UPMC/MASTER/INFO/STL/SVP
Spécification et Vérification de Programmes
Éléments d'une logique d'ordre supérieur avec le système Coq.

P. Manoury F. Peschanski

2015

Table des matières

1 Programmes et structures de données

Par *programme* il faut entendre *fonction*. Le style fonctionnel est celui des langages de la famille ML.

1.1 Fonctions booléennes

Le type des booléens est défini par les deux constantes `true` et `false`appelées *constructeurs* du type `bool`.

On peut définir des fonctions en utilisant la structure de contrôle usuelle `if-then-else` :

```
Definition negb (b:bool) : bool := if b then false else true.

Definition andb (b1 b2:bool) : bool := if b1 then b2 else false.

Definition orb (b1 b2:bool) : bool := if b1 then true else b2.
```

On spécifie le type des arguments et du résultat.

On peut aussi utiliser la construction `match-with` :

```
Definition negb (b:bool) : bool :=
  match b with
    true => false
  | false => true
  end.
```

1.2 Arithmétique

On travaillera avec des entiers idéaux et non des *entiers machines* (64 bits signés). On utilise l'ensemble des *entiers formels* de Peano qui est construit avec la constante 0 et la fonction *successeur* : *l'ensemble des entiers (naturels) est le plus petit ensemble qui contient 0 et qui est clos par la fonction successeur*. C'est-à-dire que les entiers formels sont toutes les expressions de la forme 0, $S0$, $SS0$, etc.

Cette définition correspond à un *type inductif* :

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

Le type `nat` représente le type (ou l'ensemble – `Set`) des entiers naturels. La définition dit :

1. `O` est un élément du type `nat`
2. `S` est une fonction (abstraite) qui, à tout entier associe un entier (type `nat -> nat`).

`O` et `S` sont les constructeurs du type `nat`.

Sur cette seule base, on reconstruit toute l'arithmétique, en commençant par l'addition :

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

C'est une *définition récursive* (mot clé `Fixpoint`, nous y reviendrons).

La multiplication est définie selon le même principe :

```
Fixpoint mult (n m:nat) : nat :=
  match n with
  | O => O
  | S p => (plus m (mult p m))
  end.
```

Pour définir la soustraction, il faut décomposer les deux arguments :

```
Fixpoint minus (n m:nat) : nat :=
  match n, m with
    O, _ => O
  | S p, O => n
  | S p, S q => (minus p q)
  end
```

Notez l'analyse par cas des deux arguments. Pour les entiers naturels (entiers positifs), $0 - m = 0$.

Fonctions partielles La division euclidienne (division avec des entiers) ne peut pas être définie directement car c'est une *fonction partielle* : $n/0$ n'est pas définie (dans les entiers).

Pour nous, quand une fonction `f` *est de type* `A -> B`, cela signifie que *pout tout* `a:A`*, il existe* `b:B` *tel que* `(f a) = b`. On ne pourra définir que des *fonctions totales*.

Le système Coq n'accepte une définition que s'il peut calculer le type de l'objet défini.

1.3 Listes paramètrées

Les listes paramètrées sont les listes *polymorphes* de ML. C'est un type inductif défini par les constructeurs `nil` et `cons` :

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

La fonction de calcul de la longueur d'une liste est définie récursivement :

```
Fixpoint length (A:Set) (xs: list A) : nat :=
  match xs with
    nil => O
  | (cons x xs) => (S (length A xs))
  end.
```

Stricto sensu, la fonction `length` a deux arguments : un ensemble (`A`) et une liste (`xs`). Toutefois, dans le définition de `length`, on peut s'arranger pour que la « valeur » du premier argument (l'ensemble `A`) soit implicite (puisque cette information figure dans le type du second argument – `(list A)`). On écrit :

```
Fixpoint length {A:Set} (xs: (list A)) : nat :=
  match xs with
    nil => O
  | (cons x xs) => (S (length xs))
  end.
```

L'argument implicite est indiqué entre accolades.

La concaténation :

```
Fixpoint app {A:Set} (xs ys : (list A)) : (list A) :=
  match xs with
    nil => ys
  | (cons x xs) => (cons x (app xs ys))
  end.
```

1.4 Fonctions partielles

Le type paramétré `option` est prédéfini en Coq

```
Inductive option (A:Type) : Type :=
  | Some : A -> option A
  | None : option A.
```

> Les valeurs du type `option` ne sont pas simplement des `Set`, mais des `Type`.

Ce type permet de « compléter » les définitions de fonctions partielles de manière à les rendre totales. Par exemple, la fonction qui donne l'élément d'une liste à une certaine position est partielle, on la rend totale de cette manière :

```
Fixpoint nth A:Set (i:nat) (xs:(list A)) : (option A) :=
  match i, xs with
    i, nil => None
  | 0, (cons x xs) => (Some x)
  | (S i), (cons x xs) => (nth i xs)
  end.
```

Notez la définition par cas sur les deux arguments `i` et `xs`.

2 Spécification

On entend par *spécification* toute *formule* ou ensemble de formules qui décrit les propriétés attendues d'une fonction.

2.1 Formules

Une formule est une expression qui a une *valeur de vérité*. Ces expressions ont le type `Prop`.

> Les valeurs de vérités qui ont le type `Prop` ne peuvent pas être confondues avec les valeurs booléennes, qui ont le type `bool`.

Le langage des formules que nous utiliserons est constitué des éléments suivants :

1. Des symboles de *prédicat* et de *relation*. Un symbole de prédicat est une *fonction de vérité* avec un type de la forme `A -> Prop`, où `A` est de type `Set`. Plus généralement, une relation est une fonction de vérité avec un type de la forme $\texttt{A}_1$ `->` ... `->` $\texttt{A}_n$ `-> Prop`.
2. L'implication entre deux formules (symbole `->`), la conjonction (symbole `/\`) et la disjonction (symbole `\/`) de deux formules, la négation (symbole `not`) d'une formule.
3. La quantification universelle (symbole `forall`) ou existentielle (symbole `exists`) typée d'une formule. On ne dira jamais simplement *pout tout x, blabla*, mais *pour tout x de type A, blabla*. Idem pour la quantification existentielle.

2.2 Spécification algébrique

Assez souvent, en programmation fonctionnelle, une propriété s'exprime simplement par une *équation*.

Par exemple, on spécifiera que l'addition est une opération associative, commutative et dont zéro est élément neutre à gauche et à droite :
– Associativité : `forall (x y z : nat), (plus (plus x y) z) = (plus x (plus y z)).`
– Commutativité : `forall (x y : nat), (plu x y) = (plus y x).`
– Neutre à gauche : `forall (x : nat), (plus 0 x) = x.`
– Neutre à droite : `forall (x : nat), (plus x 0) = x.`
Une spécification est une *formule close*, c'est-à-dire que toutes ses variables sont quantifiées.

On peut spécifier des propriétés analogues pour la concaténation de listes :
– Associativité : `forall (A:Set) (xs ys zs : (list A)), (app (app xs ys) zs) = (app xs (app ys zs)).`
– Neutre à gauche : `forall (A:Set) (xs : (list A)), (app nil xs) = xs.`
– Neutre à droite : `forall (A:Set) (xs : (list A)), (app xs nil) = xs.`
La concaténation n'est pas commutative.

> Avec une logique *du premier ordre*, on ne peut quantifier que sur des variables appartenant à une ensemble : `forall (x:nat), ...` avec `nat:Set`. Avec une logique *d'ordre supérieur*, on peut quantifier sur des ensembles : `forall (A:Set), ...`. On peut également quantifier sur des fonctions ou même des prédicats.

D'autres propriétés seront plus spécifiques, telle *la longueur de la concaténation de deux listes est égale à la somme des longueurs de listes* :
– `forall (A:Set) (xs ys : (list A)), (length (app xs ys)) = (plus (length xs) (length ys)).`

Égalité de fonctions Une spécification peut exprimer simplement le fait qu'une fonction est égale à une autre. Deux fonctions `f` et `g` sont égales lorsqu'elles ont le même type et qu'elles prennent les mêmes valeurs pour les mêmes arguments.

Par exemple, voici une définition altrenative de l'addition :

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

Le programmeur fonctionnel aura reconnu là une version *récursive terminale* de l'addition.

Une propriété attendue de `plus_alt` est qu'elle *implémente* la même fonction que `plus` :
– `forall (x y : nat), (plus_alt x y) = (plus x y).`

Équations conditionnelles Certaines égalités ne sont vraies que sous certaines conditions. Par exemple, si `max : nat -> nat -> nat` est la fonction qui donne le maximal de ses deux arguments, on a que *si `x` est plus grand que `y` alors `(max x y) = x`* :
– `forall (x y : nat), (ge x y) -> ((max x y) = x).`
où `ge : nat -> nat -> Prop` est la relation « supérieur ou égal à ».

2.3 Spécifications formelles

Utiliser toutes les ressources du langage logique pour spécifier les propriétés attendues des fonctions.

La fonction `remove` Opérationnellement, la fonction `remove` « retire » le $\texttt{i}^e$ élément de `xs`. Comment décrire le *résultat* de cette opération, c'est-à-dire, comment décrire `(remove i xs)` pour tout entier `i` et toute liste `xs` ?

Dénotationellement, et informellement

$$\texttt{(remove i [x}_0\texttt{;...;x}_{i-1}\texttt{;x}_i\texttt{;x}_{i+1}\texttt{;...;x}_n\texttt{])=[x}_0\texttt{;...;x}_{i-1}\texttt{;x}_{i+1}\texttt{;...;x}_n\texttt{]}$$

Cas limites :
- `(remove i [])` = `[]`
- $\texttt{(remove 0 [x}_0\texttt{; ...; x}_n\texttt{])=[...;x}_n\texttt{]}$
- $\texttt{(remove n [x}_0\texttt{; ...; x}_n\texttt{])=[x}_0\texttt{;...]}$
- $\texttt{(remove (n+k+1) [x}_0\texttt{; ...; x}_n\texttt{])=[x}_0\texttt{; ...; x}_n\texttt{]}$

Remarque : le dernier cas contient le premier.

Deux possibilités de formalisation :

1. Vision des listes comme des suites indicées (des tableaux).
   On vérifie que, pour tout `i`, pour tout `xs`, et pour tout `j`,
   si $\texttt{j} < \texttt{i}$ alors `(nth j (remove i xs))=(nth j xs)`
   et, si $\texttt{j} \geq \texttt{i}$ alors `(nth j (remove i xs))=(nth (j+1) xs)`
   Formule :
   ```
   forall (A:Set) (i:nat) (xs:(list A)) (j:nat),
     ((lt j i) -> (nth j (remove i xs)) = (nth j xs))
     /\ ((ge j i) -> (nth j (remove i xs)) = (nth (j+1) xs)).
   ```
2. Vision « monoïde » des listes.
   On vérifie que, pour tout `i` et pour tout `xs`,
   pour tout $\texttt{xs}_1$, `x` et $\texttt{xs}_2$, si $\texttt{xs=(app xs}_1\texttt{ (cons x xs}_2\texttt{))}$ avec $\texttt{(length xs}_1\texttt{)=i}$, alors
   $\texttt{(remove i xs)=(app xs}_1\ \texttt{xs}_2\texttt{)}$
   Formule
   ```
   forall (A:Set) (i:nat) (xs:(list A)) (xs1:(list A)) (x:A) (xs2:(list A)),
     (length xs1) = i -> (xs = (app xs1 (cons x xs2))
        -> (remove i xs) = (app xs1 xs2)))
   ```
   Notez comment les fonctions `length` et `app` sont des éléments de spécification.

L'algorithme *MJRTY* Algorithme de vote : on a une liste de valeurs et l'on veut savoir si un élément de cette liste y apparaît avec une fréquence strictement supérieure à la moitié du nombre d'éléments de la liste (valeur « strictement majoritaire »).

Boyer et Moore ont proposé un algorithmique qui sélectionne une valeur `x` dans la liste `xs` telle que : *pour toute valeur `x'` différente de `x`, le nombre d'occurence de `x'` dans `xs` est inférieur ou égale à la moitié du nombre d'éléments de `xs`.*
Pour savoir si la valeur sélectionnée a la majorité absolue, on compte son nombre d'occurence que l'on compare avec le nombre total d'éléments de la liste.

Pour simplifier, on considère des listes d'entiers.

Pour formaliser le résultat de la sélection, on se donne
- `length : (list A) -> nat` donne la longueur d'une liste, c'est-à-dire, son nombre d'éléments ;
- `count : A -> (list A) -> nat` donne le nombre d'occurrence d'une valeur dans une liste ;
- `search : (list nat) -> nat` donne le vainqueur potentiel ;

– `winner : nat -> (list nat) -> Prop` est telle que `(winner x xs)` vaut `True` si la nombre d'occurrences de `x` dans `xs` est supérieur strictement au nombre d'éléments de `xs`.

Le prédicat `winner` est définit par :

```
Definition winner (x:nat) (xs:(list nat)) : Prop :=
  (lt (length xs) (mul 2 (count x xs))).
```

La propriété attendue de la fonction de sélection est

```
forall (xs:(list nat)) (x:nat),
  not (x = (search xs)) -> not (winner x xs).
```

Le résultat globale de l'algorithme de vote est une valeur telle que, ou bien elle a la majorité absolue, ou bien aucune valeur n'a la majorité absolue :

```
forall (xs:(list A)),
  (winner (search xs) xs) \/ forall (x:A), not (winner x xs)
```

Préfixe d'une liste On propose la fonction suivante

```
Fixpoint is_prefix (xs ys : (list nat)) : bool :=
  match xs, ys with
    nil, ys => true
  | (cons x xs), nil => false
  | (cons x xs), (cons y ys) => (andb (nat_eqb x y) (is_prefix xs ys))
  end.
```

Nous prétendons que `(is_prefix xs ys)` vaut `true` si et seulement si `xs` ≪ est un préfixe de ≫ `ys`. Pour vérifier cela, il faut définir la relation ≪ est un préfixe de ≫.

Choisir

```
Definition Is_prefix (xs ys : (list nat)) : Prop :=
  ((is_prefix xs ys) = true).
```

donnerait guère de garantie sur la validité de notre fonction.

Il faut trouver autre chose. On peut utiliser la concaténation : `xs` ≪ est préfixe de ≫ `(app xs zs)`. On puet d'ailleurs vérifier que `(is_prefix xs (app xs zs))`=`true`. Mais on veut plus général :

```
Definition Is_prefix (xs ys : (list nat)) : Prop :=
  exists (zs:(list nat)), ys = (app xs zs).
```

On vérifie

1. la *correction* de `is_pefix` : si la fonction donne `true` alors le premier argument est un préfixe du second :
   ```
   forall (xs ys : (list nat)),
     (is_prefix xs ys) = true -> (Is_prefix xs ys).
   ```
2. la *complétude* de `is_pefix` : si `xs` est un préfixe de `ys` alors `(is_prefix xs ys)` vaut `true` :
   ```
   forall (xs ys : (list nat)),
     (Is_prefix xs ys) -> (is_prefix xs ys) = true.
   ```

3 Vérification

On entend par *vérification* toute *preuve (formelle)* qu'une formule de spécification est un théorème.

Une preuve (formelle) [1] est un ensemble de *séquents* reliés entre eux par des *règles de déductions*. Un séquent est un couple formé d'un ensemble de formules appelées *hypothèses* et d'une formule appelée *conséquence*. Logiquement, si toutes les hypothèses sont vraies alors la conséquence l'est aussi. Un séquent est trivialement valide lorsque la conclusion figure parmi les hypothèses. Une règle de déduction est une relation entre un séquent appelé *conclusion* et un ensemble de séquents appelés *prémisses*. Logiquement, si toutes les prémisses sont valides alors la conclusion l'est aussi.

Puisqu'une règle peut avoir plusieurs prémisses, les séquents sont reliés selon une structure arborescente. La racine est la formule à prouver. La preuve est achevée lorsque les feuilles sont soit des séquents trivialement valides, soit des séquents dont on possède déjà une preuve de la conclusion.

Construire une preuve, c'est
- se donner un *but* de preuve initial qui est un séquent dont l'ensemble d'hypothèses est vide et la conséquence est la formule à prouver ;
- lui appliquer une régle de déduction, ce qui engendre des *sous-buts* de preuve ;
- itérer le processus d'application de règles pour chaque sous-but qui n'est pas trivailement valide ;

Dans la pratique des systèmes interactifs de preuves informatisés Coq, une commande permet de déclarer le buts à prouver (`Theorem`, `Lemma`, etc.). Les règles sont implémentées commes des commandes de preuves appelées *tactiques*. Mais une tactique peut aussi regrouper plusieurs applications de règles, voire, lancer une *procédure de décision* sur le but courrant. À chaque application de tactique, les sous-buts engendrés sont empilés. La preuve est achevée lorsque cette pile a été vidée.

Notations : on note les séquents $\Gamma \vdash A$ où Γ est l'ensemble des hypothèses et A la conclusion. On peut distinguer une hypothèse en notant $\Gamma, H : B \vdash A$ où H est le nom de l'hypothèse et B la formule correspondante. On note les règles de déduction sous la forme $\dfrac{\Gamma_1 \vdash A_1 \dots \Gamma_k \vdash A_k}{\Gamma \vdash A}\ {}_R$ où $\Gamma_1 \vdash A_1 \dots \Gamma_k \vdash A_k$ sont les prémisses, $\Gamma \vdash A$ et R le nom de la règle. Une règle peut n'avoir pas de prémisse : $\dfrac{}{\Gamma, H : A \vdash A}\ {}^{Ax}$

On pourra ajouter parmi les prémisses d'autres indication que des séquents lorsque l'application d'une règle demande des vérifications annexes.

3.1 Évaluation symbolique

Le modèle du langage fonctionnel que nous utilisons est le *λ-calcul*. C'est un langage de programmation très rudimentaire que l'on peut baser les symboles réservés `fun`, `=>`, les parenthèses `(` et `)` et un ensemble de symboles atomiques (symboles de variables ou de constructeurs) disjoint des symboles déjà réservés. La grammaire du langage s'exprime ainsi :

$$t ::= a \mid \texttt{fun}\ x\ \texttt{=>}\ t \mid \texttt{(}t\ t\texttt{)}$$

Il définit l'ensemble t des *termes* du λ-calcul :

a tout symbole atomique est un terme.

`fun` x `=>` t l'*abstraction* du symbole de variable x vis-à-vis du terme t est un terme : la fonction de paramètre x et de corps t.

`(`t t`)` l'*application* d'un terme à un autre est un terme : le premier t est la fonction, le second, son argument.

On considère un sous ensemble des termes correspondants à ceux qui sont *correctement typés* vis-à-vis d'un ensemble de *règles de typages*. Nous ne développons pas ce point, pour l'instant.

1. Attention : ne pas lire les définitions données ici comme absolues mais comme relative à l'usage que nous aurons dans ce cours.

Le modèle de calcul associé à ce langage est la *β-réduction*, elle-même basée sur la *substitution* .

`(fun` x `=>` t u`)` se réduit en $t[u/x]$

L'écriture `(fun` x `=>` t u`)` se lit comme l'application de l'abstraction `fun` x `=>` t au terme u. L'écriture $t[u/x]$ se lit : t dans lequel u remplace (les occurrences libres de) x.

Avec les langages *à la ML*, ce noyau est enrichi de la construction `match-with` dont la forme générale est

`match` t `with` p_1 `=>` t_1 `|` $\ldots$ `|` p_n `=>` t_n `end`.

Les termes $p_1 \ldots p_n$ appartiennent à un sous-ensemble de termes appelés *motifs* (*pattern*, en anglais). Ils ne doivent contenir que des symboles de constructeurs ou des symboles de variables. De plus, ils sont *linéaires*, c'est-à-dire qu'un symbole de variable ne peut y avoir au plus qu'une seule occurrence.

La construction `if-then-else` est un cas particulier de `match-with` : le terme `if` e `then` e_1 `else` e_2 est égal à `match` e `with true =>` e_1 `|` `false =>` e_2 `end.`

La règle de réduction de la construction `match-with` met en jeu la notion de *filtrage* (*pattern matching*, en anglais) :

`match` t `with` p_1 `=>` t_1 `|` $\ldots$ `|` p_n `=>` t_n `end` se réduit en $t_i[u_1/x_1, \ldots, u_k/x_k]$
où i est le plus petit parmi $[1 \ldots n]$ tel que t se réduit en $p_i[u_1/x_1, \ldots, u_k/x_k]$.

Stricto sensu il existe une règle de réduction correspondant au fait de remplacer un symbole par sa définition, lorsqu'il en a une. Nous n'insistons pas ici sur ce point.

Évaluation et valeurs Pour se rapprocher des usages du domaine de langages de programmation, on emploiera le terme *évaluation* plutôt que réduction.

Le processus d'évaluation est la répétition de l'application de règles d'évaluation (*clôture transitive* de la relation définie par les règles d'évaluations).

C'est un *calcul formel* qui associe un terme à un autre terme. Ce calcul peut donner une ≪valeur≫ au sens usuel. Par exemple, `(plus (S 0) (S 0))` s'évalue en `(S (S 0))` qui est la ≪valeur≫ 2. Mais il peut avoir également pour résultat un terme que l'on ne considère pas en général comme une ≪valeur≫ dans la mesure où il peut contenir des variables dont la ≪valeur≫ est indéterminée. Par exemple `(plus 0 y)` s'évalue en `y`. On parle alors *d'évaluation symbolique*.

Évaluation et formules Dans une formule A, on peut remplacer un terme t par un terme u lorsque t s'évalue et u ou lorsque u s'évalue en t, sans changer la valeur de vérité de la formule. Cela donne deux règles de déduction :
– pour prouver $A[t]$, si t s'évalue en u, il suffit de prouver $A[u]$;
– pour prouver $A[u]$, si u s'évalue en t, il suffit de prouver $A[t]$.
Notons $t \hookrightarrow u$ pour ≪t s'évalue en u≫. On note ces règles

$$\frac{\Gamma \vdash A[u] \qquad t \hookrightarrow u}{\Gamma \vdash A[t]}\ evalC1 \qquad \frac{\Gamma \vdash A[t] \qquad t \hookrightarrow u}{\Gamma \vdash A[u]}\ evalC2$$

La même chose vaut pour les hypothèses :

$$\frac{\Gamma, H : B[u] \vdash A \qquad t \hookrightarrow u}{\Gamma, H : B[t] \vdash A}\ evalH1 \qquad \frac{\Gamma, H : B[t] \vdash A \qquad t \hookrightarrow u}{\Gamma, H : B[u] \vdash A}\ evalH2$$

Voir en Coq : les tactiques `simpl` et `simpl in`.

3.2 Propriétés équationnelles des fonctions

On en distingue deux espèces : celles qui s'obtiennent simplement par évaluation et les autres.

Équations de la première espèce L'équation `(andb true b) = b` s'obtient par évaluation.

En effet, comme `(andb true)` s'évalue en `b`, en appliquant la règle $evalC1$ au but `forall (b:bool), (andb true b) = b`, il rest à prouver que `b = b` ; ce qui est vérifié par réflexivité de l'égalité.

En Coq l'application de la réflexivité de l'égalité est implémentée par la tactique `reflexivity`.

Preuve de `(plus 0 y)` $\hookrightarrow$ `y` avec le système Coq :

```
Coq < Lemma and_true_left : forall (b:bool), (andb true b)=b.
1 subgoal

  ============================
   forall b : bool, (true && b)%bool = b

and_true_left < simpl.
1 subgoal

  ============================
   forall b : bool, b = b

and_true_left < reflexivity.
No more subgoals.
and_true_left < Qed.
simpl.
reflexivity.

and_true_left is defined

Coq <
```

L'existence des équations de première espèce nous dit que la relation d'évaluation *est contenue* dans la relation d'égalité puisque si $t \hookrightarrow u$ alors $t = u$.

Équations de la seconde espèce L'inverse n'est pas vrai. Par exemple, l'équation `(andb b true) = b` ne s'obtient pas simplement avec l'évaluation.

Cela tient à la manière dont la conjonction `andb` est définie :

```
Definition andb (b1 b2:bool) : bool := if b1 then b2 else false.
```

Dès lors, `(andb b true)` est égal à `if b then true else false`. Le processus d'évaluation est «bloqué» par la variable `b` dont il n'a pas la valeur.

```
Coq < Lemma andb_true_right : forall (b:bool), (andb b true) = b.
1 subgoal

  ============================
   forall b : bool, (b && true)%bool = b
```

```
andb_true_right < simpl.
1 subgoal

  ============================
   forall b : bool, (b && true)%bool = b

andb_true_right <
```

On se retrouve ≪bloqué≫.

3.3 Cas de constructeurs et induction

Preuve par cas de constructeurs Pour ≪débloquer≫ la situation, il faut envisager les deux valeurs possibles de `b`, c'est-à-dire ; *raisonner par cas* sur `b` :
– si `b=true` alors l'équation à montrer est `(andb true true) = true`, ce qui s'obtient par évaluation et réflexivité de l'égalité ;
– si `b=false` alors l'équation à montrer est `(andb false true) = false`, ce qui s'obtient de la même manière.
On parle de *cas de constructeurs* car `true` et `false` sont les deux constructeurs du type `bool` auquel appartient `b`.

On peut formuler ce *principe* de raisonnement avec les booléens par la règle

$$\frac{\Gamma \vdash A[\texttt{true}] \qquad \Gamma \vdash A[\texttt{false}]}{\Gamma \vdash \texttt{forall}(b : \texttt{bool}), A[b]}\ bool\_case$$

En Coq La tactique `destruct` permet un tel raisonnement par cas.

```
Coq < Lemma andb_true_right : forall (b:bool), (andb b true) = b.
1 subgoal

  ============================
   forall b : bool, (b && true)%bool = b

andb_true_right < destruct b.
2 subgoals

  ============================
   (true && true)%bool = true

subgoal 2 is:
 (false && true)%bool = false
andb_true_right < simpl; reflexivity.
1 subgoal

  ============================
   (false && true)%bool = false

andb_true_right < simpl; reflexivity.
No more subgoals.
andb_true_right < Qed.
destruct b.
```

```
 simpl; reflexivity.

 simpl; reflexivity.

andb_true_right is defined

Coq <
```

Pour tout type inductif il existe une règle de raisonnement par cas de constructeur. Pour le type `nat`, par exemple, on a

$$\frac{\Gamma \vdash A[\mathtt{0}] \qquad \Gamma, x : \mathtt{nat} \vdash A[\mathtt{S}x]}{\Gamma \vdash \mathtt{forall}(: \mathtt{nat}), A[x]}\ \mathtt{nat\_case}$$

Preuve par induction Examinons un cas assez similaire. L'équation `(plus 0 y) = y` s'obtient par évaluation (et réflexivité de l'égalité), mais pas `(plus x 0) = x`. La raison de cette disparité est nalaogue et tient à la manière dont `plus` est définie :

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

Elle est définie *par récurrence son permier argument (`n`)*. Dés lors, `(plus x 0)` s'évalue en

`match x with 0 => 0 | (S p) => S (plus p 0) end.`

Pour débloquer cette situation, un raisonnement par cas ne suffit pas : il faut un *raisonnement par induction structurelle* sur `x`. Le raisonnement par induction ajoute au raisonnement par cas une *hypothèse d'induction*. La règle de raisonnement par induction sur un entier peut s'écrire ainsi :

$$\frac{\Gamma \vdash A[0] \qquad \Gamma, x : \mathtt{nat}, H : A[x] \vdash A[\mathtt{S}x]}{\Gamma \vdash \mathtt{forall}(x : \mathtt{nat}), A[x]}\ nat\_ind$$

Appliquons ce principe avec la formule `(plus x 0) = x` :
- à montrer : `(plus 0 0) = 0`. C'est trivial.
- à montrer : `(plus (S x) 0) = (S x)`, sous les hypothèses `x:nat` et `H:(plus x 0) = x`.
  - En appliquant l'évaluation à `(plus (S x) 0) = (S x)`, reste à monter `S (plus x 0) = (S x)`.
  - En utilisant l'équation en hypothèse `H`, on remplace `(plus x 0)` dans `S (plus x 0) = (S x)` par `x` ; reste alors à monter `S x = S x`. Ce qui est trivial.

En logique il existe une règle qui permet d'utiliser des équations à la manière dont on utilise l'évaluation ; c'est-à-dire en remplaçant dans une formule un terme t par un terme u si l'on sait que $t = u$. Elle s'écrit

$$\frac{\Gamma \vdash A[u] \qquad \Gamma \vdash t = u}{\Gamma \vdash A[t]}\ eq$$

Plutôt que «remplacer», on dit également que l'on *réécrit* un terme par un autre selon une équation. La règle de *raisonnment équationnelle* est également appelée *règle de réécriture*.

En Coq la tactique `induction` implémente le raisonnment par induction et la tactique `rewrite`, la règle de réécriture.

```
Coq < Lemma plus_0_right : forall (x:nat), (plus x 0) = x.
1 subgoal
```

```
  ============================
   forall x : nat, x + 0 = x

plus_0_right < induction x.
2 subgoals

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

subgoal 2 is:
 S x + 0 = S x
plus_0_right < trivial.
1 subgoal

  x : nat
  IHx : x + 0 = x
  ============================
   S x + 0 = S x

plus_0_right < simpl.
1 subgoal

  x : nat
  IHx : x + 0 = x
  ============================
   S (x + 0) = S x

plus_0_right < rewrite IHx.
1 subgoal

  x : nat
  IHx : x + 0 = x
  ============================
   S x = S x

plus_0_right < trivial.
No more subgoals.
plus_0_right < Qed.
induction x.
 trivial.

 simpl.
 rewrite IHx.
 trivial.

plus_0_right is defined

Coq <
```