'écrit `S`. Ce `S` peut effectivement être utilisé comme une fonction qui prend en argument un nombre naturel.

La notation avec des chiffres, comme 13, 5, 8 est seulement une notation. Elle est fournie par le système Coq pour afficher les données d'une façon plus compréhensible, mais la donnée qui est vraiment manipulée en interne est réellement une expression symbolique de la forme `S (S (S (S (S O))))` pour le nombre 5. En fait la fonction `S` ne fait aucun calcul. Elle est là en mémoire c'est tout. On peut illustrer ceci avec la commande suivante:

```
Check (S O).
1 : nat
```

3.1.2 Les listes

La seconde famille de types de données que nous allons étudier est la famille des listes. C'est très similaire aux tableaux que nous avons l'habitude de voir dans les langages de programmation conventionnels. Le point commun est que les listes doivent aussi contenir des éléments qui ont tous le même type. La différence st que nous n'avons pas besoin de choisir à l'avance la taille d'une liste.

Comme pour les nombres naturels, il n'y a que deux cas dans les listes. Une liste est soit une liste vide, dont le nom interne est `nil`, soit un objet composé avec avec deux sous-composantes, avec le nom interne `cons`. La première sous-composante est le premier élément de la liste, la seconde sous-composante est une autre liste qui contient les autres éléments.

Le nom interne `cons` n'est pratiquement jamais utilisé directement. A la place nous utilisons la notation "::". Par exemple, nous pouvons construire une liste de trois éléments en utilisant la fonction `cons` mais la réponse du système utilise la notation.

```
Check cons 3 (cons 2 (cons 1 nil)).
3 :: 2 :: 1 :: nil
     : list nat
```

Notez que le type de l'objet considéré ci-dessus est `list nat`. Lorsque nous travaillons avec une liste, le système essaie toujours de connaître le type des éléments. Nous pouvons aussi construire des données plus complexes, comme des listes de listes.

3.1.3 Les couples

Les listes peuvent être utilisés pour grouper plusieurs choses ensemble dans un objet unique, mais tous les éléments doivent avoir le même type. Si nous voulons grouper ensemble des choses qui sont dans des types différents, nous utiliserons un autre type de données appelé le type des couples (*pair* en anglais et dans le système Coq). Nous écrirons simplement comme ce qui suit:

```
Check (3, 2::1::nil).
(3, 2::1::nil) : nat * list nat
```

L'objet composé que nous considérons ici est le couple d'un nombre naturel et d'une liste de nombres naturels. Il n'y a qu'un seul cas dans ce type de données.

3.1.4 Les valeurs booléennes

Dans la programmation nous utilisons souvent un type avec seulement deux éléments, appelés `true` et `false`. Ce type de données est appelé `bool` et il s'agit évidemment d'un type avec deux cas.

3.1.5 Valeurs optionnelles

Parfois nous avons parfois besoin d'écrire des programmes dont le comportemnt se décrit avec deux: soit ils retournent exactement une valeur, soit il retournent une condition exceptionnele. Par exemple, on peut vouloir définir une fontion qui prend en entrée un nombre naturel et retourne son prédécesseur lorsqu'il existe. Donc cette fonction envoie 2 vers 1, 1 vers 0. Mais pour l'entrée 0, nous voulons exprimer qu'il n'y a pas de nombre qui correspond. Nous pouvons choisir que cette fonction retourne un élément du type `option nat`. Ce type de données a deux cas, où le premier cas est appelé `Some` et a une sous-composante qui est la valeur que l'on veut retourner (lorsqu'elle existe). Le second cas est appelé `None` et n'a pas de sous-composante. Donc par exemple, pour construire une valeur de type `option nat` nous pouvons écrire l'expression suivante:

```
Check Some 3.
Some 3 : option nat
```

3.2 Observer, tester et prendre des décisions

Lorsque nous manipulons un objet dans l'un des types de données, nous pouvons avoir besoin de savoir dans quel cas chaque donnée se trouve et quelles sont les sous composante. Pour cela nous utilisons une *construction de filtrage* (*pattern matching construct* en anglais).

Le premier exemple simple apparait lorsque le type de donnée n'a qu'un cas. Si nous supposons que `p` est un couple, nous pouvons écrire l'expression suivante:

```
match p with
  (a, b) => ...
end
```

A l'intérieur de l'expression cachée par les petits points, nous sommes autorisés à utiliser `a` pour parler de la première composante de `p` et `b` pour parler de la deuxième composantes. Concrètement, `a` et `b` sont des variables locales qui sont définies et ne peuvent être utilisées qu'à l'intérieur de cette expression.

Pour les listes, c'est un peu plus compliqué, parce que le programme que nous écrivons doit être prêt pour tous les cas prévus par le type de données. Ainsi nous devons expliquer quelle valeur sera retournée si `p` est dans le cas `cons` (noté aved "`::`") et quelle valeur sera retournée si `p` est dans le cas `nil`.

```
match p with
  a::v => ...
| nil =>  ...
end
```

Donc la construction de filtrage *teste* si `p` est vide ou non, prend une décision en fonction de ce test et exécute le bout de code qui correspond, utilisant les variables `a` et `v`, pour accéder aux sous-composantes de la liste lorsque cela a un sens.

Si nous utilisons la construction de filtrage sur une valeur booléenne, nous n'observons pas de sous-composantes, mais nous prenons quand même des décisions. La construction de filtrage se comporte donc comme une expression *if-then-else*. C'est illustré par les notations du système Coq.

```
Check match true with true => 1 | false => 2 end.
if true then 1 else 2
     : nat
```

Nous pouvons illustrer le comportement de la commande de filtrage grace à la commande `Compute`.

```
Compute match true with true => 1 | false => 2 end.
 = 1 : nat

Compute match 3::2::1::nil with nil => 0 | a::v => a end.
 = 3 : nat
```

```
Compute match (3, true) with
  (a, b) => match b with false => 4 | true => 2 end
 end.
= 2 : nat
```

3.3 Appliquer des fonctions

Nous pouvons obtenir des fonctions de deux manières différentes. Soti ces fonctions ont été définies plus tôt et elles ont reçu un nom (par exemple avec `Definition` ou avec `let`) soit nous construisons une fonction pour l'utiliser directement. Pour appliquer une fonction à un argument, nous devons simplement écrire cette fonction à gauche de l'argument. Les quatre commandes suivantes sont correctes:

```
Check polynom1 1.
polynom1 1 : nat

Check (fun x => x * x) 3.
(fun x => x * x) 3 : nat

Check (fun x => (fun y => x + y)) 3.
(fun x => (fun y => x + y)) 3 : nat -> nat

Check (fun x => (fun y => x + y)) 3 4.
```

Dans l'avant-dernière commane, nous voyons que l'expression complète est une fonction et elle peut être appliquée à deuxième argument, tous simplement en l'écrivant à gauche de ce deuxième argument. C'est ce qui se passe dans la dernière commande. Notons bien qu'il n'est pas nécessaire d'ajouter des parenthèses (pas de parenthèse fermante entre 3 et 4).

3.4 Répéter des calculs

Pour répéter des calculs, le langage de programmation de Coq est encore moins conventionnel. Le point principal est que Coq refuse de aisser définir des fonctions qui peuvent boucler indéfiniment. La répétition est seulement autorisée à l'aide de la récursion, et la récursion est seulement autorisée lorsque l'argument principal de la fonction récursive devient plus plus petit. Dans nos premiers pas avec le système Coq, nous ne considérons que des fonctions récursives qui prennent des nombres et des listes comme arguments.

Voici un exemple simple de fonction avec récursion. Cette fonction prend en entrée une liste de nombres naturels et ajoute tous les éléments de cette liste.

```
Fixpoint slnat (l : list nat) :=
 match l with nil => 0 | a::tl => a + slnat tl end.
```

Premièrement, il faut noter que le mot-clef de cette définition est `Fixpoint`. Nous devons utiliser ce mot-clef si nous voulons définir une fonction récursive. Deuxièmement, la fonction utilise une construction de filtrage pour décrire comment les calculs s'effectuent. Dans le cas ou la liste en entrée est vide le calcul s'arrête et retourne 0. Dans le cas où la liste en entrée contient un premier nombre `a` et une liste plus petite `tl`, le calcul d'un appel récursif de `slnat` sur la liste plus petite est effectué. Ceci retourne un nombre naturel auquel `a` est ajouté. Donc la fonction exprime explicitement que l'argument sur lequel est effectué l'appel récursif est une liste plus petite.

Les appels récursifs ne doivent pas nécessairement être effectués sur des sous-composantes directes de l'argument initial. Par exemple, la fonction suivante est acceptée.

```
Fixpoint teo (l : list nat) :=
  match l with
    nil => nil
  | a::nil => a::nil
  | a::b::tl => a::teo tl
  end.
```

La fonction `teo` prend un élément sur deux de la liste en entrée. Dans le cas où la liste a la forme `a::b::tl`, l'appel récursif est effectué sur la liste `tl` qui est visiblement une sous-composante, mais pas une sous-composante directe.

Pour les fonctions récursives qui prennent des nombres naturels comme arguments, nous avons une structure similaire. La fonction récursive doit tester si l'argument est 0 ou non et un appel récursif est possible seulement lorsque le nombre est dans le cas `S`. Quand l'appel récursif est possible, il doit être faire sur la sous-composante, ou sa sous-composante, etc. Voici un exemple, où la fonction récursive construit la liste de tous les entiers inférieurs au nombre naturel en entrée.

```
Fixpoint lsn (n : nat) :=
 match n with
   0 => nil
 | S p => p :: lsn p
 end.
```

Nous voyons dans cet exemple que la construction de filtrage fournit un moyen simple de calcul le prédécesseur d'un nombre. Nous pouvons tester cette fonction à l'aide de la commande `Compute`.

```
Compute lsn 3.
 = 2::1::0::nil
```

Nous pouvons aussi définir des fonctions avec plusieurs arguments, mais la vérification sur les appels récursifs ne doit considérer qu'un seul des arguments. L'exemple suivant décrit une fonction d'addition.

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

4 Exercises

1. Définir une fonction qui prend en entrée une liste de nombres et retourne le produit de ces nombres,

2. Définir une fonction qui prend en entrée deux nombres et retourne `true` si et seulement si ces deux nombres sont le même entier naturel (une telle fonction existe déjà dans les bibliothèques de Coq, mais comment la définiriez vous?); vous devez seulement utiliser la construction de filtrage et la récursion: tester si les deux nombres sont 0, lorsque les deux nombres sont de la forme `S`, faites un appel récursif.

3. Définir une fonction qui prend en entrée un nombre naturel et retourne un élément de `option nat` contenant le prédécesseur s'il existe ou `None` si l'entrée est 0.

4. Définir une fonction qui prend en entrée une liste de nombres et retourne la longueur de cette liste.

5. A chaque nombre naturel nous pouvons associer la liste de ces chiffres de la droite vers la gauche. Par exemple, 239 devrait être associé à `9::3::2::nil`. Il est également naturel d'associer 0 à la liste vide `nil`. Si `l` est une telle liste, nous pouvons considérer son successeur comme une liste de chiffre. Par exemple, le successeur de `9::3::2` est `0::4::2`. Définir l'algorithme sur les listes de nombres naturels qui calcul le successeur de cette liste.

6. En supposant que `lsuc` est la fonction définie dans l'exercice précédent, vous pouvez définir `nat_digits` qui associent tout nombre naturel à la liste de chiffres qui le représente. Pour tout nombre naturel de la forme `S p`, il suffit de faire un appel récursif sur `p` puis d'appeler `lsuc` sur le résultat.

7. Dans le même contexte que l'exercice précédent, définir une fonction `value` qui envoie chaque liste de chiffres vers le nombre naturel qu'elle représente. Ainsi, `value (9::3::2::nil)` devrait retourner 239.

8. Dans le même contexte que précédemment, définir une fonction `licit` qui indique si une liste de nombres naturels correspond aux chiffres d'un nombre naturel: aucun chiffre ne devrait être plus grand que 9. Par exemple `licit (9::3::2::0::nil)` devrait retourner `true` et `licit (239::nil)` devrait retourner `false`.

5 More 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`