Une introduction rapide à Coq

Yves Bertot

Janvier 2009

1 Démarrer le système

Pour utiliser Coq on dispose de trois commandes, une commande qui sert à compiler des fichiers (`coqc`), une commande qui fournit une boucle d'interaction sur un terminal textuel (`coqtop`), et une commande pour avoir une interface graphique(`coqide`). Nous conseillons d'utiliser `coqide`, mais les exemples traités dans ces notes de cours peuvent aussi être étudiés avec `coqtop`.

Si vous utilisez `coqide`, vous voyez apparaitre une grande fenêtre divisée en trois parties. La partie de gauche est prévue pour recevoir le texte que vous écrivez, les deux fenêtres de droite sont prévues pour afficher le résultat de vos commandes. Lorsque vous avez écrit une commande complète dans la fenêtre de gauche, vous pouvez en demander l'exécution en cliquant sur la flêche vers le bas qui apparait en haut de la fenêtre. Les commandes exécutées changent de couleur pour indiquer qu'elles ont été exécutées, et il n'est plus possible de les modifier. Pour modifier une commande, il faut d'abord annuler son exécution, ce que l'on peut faire en cliquant sur la flêche vers le haut qui apparaît en haut. La commande revient alors dans la couleur de départ.

L'objectif de ces notes de cours est d'apprendre à faire quelques opérations en Coq, mais pour plus de détails, il vaut mieux se reporter au manuel de référence disponible à la page suivante:

`http://coq.inria.fr`

2 Apprendre le langage de programmation

Le sytème Coq contient un langage de programmation à la fois simple et compliqué. Dans cette première partie nous ne regarderons que la partie simple.

Pour commencer, il vaut mieux utiliser des éléments du langage qui sont déjà définis. Pour cela on utilise une commande pour charger des modules de la librairie du système. Nous utiliserons la commande suivante:

```
Require Import ZArith List String Recdef.
```

Ainsi nous pourrons utiliser du calcul sur les nombres entiers relatifs (type `Z`, et sur les listes. Pour indiquer que "1 + 3" doit être compris comme l'addition de deux nombres dans `Z`, nous devons ensuite ajouter la commande suivante:

```
Open Scope Z_scope.
```

Nous pouvons maintenant définir des valeurs et des fonctions en utilisant le mot clef `Definition`:

```
Definition quatre := 4.

Definition polynome1 (x:Z) := x^2 + 3 * x + 1.
```

A tout moment on peut demander à Coq de vérifier si une expression est bien formée, à l'aide de la commande `Check`. Cette commande est donc bien utile pour faire un essai sans changer l'état du système. Lorsque l'on vérifie le type d'une expression, on n'effectue pas le calcul décrit par cette expression.

```
Check polynome1 (polynome1 (polynome1 (polynome1 (polynome1 4)))).
```

On peut également demander à Coq d'exécuter une fonction ou de calculer une valeur:

```
Eval vm_compute in
  polynome1 (polynome1 (polynome1
      (polynome1 (polynome1 4)))).
```

Dans ce langage on peut aussi construire des fonctions sans leur donner de nom, et par exemple, mettre ces fonctions dans un couple:

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

2.1 Les structures de données et le filtrage

Dans tout langage de programmation, on peut avoir besoin de réunir plusieurs données ensemble. Dans Coq, on peut utiliser les couples, mais il est également possible d'utiliser des listes: dans une liste tous les éléments doivent avoir le même type, mais la longueur peut être arbitraire. Il existe une notation infixe pour ajouter un élément devant une liste (`::`) et une constante pour représenter une liste vide: `nil`.

Le langage de Coq utilise une approche différente des langages de programmation de la famille de `C, C++`, ou `Java` pour observer le contenu de structures de données. Cette approche combine deux aspects: un aspect de test pour vérifier que les données ont la bonne forme, et un aspect d'accès aux structures. Cette approche est appelé *filtrage*. On écrit une expression de la forme suivante:

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

L'expression cachée dans les petits points peut faire référence à `a` pour parler de la première composante du couple `p`. Ainsi, la variable `a` est utilisée pour accéder à une sous-partie de la donnée `p`. C'est à peu près la même chose pour les listes, sauf qu'il fait également dire ce que l'on fera si la liste que l'on observe est vide:

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

On voit ici que la commande de filtrage effectue un test de la forme *si-alors-sinon*, puisque la première zone de trois petits points correspond à ce que l'on fait *si la liste est non vide et de la forme* `a::v` et la deuxième zone de trois petits points correspond à ce que l'on fait *sinon*. En d'autres termes, il s'agit donc d'un style de programmation où l'on décrit ce qui se passe dans chaque cas, et le langage nous oblige à prévoir tous les cas possibles.

Par exemple, nous pouvons décrire une fonction qui reçoit une liste d'entiers, retourne l'entier contenu dans cette liste s'il n'y en a qu'un, et retourne 0 dans les autres cas, de la façon suivante:

```
Definition list1_to_Z (l : list Z) : Z :=
  match l with
    a::nil => a
  | _ => 0
  end.
```

2.2 Polymorphisme et arguments implicites

Pour chaque type T dans Coq, il existe un type `list` T dont les éléments sont des listes qui contiennent des objets de type T. La valeur `nil` est particulière, car elle doit appartenir au type `list` T pour un certain T. En général, le système Coq est capable de deviner le type T grâce au contexte, mais dans certains cas, il n'y parvient pas et émet un message d'erreur sybillin.

```
Check nil.
Error: Cannot infer the implicit parameter A of nil
```

Dans ce cas, il souvent judicieux d'indiquer au système Coq que l'on veut parler de `nil` comme élément d'un certain type de listes, avec une notation de "forçage du type" `( ... : ... )`.

```
Check (nil: list Z).
```

Plusieurs fonctions sont fournies dans le système Coq pour manipuler des listes sans tenir compte du type des éléments. Par exemple, il existe une fonction de concaténation de listes, appelée `app` et notée `... ++ ...`. Elle ne peut être utilisée qu'en concaténant des listes de même type:

```
Check app (1::2::nil) (3::4::nil).
(1::2::nil) ++ 3 :: 4 :: nil : list Z
```

Les fonctions de cette forme sont appelées *fonctions polymorphes*. Lorsque l'on demande à Coq si une telle fonction est bien formée on voit apparaître une nouvelle notation, qui indique que `app` appartient à toute une famille de types.

```
Check app.
app : forall A : Type, list A -> list A -> list A
```

2.3 Définir son propre type de données

Nous pouvons également définir un nouveau type de données en décrivant chacun des cas possibles pour ce type de données. Par exemple, nous pouvons définir un type de données avec deux cas, le premier qui regroupe deux entiers et le deuxième qui ne contient qu'une chaîne de caractères:

```
Inductive ex_type : Type :=
  case1 (n1 n2 : Z)
| case2 (s : string).
```

Pour construire des éléments de ce type, on va pouvoir utiliser directement les fonctions `case1` et `case2`, en leur donnant les arguments appropriés. Par exemple, voici deux définitions d'éléments du type `ex_type`.

```
Definition vex1 : ex_type := case1 1 3.
Definition vex2 : ex_type := case2 "hello world!".
```

Les fonctions qui décrivent chacun des cas sont appelées des *constructeurs*. Dans le type `ex_type` les constructeurs sont `case1` et `case2`.

Lorsque l'on écrit une fonction qui prend en argument un élément du type `ex_type` pour faire un traitement par cas, on doit donc couvrir les cas donnés par tous les constructeurs, ce qui va s'écrire comme dans l'exemple suivant:

```
Definition ex_type_to_Z (v : ex_type) :=
  match v with
    case1 n1 n2 => n1 + n2
  | case2 s => 0
  end.
```

2.4 La récursion

Dans un type de données, il est autorisé que certains champs de l'un des constructeurs soient dans le type même que l'on est en train de définir. Dans ce cas, une valeur du type pour contenir une autre valeur du même type, et ce répétitivement, de telle sorte qu'une telle donnée peut être potentiellement très grosse. Ceci est illustré dans l'exemple suivant:

```
Inductive String_Z_tree : Type :=
  bin_Z (n1 : Z) (t1 t2 : String_Z_tree)
| one_string (s : string) (t : String_Z_tree)
| sz3.
```

L'expression suivante appartient à ce type:

```
Check bin_Z 0 (bin_Z 1 sz3 (one_string "hello" sz3))
       (bin_Z 2 (bin_Z 3 sz3 sz3) (bin_Z 4 sz3 sz3)).
```

Lorsque l'on définit une fonction sur un tel type de données, il est autorisé d'inclure des appels récursifs, mais seulement sur les éléments du type qui sont des champs de constructeurs observés par un traitement par cas sur la valeur initiale. Une telle fonction récursive doit nécessairement être définie à l'aide du mot-clef `Fixpoint`. Par exemple, la fonction suivante est acceptée, cette fonction additionne toutes les valeurs entières qui apparaissent dans un terme de type `String_Z_tree`.

```
Fixpoint sum_sz_tree (t : String_Z_tree) : Z :=
  match t with
    bin_Z n t1 t2 => n + sum_sz_tree t1 + sum_sz_tree t2
  | one_string s t1 => sum_sz_tree t1
  | sz3 => 0
  end.
```

Ainsi, le système Coq autorise les fonctions récursives, mais seulement avec une discipline qui assure qu'aucune fonction récursive ne boucle indéfiniment.

Lorsque l'on définit une fonction récursive à plusieurs arguments, il est parfois nécessaire d'indiquer quel est l'argument pour lequel la discipline de récursion doit être respectée, ceci se fait avec le mot-clef `struct`. Par exemple, la fonction suivante remplace tous les sous-termes de la forme `sz3` par un nouveau terme, cette fonction prend deux arguments, mais c'est le premier qui décroît à chaque appel récursif.

```
Fixpoint subst_sz3 (t t' : String_Z_tree) {struct t}
   : String_Z_tree :=
  match t with
    bin_Z n t1 t2 => bin_Z n (subst_sz3 t1 t')
                        (subst_sz3 t2 t')
  | one_string s t1 => one_string s (subst_sz3 t1 t')
  | sz3 => t'
  end.
```

Les listes fournies par Coq sont justement un exemple de type de données où l'un des constructeurs contient un champ du même type. La définition des listes peut être observée à l'aide de la commande suivante:

```
Print list.
```

```
 Inductive list (A : Type) : Type :=
    nil : list A  | cons : A -> list A -> list A
For nil: Argument A is implicit
For cons: Argument A is implicit
For list: Argument scope is [type_scope]
For nil: Argument scope is [type_scope]
For cons: Argument scopes are [type_scope _ _]
```

Ce dialogue nous apprend que `list` est un type de données qui dépend d'un autre type, nommé `A` dans cette définition. Les deux constructeurs `cons` et `nil` prennent tous les deux un type en premier argument, mais celui-ci est implicite. Par ailleurs, la notation `a::l'` représente en fait l'expression `cons a l'` (mais ce n'est pas ce dialogue qui nous permet de l'apprendre).

Du point de vue de la programmation récursive, le constructeur `cons` contient un champ qui est dans le type `list`: les appels récursifs sont autorisés sur les champs de cette forme. Voici par exemple une fonction qui prend en entrée une liste contenant les entiers a_1, a_2, ..., a_n et retourne la liste $a_1 + a_2$, $a_3 + a_4$, $\ldots a_{2k} + a_{2k+1}$.

```
Fixpoint sum_pairs (l:list Z) : list Z :=
  match l with
    a::b::tl => (a+b)::sum_pairs tl
  | _ => l
  end.
```

Il faut noter que dans cet exemple, l'argument de l'appel récursif `tl` est bien un sous terme de l'argument initial `l`, même si ce n'est pas un sous-terme direct.

Il existe une façon de décrire les nombres entiers positifs ou nuls qui est particulièrement bien adaptée pour cette forme de récursion. C'est ce que l'on appelle les entiers de peano. Il s'agit de dire que tout entier positif ou nul est soit 0 soit le successeur d'un autre entier positif ou nul. Ceci s'exprime avec le type de données `nat` qui s'écrit de la façon suivante:

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

Dans ce type de données, le nombre 3 peut s'écrire `S (S (S O))`. Comme nous avons écrit `Open Scope Z_scope` en début de session, si l'on écrit 3, c'est un nombre de type `Z` qui est construit, mais si nous tapons `3%nat` ou `(3 + 4)%nat`, ce sont des expressions de type `nat` qui sont construites.

Avec ce type de données, les nombres ne sont pas représentés efficacement mais la programmation récursive est assez facile : pour définir une fonction récursive il suffit de savoir exprimer le résultat pour un nombre de la forme $n+1$ à partir du résultat de la même fonction pour le prédécesseur de ce nombre n. Par exemple, on peut décrire la fonction factorielle de la façon suivante:

```
Fixpoint fact (n:nat) : nat :=
  match n with O => 0%nat | S p => (n * fact p)%nat end.
```

On peut également définir la fonction de Fibonacci de la façon suivante, où l'on calcule la valeur de la fonction pour n et $n+1$ à la fois.

```
Fixpoint fibo (n:nat) : nat*nat :=
  match n with
    O => (0,1)%nat
  | S p =>
    match fibo p with
      (vp, vsp) => (vsp, vp + vsp)%nat
    end
  end.
```

Malheureusement, la fonction `fibo` ainsi définie est très inefficace et ne permet pratiquement pas de calculer la valeur pour des entrées moyennes, comme 1000.

2.5 Une forme plus générale de récursion

Les nombres du type `Z` ne sont pas définis comme les nombres du type `nat`. Il s'agit en fait de séquences de bits avec un signe, ce qui assure une représentation beaucoup plus compacte que la représentation de `nat`. L'avantage est certain, mais la contrepartie est que la programmation récursive est beaucoup moins aisée.

Coq fournit une commande qui permet de définir une fonction récursive sans suivre la structure imposée par le type de données. Il suffit d'être capable de fournir une fonction allant du type de l'argument vers le type `nat`, et d'être capable de démontrer que le résultat de cette fonction décroit à chaque appel récursif. La fonction en question est appelée une fonction de mesure.

Par exemple, si nous voulons définir une variante de la fonction de Fibonacci qui prend ses arguments dans le type `Z`, nous pouvons convenir que cette fonction retourne 0 pour tous les entrées négatives ou nulles et nous pouvons utiliser pour la mesure la fonction qui calcule la valeur absolue d'un nombre dans `Z` puis convertit le résultat en un entier du type `nat`. La fonction s'écrit de la façon suivante:

```
Function fibo_Z (n:Z) {measure Zabs_nat n} : Z*Z :=
  if Zle_bool n 0 then (0,1) else
  match fibo_Z (n-1) with
    (vn, vsn) => (vsn, vn + vsn)
  end.
```

Le système répond alors en demandant de prouver l'énoncé suivant:

```
1 subgoal

  ============================
   forall n : Z, Zle_bool n 0 = false  ->
     (Zabs_nat (n - 1) < Zabs_nat n)%nat
```

Il faut donc démontrer que si la fonction `Zle_bool` retourne `true`, alors la valeur absolue de `n - 1` est bien un nombre naturel strictement plus petit que la valeur absolue de `n`. Pour le démontrer, nous utilisons le théorème `Zle_cases`, qui permet de savoir dans quelles conditions `Zle_bool` retourne `true` ou `false` et le théorème `Zabs_nat_lt` qui permet de relier les comparaison entre nombres naturels et comparaisons entre nombres entiers relatifs (notez que nous utilisons la commande `Check` pour connaître l'énoncé de théorèmes).

```
Check Zle_cases.
Zle_cases
     : forall n m : Z,
       if Zle_bool n m then n  <= m else  n > m
Check Zabs_nat_lt.
Zabs_nat_lt
     : forall n m : Z, 0 <= n < m -> (Zabs_nat n < Zabs_nat m)%nat
```

Pour finir la démonstration que notre fonction s'arrêtera toujours, nous pouvons utiliser les quelques lignes suivantes:

```
intros n np; assert (np':= Zle_cases n 0); rewrite np in np'.
apply Zabs_nat_lt; omega.
Defined.
```

Le système répond en indiquant que plusieurs constantes sont définies, en particulier la fonction `fibo_Z` est définie. Il est trop tôt dans ces notes pour étudier présisément comment la commande utilisée fonctionne, mais il est bon de connaître cette commande pour les fonctions qui calculent récursivement sur les entiers par décrémentation successive.

nous pouvons maintenant tester cette fonction par exemple dans la commande suivante:

```
Eval vm_compute in let (a, _) := fibo_Z 1000 in a.
     = 43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
     : Z
```

La réponse est pratiquement instantanée.

Cet exemple montre que pour programmer en Coq on est parfois obligé de faire des preuves, au moins pour prouver que les fonctions dont on parle sont bien définies.

3 Les théorèmes en Coq

Vous pouvez connaître toutes les fonctions connues dans Coq qui retournent des nombres entiers en utilisant la commande suivante:

```
Search Z.
```

Vous pouvez connaître tous les objets de Coq qui parlent de `Z` en utilisant la commande suivante.

```
SearchAbout Z.
```

le résultat est énorme. Vous pouvez être plus précis en demandant à voir seulement les objets qui parlent de la division, par exemple:

```
SearchAbout Zdiv.
```

Par cette commande, nous apprenons par exemple que le système Coq connait des théorèmes. Par exemple, il existe un théorème `Z_div_mult` qui dit que si `b` est strictement positif, alors multiplier un nombre `a` par `b`, puis diviser par le même nombre `b` retourne le même nombre `a`. Dans la section 2.5, nous avons également vu deux théorèmes concernant les fonctions `Zle_bool` et `Zabs_nat`.

Parfois, il est difficile de savoir quelle est la fonction qui se cache derrière une notation. La solution est alors d'utiliser la commande `Locate`. Par exemple, la commande suivante nous permet de savoir quelle est le nom du prédicat qui se cache derrière le symbole de comparaison:

```
Locate "_ < _".
Notation             Scope
"x < y" := lt x y     : nat_scope
                       (default interpretation)
"x < y" := Zlt x y    : Z_scope
```

On peut ensuite taper la commande `Search Zlt.` pour connaître tous les théorèmes dont la conclusion porte sur une comparaison de la forme `x < y`.

3.1 Faire ses propres théorèmes

Pour démontrer de nouveaux théorèmes dans Coq, le système fournit une collection de commandes. La première est la commande `Lemma`[1], qui permet de démarrer une preuve en donnant le nom du théorème et son énoncé. L'énoncé est écrit avec des connecteurs logiques : `forall` (quantification universelle, "quel que soit"), `->` (implication, "si ... alors ..."), `exists` (quantification existentielle, "il existe"), `/\` (conjonction, "et"), `\/` (disjonction, "ou"), `~` (négation, "non"). Les formules atomiques sont dans le type `Prop` et les connecteurs logiques prennent des formules dans le type `Prop` et construisent de nouvelles formules dans le type `Prop`. Parmi les notions connues, on dispose de la comparaison des nombres, l'égalité, etc. Par exemple, on peut définir la notion "être un nombre premier" de la façon suivante:

```
Definition prime (n:Z) : Prop :=
  1 < n /\ forall x, 1 < x < n -> ~exists k, n = x * k.
```

[1] on peut aussi remplacer le mot-clef `Lemma` par `Theorem` si l'on veut souligner l'importance d'un résultat.

3.2 Preuve dirigée par les buts

Une fois que l'énoncé est accepté par Coq, il faut appliquer des commandes qui vont réduire cet énoncé en des énoncés plus simples jusqu'à qu'il soient solubles trivialement ou solubles par un outil automatique. Nous aurons besoin de connaître deux outils automatiques: (1) `omega` permet de prouver les comparaisons linéaires qui sont des conséquences des comparaisons linéaires déjà dans le contexte. On appelle comparaison linéaire une formule de comparaison dont le membre de droite et le membre de gauches sont des sommes de produits entre une constante numérique et une variable; (2) `ring` permet de prouver les égalités entre deux formules qui sont égales modulo l'associativité et la commutativité de l'addition et la multiplication et la distributivité. Un énoncé est soluble trivialement si la formule à prouver est déjà dans les hypothèses ou si c'est une égalité dont les deux membres sont égaux.

Pour simplifier un but, nous pouvons appliquer plusieurs commandes, adaptées pour chaque connecteur logique. Par exemple, si le but commence par une quantification universelle (`forall`) ou une implication `->`, il est souvent judicieux de commencer par une commande `intros`. Si une hypothèse est une conjonction, il est souvent judicieux de décomposer cette hypothèse pour récupérer séparément les deux sous-formules qu'elle contient. Si la formule à prouver est une conjonction, il est souvent judicieux de casser cette formule pour prouver séparément les deux sous formules qu'elle contient. Ces différentes commandes, appelées tactiques, sont récapitulées dans le tableau suivant:

	$\Rightarrow$	$\forall$	$\wedge$	$\vee$	$\exists$
Hypothesis	`apply`	`apply`	`elim`	`elim`	`elim`
goal	`intros`	`intros`	`split`	`left` or `right`	`exists` v
	$\neg$	$=$			

Hypothesis	`elim`	`rewrite`
		`injection`
		`discriminate`			

goal	`intro` `discriminate`	`reflexivity`			

La tactique `discriminate` n'est utilisable pour une négation que lorsque la formule niée est une égalité. La tactique `injection` est utilisable pour une hypothèse lorsque l'égalité dans cette hypothèse est une égalité entre deux formules qui commencent par le même constructeur d'un type inductif. La tactique `discriminate` sur une hypothèse est une égalité entre deux formules qui commence par des constructeurs différents. Il faut parfois faire attention à quelle commande est utilisée, par exemple, lorsque l'on doit prouver une disjonction, appliquer `left` ou `right` est un choix important, car cela permet de choisir celle des deux sous formules qui est prouvable.

Pour faire avancer une preuve, il peut également être judicieux de faire apparaître une formule logique intermédiaire, que l'on prouve séparément, et que

l'on peut ensuite réutiliser pour faire avancer la preuve principale. Pour effectuer cette étape de raisonnement, il suffit d'utiliser la command `assert`. comme dans l'exemple suivant:

```
assert (nle10 : n <= 10).
```

Le système Coq ajoute la formule `n <= 10` parmi les formules à prouver. Lorsque cette formule est prouvée, l'ancien but réapparaît avec l'hypothèse `nle10` est ajoutée dans son contexte.

3.3 Utilisation de théorèmes existants

Souvent, le meilleur moyen de faire progresser une preuve est d'utiliser un théorème existant. C'est possible dès que la conclusion de ce théorème peut s'instancier en la même formule que la conclusion du but. Dans ce cas, la tactique `apply` retrouve automatiquement les valeurs à donner aux variables quantifiées du théorème.

Il peut arriver que la comparaison entre la conclusion du but et la conclusion d'un théorème ne suffise pas à déterminer la valeur de toutes les variables quantifiées universellement dans le théorème. Dans ce cas, on doit apporter des informations supplémentaires à la commande `apply` en utilisant la directive `with`, comme dans l'exemple suivant:

```
apply Zle_trans with (m := S p).
```

La directive `with` peut également être utilisée pour des variables que le système Coq pourrait deviner.

Une autre façon d'instancier les variables quantifiées universellement d'un théorème est d'utiliser ce théorème comme une fonction, en appliquant simplement le théorème aux valeurs pour chaque variable et en mettant le tout dans une commande `assert`, avec une syntaxe différente (faite bien attention que nous utilisons la notation `:=` cette fois-ci).

Par exemple, si le théorème `Zle_cases` a l'énoncé suivant:

```
Zle_cases
     : forall n m : Z,
         if Zle_bool n m then <= m else n > m
```

la tactique suivante permet d'enlever les quantifications universelles sur `n` et `m` en les remplaçant par les valeurs `n` et `0`.

```
assert (npos' := Zle_cases n 0).
  ...
  npos' : if Zle_bool n 0 then n <= 0 else n > 0
```

3.4 Démonstrations par récurrence

Lorsqu'un énoncé logique porte sur une variable d'un type de données inductif, il est possible de faire une preuve par récurrence suivant la structure de ce type

de données. La preuve demande alors de vérifier que la formule à prouver est bien satisfaite dans chacun des cas décrits par les constructeurs, en ajoutant une hypothèse de récurrence pour chacun des champs des constructeurs qui sont déjà dans le type inductif.

Par exemple, pour les nombres naturels, nous pouvons faire une preuve par récurrence, nous allons montrer que pour `n` et `m` qui sont des nombres naturels, `n` est inférieur ou égal à `n + m`. Ce théorème est déjà connu de Coq, mais nous allons le re-démontrer en utilisant seulement les deux propriété suivantes:

```
le_n : forall n, n <= n
le_S : forall n m, n <= m -> n <= S m
```

Puisque nous voulons faire une preuve sur les nombres naturels, nous allons commencer par remettre en place l'interprétation des notations arithmétiques pour qu'elle concerne des expressions naturelles en priorité.

```
Close Scope Z_scope.
Lemma plus_le : forall n m, n <= m + n.
intros n m; induction m.
2 subgoals

  n : nat
  ============================
   n <= 0 + n

subgoal 2 is:
 n <= S m + n
```

L'étape de preuve engendre donc deux buts correspondant aux deux constructeurs du type `nat`. Nous ne voyons pas les hypothèses du second but pour l'instant. Concentrons nous sur le premier.

La définition de l'addition dans Coq fait que `0 + n` est la même chose que n̂. Nous pouvons demander à `coq` de changer ceci avec la tactique suivante:

```
change (0 + n) with n.
2 subgoals

  n : nat
  ============================
   n <= n
  ...
```

Nous pouvons maintenant appliquer le théorème `le_n`. La tactique pour cela est `apply`.

```
apply le_n.
1 subgoal
```

```
  n : nat
  m : nat
  IHm : n <= m + n
  ============================
   n <= S m + n
```

Nous voyons maintenant apparaître les hypothèses du second but. L'hypothèse `IHm` est une hypothèse de récurrence, qui indique que la propriété que nous voulons prouver est déjà satisfaite pour `m`. La conclusion du but indique que nous voulons prouver cette propriété pour `S m`. Ici encore, la définition de l'addition pour les entiers naturels est telle que `S m + n` et `S (m + n)` sont considérés comme la même expression, nous pouvons demander à Coq de faire apparaître cette similitude avec la tactique suivante:

```
change (S m + n) with (S (m + n)).
1 subgoal

  n : nat
  m : nat
  IHm : n <= m + n
  ============================
   n <= S (m + n)
```

Nous pouvons maintenant appliquer le théorème `le_S`, ce qui requiert ensuite que nous prouvions l'hypothèse de ce théorème. C'est facile car elle apparaît dans les hypothèses.

```
apply le_S.
1 subgoal

  n : nat
  m : nat
  IHm : n <= m + n
  ============================
   n <= m + n

trivial.
Proof completed.
Qed.
```

Quand la preuve est complète, il faut la terminer avec la commande `Qed.`

3.5 raisonner sur le calcul des fonctions

Les fonctions que nous avons définies en Coq peuvent apparaître dans les buts. Parfois, nous aurons besoin d'exprimer que le calcul d'une certaine fonction retourne une certaine valeur, simplement parce que c'est la définition de cette fonction. La façon la plus systèmatique de faire ce genre de calcul est d'appeler

la tactique `simpl`. Cette tactique parcourt toute la conclusion du but pour trouver des instances de fonctions récursives et faire progresser le calcul des ces fonctions récursives en respectant la définition de ces fonctions.

Parfois, la fonction `simpl` ne fait pas ce que l'on veut. Dans ce cas, il est possible d'être plus précis en utilisant la tactique `change` et en donnant précisément l'expression que l'on veut remplacer et l'expression que l'on veut mettre à la place. Il faut faire bien attention que le remplacement de l'une par l'autre doit bien être une conséquence directe de la définition des fonctions qui apparaissent dans les deux formules. Un exemple d'utilisation de la tactique `change` est déjà apparu dans la section 3.4.

4 Définir de nouveaux prédicats

On peut définir de nouvelles propriétés en indiquant par quelles règles ces propriétés peuvnt être prouvées et en précisant que toute preuve d'une de ces propriétés doit ultimement reposer uniquement sur ces règles.

Par exemple, on peut décrire l'ordre "inférieur ou égal" sur les entiers naturels en indiquant que tout nombre est inférieur à lui-même et que si un nombre `n` est déjà inférieur à `m`, alors il est également inférieur `S m`. Cette définition est exprimée de la façon suivante[2]

```
Inductive le (n : nat) : nat -> Prop :=
  le_n : le n n
| le_S : forall m, le n m -> le n (S m).
```

Les règles de raisonnement fournies dans la définition inductive sont encore appelées des constructeurs: ces constructeurs peuvent être utilisés comme des théorèmes. Ici les deux théorèmes fournis dans cette définition ont les énoncés suivants:

```
le_n : forall n:nat, le n n
le_S : forall n m:nat, le n m -> le n (S m)
```

Lorsque l'on a défini une propriété de cette manière, il est également possible de faire des preuves par récurrence sur la propriété: le système Coq engendre alors autant de buts que le nombre de constructeurs pour la définition inductive. Pour ceux des constructeurs qui ont en hypothèse une instance de la propriété définie dans cette définition inductive, le système Coq engendre aussi une hypothèse de récurrence.

Par exemple, nous pouvons prouver la réciproque du théorème étudié dans la section 3.4[3]

```
Lemma le_ex_plus : forall n m, le n m -> exists p, m = n + p.
Proof.
intros n m Hle.
```

[2] C'est de cette manière que l'ordre sur les entiers naturels est défini dans Coq.

[3] Nous vous invitons à rejouer cette preuve dans Coq.

```
induction Hle.
exists 0.
rewrite plus_0_r.
trivial.
```

La tactique `induction Hle` avait engendré 2 buts et les trois tactiques qui la suivent ont résolu le premier but. Le deuxième but a la forme suivante:

1 subgoal

n : nat
m : nat
Hle : n <= m
IHHle : exists p : nat, m = n + p
============================
exists p : nat, S m = n + p

L'hypothèse `IHHle` est une hypothèse de récurrence qui vient du fait que l'on suppose que la propriété recherchée est déjà satisfaite pour les nombres `n` et `m`, qui apparaissent dans l'hypothèse `Hle`. Nous pouvons utiliser cette hypothèse en suivant les indication de la table donnée en section 3.2.

```
destruct IHHle as [x meqnplusx].
exists (S x).
rewrite meqnplusx.
rewrite plus_n_Sm.
trivial.
Qed.
```

5 Plus d'information

Vous pourrez trouver plus d'information dans le livre [1] et le manuel de référence [3]. Il y a un tutoriel encore plus rapide que celui-ci dans [6]. Il existe également quelques tutoriaux en anglais sur le web [4, 2].

References

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

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

[3] 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`

[4] 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`

[5] 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`

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