SU/FSI/MASTER/INFO/STL/MU5IN554
Spécification et Vérification de Programmes.

P. Manoury

sept. 2021

Par *programme* il faut entendre *fonction*. Le style fonctionnel est celui des langages de la famille ML. Le langage est celui dit système Coq. La *programmation récursive* repose sur la *récurrence structurelle* induite par la définition de *types inductifs*. Cette manière de définir les types de données et les fonctions les manipulant est complétée par la possibilité de *prouver* prouver les propriétés attendue d'une fonction par *induction structurelle*.
L'article de R.M.Burstall intitulé ≪*Proving Properties of Programs by Structural Induction*≫ (*The Computer Journal*, Volume 12, Issue 1, February 1969, Pages 41–48) donne une excellente illustration de ce point dans un cadre antérieur à l'apparition de l'outil formel que nous utiliserons.

1 Fonctions booléennes

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

On peut définir les fonctions booléennes de base en utilisant la structure de contrôle `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`, dite construction de *filtrage de motif* (*pattern matching*, en anglais) :

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

Dans l'expression `match b with true => false | false => true`, les constructeurs `true` et `false` sont appelés *motifs* (du filtrage).

En fait, l'expression `if` e `then` e_1 `else` e_2 est une abréviation pour `match` e `with true =>` e_1 `| false =>` e_2.

Le type `bool` est défini comme un *type inductif* :

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

Le type `bool` est lui-même de type `Set` qui est le type utilisé en général pour les structures de données.

C'est parce que le type `bool` est défini comme un type inductif, c'est-à-dire en termes de constructeurs, que l'on peut utiliser la structure de contrôle `match-with` pour décomposer les arguments booléens d'une fonction. On dit alors que la fonction est définie *par cas de constructeurs*.

Évaluation symbolique On munit notre *langage de programmation fonctionnelle* d'un système d'*évaluation symbolique*.

Le type énuméré `bool` est entièrement défini par les deux constructeurs `true` et `false`. Toute expression de type `bool` a pour *valeur*, soit la constante symbolique `true`, soit la constante symbolique `false`.

D'après la *définition* de la fonction `negb`, la valeur de l'application `(negb true)` est égale à la valeur de l'expression `match b with true => false | false => true` dans laquelle le *paramètre formel* `b` est remplacé par `true`. C'est-à-dire que la valeur de `(negb true)` est égale à la valeur de `match true with true => false | false => true`.

Le principe d'évaluation d'une construction de la forme `match` e `with true =>` e_1 `| false =>` e_2, lorsuqe e est de type `bool`, est exactement celui du `if-then-else` :

- si la valeur de e est `true` alors la valeur de `match` e `with true =>` e_1 `| false =>` e_2 est la valeur de e_1 ;
- sinon, la valeur de e est nécessairement `false`, et la valeur de `match` e `with true =>` e_1 `| false =>` e_2 est la valeur de e_2.

D'après ce principe, la valeur de `match true with true => false | false => true` est `false`.

D'après ce même principe, on obtient également que la valeur de `(andb true b)`, quelque soit le booléen `b`, est égale à la valeur de `b`. C'est pourquoi, on parle d'évaluation *symbolique*. La «valeur» d'une application obtenue par évaluation symbolique peut contenir des inconnues.

On pourra utiliser le terme de *réduction* ou de *simplification* à la place d'évaluation.

Si l'on compose deux applications de `negb`, on pose que la valeur de l'expression `(negb (negb true))` est celle de `(negb false)`, puisque la valeur de `(negb true)` (paramètre d'appel du premier `negb`) est `false`. La valeur de `(negb false)` est celle de `match false with true => false | false => true` ; c'est-à-dire : `true`. Ce principe général d'évaluation des applications est appelé *appel par valeur* (en anglais : *call by value*).

Évaluation symbolique et égalité Nous avons vu que c'est une propriété générale de la conjonction que *pour tout booléen* b, `(andb true` b`)` $= b$.

On peut exprimer cette propriété dans le système Coq :

```
forall (b:bool), (andb true b) = b.
```

Mieux, on peut y vérifier sa validité. Le schéma de preuve est le suivant :

1. On suppose un `b:bool` quelconque.
2. On applique l'évaluation symbolique (`(andb true b)` devient `b`).
3. On applique la réflexivité de l'égalité (`b = b`).

Pour démontrer notre propriété dans le système Coq, il faut, dans un premier temps se la donner comme *but de preuve*.

```
Fact andb_true : forall (b:bool), (andb true b) = b.
```

Le mot clé `Fact` indique au système que l'on désire prouver une formule. On peut libéralement utiliser `Lemma` ou `Theorem`. L'identificateur `andb_true` est le nom que l'on donne à la formule démontrée. Vient ensuite la formule elle-même.

Pour démontrer effectivement notre formule, on applique successivement trois *commandes de preuves*, appelées également *tactiques*, correspondant aux trois étapes de notre schéma de preuve. À chaque application d'une tactique, le *but de preuve* est modifié.

Voici ce qui se passe si nous utilisons le mode interactif de base du système Coq (commande `coqtop` depuis un terminal).

```
othe13:~ eleph$ coqtop
Welcome to Coq 8.5 (April 2016)

Coq < Fact andb_true : forall (b:bool), (andb true b) = b.
1 subgoal

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

andb_true < intro.
1 subgoal

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

andb_true < simpl.
1 subgoal

  b : bool
  ============================
   b = b

andb_true < reflexivity.
No more subgoals.

andb_true < Qed.
intro.
simpl.
reflexivity.

Qed.
andb_true is defined

Coq <
```

Quelques remarques et commentaires :

- notez qu'après la déclaration du but de preuve initial, le *prompt* du système a changé (de `Coq <` à `and_true <`;
- notez la variante de syntaxe : le système affiche `(true && b)%bool` là où nous avions écrit `(andb true b)`;
- un *but de preuve* est constitué d'un ensemble d'hypothèses et d'une formule séparés par une ligne de

symboles `=` ;
- la commande `intro` fait remonter la déclaration `b:bool` en hypothèse. Elle est l'équivalent de ≪*supposons un* `b:bool` *quelconque* ;
- la commande `simpl` applique le processus d'évaluation symbolique à la formule à prouver ;
- la commande `reflexivity` invoque la réflexivité de l'égalité (connue du système) ;
- après cette étape, le système indique qu'il n'y a plus rien à prouver (`No more subgoals`) ;
- on enregistre le nom, la formule et la preuve elle-même avec la commande `Qed` ;
- notez que le *prompt* `Coq <` est rétabli.

Raisonnement par cas de constructeur Une autre propriété de la conjonction est

```
Fact andb_true2 : forall (b:bool), (andb b true) = b.
```

Mais on ne peut établir celle-ci par simple évaluation symbolique :

```
Coq < Fact andb_true2 : forall (b:bool), (andb b true) = b.
1 subgoal

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

andb_true2 < intro.
1 subgoal

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

andb_true2 < simpl.
1 subgoal

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

andb_true2 <
```

Ici, la commande `simpl` n'a pas modifié le but de preuve car, il n'existe aucune règle pour évaluer une expression de la forme `match b with true => ...| false => ...` lorsque `b` n'est ni la constante `true`, ni la constante `false`.

En revanche, nous savons, par hypothèse que `b:bool`. Par définition de `bool`, la variable `b` ne peut prendre que 2 valeurs : soit `true`, soit `false`. On peut donc *raisonner par cas* sur `b`. Dans chaque cas, on remplace `b` par l'un des deux constructeurs et l'on obtient l'égalité voulue par évaluation symbolique (et réflexivité).

Voici le déroulé de cette preuve en Coq :

```
1 subgoal

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

andb_true2 < case b.
```

```
2 subgoals

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

subgoal 2 is:
 (false && true)%bool = false

andb_true2 < simpl.
2 subgoals

  b : bool
  ============================
   true = true

subgoal 2 is:
 (false && true)%bool = false

andb_true2 < reflexivity.
1 subgoal

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

andb_true2 < simpl.
1 subgoal

  b : bool
  ============================
   false = false

andb_true2 < reflexivity.
No more subgoals.
```

Remarques et commentaires :

- la commande `case b` applique le raisonnement par cas (de valeurs booléennes) sur `b`. Notez comment cette commande engendre 2 buts de preuves.
- chaque cas est traité successivement ;
- lorsque le premier cas a été traité, il disparaît au profit du second.

Script de preuve alternatif

```
Coq < Fact andb_true2 : forall (b:bool), (andb b true) = b.
1 subgoal

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

andb_true2 < destruct b.
2 subgoals
```

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

subgoal 2 is:
 (false && true)%bool = false

andb_true2 < reflexivity.
1 subgoal

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

andb_true2 < reflexivity.
No more subgoals.
```

Commentaires :

- nous avons utilisé la tactique `destruct` en place de `case`. On peut utiliser cette tactique lorsque l'on veut raisonner par cas sur une variable liée. L'utilisation de `destruct` est ici équivalenete à `intro. case b`;
- nous n'avons pas explicité les étapes d'évaluation symbolique avec la tactique `simpl`. Le système a été capable d'égaliser lui-même les termes des égalité par évaluation symbolique.

Le principe de raisonnement par cas est le pendant logique de la construction `match-with`. Il est fournit par le système d'après la définition du type (inductif) `bool` avec la clause `Inductive`. Nous allons voir avec la définition Coq du type des entiers comment ce principe se généralise en *principe d'induction structurelle*.

2 Arithmétique et fonctions récursives

Pour notre introduction à la programmation récursive, on travaillera avec des *entiers naturels* et non des entiers relatifs, non plus que des *entiers machines* (64 bits signés, par exemple). Plus précisément, on utilise l'ensemble des entiers formels dit *de Peano* qui est construit avec la constante `0` et la fonction *successeur*, notée `S`. Intuitivement la fonction `S` est l'opération d'ajout d'une unité ($x \mapsto 1 + x$). Les deux symboles `0` et `S` sont les *constructeurs* du type `nat`.

Les expressions construites avec les constructeurs `0` et `S` ont la forme suivante : `0`, `(S 0)`, `(S (S 0))`, `(S (S (S 0)))`, etc. Chacune d'elle représente un nombre entier (naturel) : le nombre de symboles `S` qu'elle contient. Toutes ces expressions sont les valeurs du type `nat` et, réciproquement, toute expression de type `nat` a pour valeur construite uniquement avec les symboles `0` et `S`. Il y en a une infinité (*dénombrable*). Du point de vue théorique : *l'ensemble des entiers (naturels) est le plus petit ensemble qui contient 0 et qui est clos par la fonction successeur.*

Définition du type `nat` dans le système Coq (module Datatypes) :

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

Ce type mérite pleinement son qualificatif d'*inductif*, puisque le constructeur `S` a besoin d'un entier pour donner un nouvel entier.

Le mécanisme de filtrage de la construction `match-with` est applicable aux expressions de type `nat`. Toutefois, il va différer un peu de ce que nous avions vu avec le type (fini) des booléens. Son utilisation la plus simple consiste à distinguer une valeur entière nulle (constructeur `0`) d'une valeur non nulle (constructeur `S`). Par exemple, on définit de cette manière la fonction *prédécesseur*, inverse de l'opération successeur, avec la particuliarité que, sur les entiers naturels, le prédécesseur de `0` est lui-même :

```
Definition pred (n:nat) : nat :=
  match n with
  | 0 => 0
  | (S p) => p
  end.
```

Ce qui est nouveau ici, par rapport aux booléens, c'est l'utilisation du *symbole de variable* `p` dans le motif `(S p)`. Ce motif a deux propos :

1. *reconnaître* que la valeur de `n` est un entier non nul, puisque l'expression de cette valeur «commence» par l'application du constructeur `S` ;
2. *nommer* la valeur de l'entier qui suit l'application du «premier» constructeur `S` : si `n` a la valeur `(S 0)` alors, `p` prend la valeur `0` ; si `n` a la valeur `(S (S 0))`, alors `p` prend la valeur `(S 0)`, etc.

Du point de vue de l'évaluation symbolique la valeur de l'application `(pred (S (S 0)))` est la valeur l'expression `match (S (S 0)) with 0 => 0 | (S p) => p`. Selon le principe d'évaluation du filtrage, avec liaison de la variable de filtrage `p`, cette expression a pour valeur `(S 0)`. Ici, l'évaluation de `match (S (S 0)) with ...`

1. *branche* le processus d'évaluation sur le cas de filtrage `(S p)`.
2. *lie* l'expression `(S 0)` à la variable de motif `p`.

Techniquement, on obtient la liaison de `p` à `(S 0)` car l'expression `(S (S 0))` est syntaxiquement égale au motif `(S p)` où l'on remplace `p` par `(S 0)`.

Définitions récursives de fonctions Sur la base des deux constructeurs `0` et `S`, jointe à la possibilité d'analyser une valeur entière par filtrage et à celle de définir des fonctions récursives, on reconstruit toute l'arithmétique.

Commençons par l'addition :

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

Mathématiquement, on a simplement dit ici que $0 + m = m$ et que $(1 + n) + m = 1 + (n + m)$.

Opérationnellement : pour faire la somme de n et m, il suffit d'ajouter n fois 1 à m («ajouter 1» étant représenté par le constructeur `S`). Ainsi, on réduit l'opération d'addition à l'*itération* de l'opération primitive `S`. L'itération est réalisée par la *définition récursive* (mot clé `Fixpoint`).

Intuitivement, le calcul exprimé par notre définition de l'addition ressemble à ce que l'on écrirait en C de la manière suivante :

```
int add(int n, int m) {
  int r = m;
  while (n > 0)  r = r+1; n = n-1;
  return r;
```

}

Type et terminaison Toute définition, et donc toute définition de fonction, est soumise à une *vérification de type*. Mais la vérification effectuée par les système Coq est beaucoup plus exigente que celle que l'on trouve dans les langages de programmation.

La fonction `add` est de type `nat -> nat -> nat`. Ce qui signifie que, sous l'hypothèse où les expressions `e1` et `e2` sont de type `nat` alors l'application `(add e1 e2)` est également de type `nat`. Mais là où Coq est plus exigeant, c'est qu'il réclame qu'effectivement il existe une valeur `v` de type `nat` telle que `(add e1 e2)`=`v`. Vérifier cela, c'est *prouver* que : *pour tout* `n:nat`, `m:nat`, *il existe* `r:nat` *tel que* `(add n m)`=`r`. En d'autres termes, il s'agit de *prouver* que la fonction `add` construit effectivement une valeur (de type `nat`) quelles que soit la valeur (de type `nat`) de ses entrées ; c'est-à-dire qu'il faut *prouver* que le calcul décrit par la définition de `add` termine toujours. Or, c'est un résultat connu en théorie de la calculabilté : le problème de l'arrêt est indécidable. Donc il n'y a pas de méthode automatique pour prouver qu'une fonction termine toujours.

Toutefois, la définition que nous avons donnée de `add` est acceptable car un argument général peut garantir qu'elle termine toujours. En effet, dans la définition donnée, l'appel récursif de `add` a pour premier argument une obtenue par suppression du symbole `S` en tête de l'expression qui désignait ce premier argument dans l'appel d'origine. L'argument de l'appel récursif est donc *structurellement* plus petit que l'argument d'origine (il s'écrit avec un symbole `S` de moins). Cette diminution de taille de l'expression suffit à garantir en général la terminaison. Il est facile de vérifier cette *propriété syntaxique* dans notre définition. Ce qui permet d'accepter notre définition.

On peut bien entendu accepter des diminutions plus importantes, comme dans :

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

Évaluation symbolique de `(add (S (S 0)) (S 0))`.
Cette expression a pour valeur celle de l'expression `match (S (S 0)) with 0 => (S 0) | (S p) => (S (add p (S 0)))`. On a obtenu cette expression en remplaçant les paramètres formels `n` et `m` de la définition de `add` par, respectivement `(S (S 0))` et `(S 0)`. Selon le principe de l'évaluation du filtrage, `p` prend le valeur `(S 0)` et notre expression a pour valeur celle de `(S (add (S 0) (S 0)))`. Par définition de `add`, cette expression a pour valeur celle de

```
(S (match (S 0) with 0 => (S 0) | (S p) => (S (add 0 (S 0)))))
```

(notez le `S` en début d'expression). Par évaluation du filtrage, on obtient `(S (S (add 0 (S 0))))`, dont la valeur est celle de

```
(S (S (match 0 with 0 => (S 0) | (S p) => (S (add p (S 0))))))
```

Ce qui donne, pour ce cas du filtrage : `(S (S (S 0)))`.

Nous résumons ces étapes dans la table suivante où le symbole $\rightsquigarrow$ se lit comme «*s'évalue symboliquement en*» :

```
(add (S (S 0)) (S 0)
⇝ match (S (S 0)) with 0 => (S 0) | (S p) => (S (add p (S 0)))
⇝ (S (add (S 0) (S 0)))
⇝ (S (match (S 0) with 0 => (S 0) | (S p) => (S (add 0 (S 0)))))
⇝ (S (S (add 0 (S 0))))
⇝ (S (S (match 0 with 0 => (S 0) | (S p) => (S (add p (S 0))))))
⇝ (S (S (S 0)))
```

Évaluation symbolique et égalité Nous savons que 0 est un élément neutre pour l'addition. En particulier, $0 + m = m$ et l'évaluation symbolique nous permet d'établir cette propriété :

```
Coq < Fact add_0 : forall (m:nat), (add 0 m) = m.
1 subgoal

  ============================
   forall m : nat, add 0 m = m

add_0 < intro.
1 subgoal

  m : nat
  ============================
   add 0 m = m

add_0 < simpl.
1 subgoal

  m : nat
  ============================
   m = m

add_0 < reflexivity.
No more subgoals.
```

Pour ce qui est de la propriété `(add n 0) = n`, qui ressemble à `(andb b true) = b`, on pourrait essayer de l'établir avec un raisonnement par cas. Mais voici ce qui se passe :

```
Coq < Fact add_02 : forall (n:nat), (add n 0) = n.
1 subgoal

  ============================
   forall n : nat, add n 0 = n

add_02 < intro.
1 subgoal

  n : nat
  ============================
   add n 0 = n

add_02 < case n.
2 subgoals

  n : nat
  ============================
   add 0 0 = 0

subgoal 2 is:
 forall n0 : nat, add (S n0) 0 = S n0

add_02 < simpl.
```

```
2 subgoals

  n : nat
  ============================
   0 = 0

subgoal 2 is:
 forall n0 : nat, add (S n0) 0 = S n0

add_02 < reflexivity.
1 subgoal

  n : nat
  ============================
   forall n0 : nat, add (S n0) 0 = S n0

add_02 < intro.
1 subgoal

  n, n0 : nat
  ============================
   add (S n0) 0 = S n0

add_02 < simpl.
1 subgoal

  n, n0 : nat
  ============================
   S (add n0 0) = S n0

add_02 <
```

Si, dans le premier cas, lorsque `n` est remplacé par `0`, tout se passe correctement, il n'en va pas de même dans le cas où `n` est remplacé `(S n0)`, avec `n0:nat` quelconque. L'évaluation symbolique appliquée à la formule `(add (S n0) 0) = S n0` nous donne `(S (add n0 0)) = S n0`. Et nous ne savons que faire du terme `(add n0 0)` qui «revient», formellement, au terme dont nous sommes partis.

Un simple raisonnement par cas sur `n` n'est donc pas suffisant ici.

Principe d'induction structurelle Le raisonnement par cas a été suffisant pour les booléens car il remplace la variable sur laquelle on raisonne par cas par l'une des deux valeurs possibles pour les booléens ; et alors l'évaluation symbolique peut opérer pleinement. Mais, avec les entiers, dans le cas du constructeur `S`, il y a une infinité de valeurs possibles. Le principe de raisonnement par cas ne peut les énumérer toutes et il se contente de donner la *forme générale* des expressions qui tombent sous ce cas, à savoir `(S n0)` avec le nouveau symbole de variable `n0` qui bloque le processus d'évaluation symbolique :

```
(add (S n0) 0)
⇝ match (S n0) with 0 => 0 | (S p) => (S (add p 0))
⇝ (S (add n0 0))
⇝ (S (match n0 with 0 => 0 | (S p) => (S (add p 0))))
```

Examinons la formule sur laquelle nous bloquons : `(S (add n0 0)) = (S n0)`. Les termes de cette égalité commencent tout deux par l'application de `S`. Comme c'est une fonction, on saura égaliser ces deux termes,

si l'on sait égaliser `(add n0 0)` et `n0`. C'est-à-dire montrer `(add n0 0) = n0` pour un `n0` quelconque. En d'autres termes, on pourrait débloquer la situation en montrant

`forall (n:nat), (add n 0)=n -> (add (S n) 0)= (S n)` .

En effet, si l'on a

(1) `(add 0 0)=0`

(2) `forall (n:nat), (add n 0)=n -> (add (S n) 0)= (S n)`

Alors, par (1) on a `(add 0 0)=0` ; de celà et (2), on déduit `(add (S 0) 0)=(S 0)` ; de celà et (2), on déduit `(add (S (S 0)) 0)=(S (S 0))` ; etc. Comme chaque valeur concrète de type `nat` s'écrit comme une suite finie de `S` appliquées à `0`, (1) et (2) suffisent à établir l'égalité cherchée.

Cela est vrai de toute propriété portant sur un entier et est résumé par le principe *d'induction structurelle* : pour toute propriété sur les entiers `P: nat -> Prop`,

1. Si `(P 0)`
2. Si, pour tout `n0:nat`, `(P n0) -> (P (S n0))`.
3. Alors, pour tout `n:nat`, `(P n)`.

Preuve de `(add n 0)=n` Utilisons donc le principe d'induction structurelle pour montrer que pour tout `n:nat`, `(add n 0) = n`. Par induction sur `n:nat`, il faut montrer :

1. `(add 0 0)=0`
2. si `(add n0 0)=n0` alors `(add (S n0) 0) = S n0`

— Preuve de `(add 0 0)=0` :
par évaluation, il faut montrer `0 = 0`, ce qui est donné par réflexivité de l'égalité.

— Preuve de si `(add n0 0)=n0` alors `(add (S n0) 0) = S n0` :
on suppose (HR)`(add n0 0)=n0` et on montre `(add (S n0) 0) = n0`. Par évaluation, il faut montrer `(S (add n0 0)) = (S n0)`. Par (HR), on peut remplacer `(add n0 0)` par `n0` ; reste alors à montrer que `(S n0) = (S n0)`. Ce qui est donné par réflexivité de l'égalité.

Nous transcrivons cette preuve dans le système Coq :

```
Coq < Lemma add_02 : forall (n:nat), (add n 0) = n.
1 subgoal

  ============================
   forall n : nat, add n 0 = n

add_02 < induction n.
2 subgoals

  ============================
   add 0 0 = 0

subgoal 2 is:
 add (S n) 0 = S n

add_02 < simpl.
2 subgoals

  ============================
   0 = 0
```

```
subgoal 2 is:
 add (S n) 0 = S n

add_02 < reflexivity.
1 subgoal

  n : nat
  IHn : add n 0 = n
  ============================
   add (S n) 0 = S n

add_02 < simpl.
1 subgoal

  n : nat
  IHn : add n 0 = n
  ============================
   S (add n 0) = S n

add_02 < rewrite IHn.
1 subgoal

  n : nat
  IHn : add n 0 = n
  ============================
   S n = S n

add_02 < reflexivity.
No more subgoals.

add_02 <
```

Commentaires :

- notez l'utilisation de la tactique `induction n` ;
- la tactique `rewrite` utilisée avec une égalité (désignée ici par le nom de l'hypothèse `IHn`) opère la *réécriture* d'un terme de l'égalité par l'autre. Par défaut, la réécriture s'applique dans la formule à montrer en utilisant l'égalité de gauche à droite. Nous verrons plus tard d'autres options de cette tactique.

On montre selon un schèma analogue

```
Lemma add_S2 : forall (m n:nat), (add m (S n)) = S (add m n).
```

(exercice)

Récurrence terminale la version plus proche de la définition en C est la suivante :

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

On montre que les fonctions `add` et `add_t` sont égales ; c'est-à-dire qu'appliquées aux mêmes arguments, elles prennent même valeur :

```
Coq < Lemma add_t_add : forall (n m:nat), (add_t n m)=(add n m).
1 subgoal

  ============================
   forall n m : nat, add_t n m = add n m

add_t_add < induction n.
2 subgoals

  ============================
   forall m : nat, add_t 0 m = add 0 m

subgoal 2 is:
 forall m : nat, add_t (S n) m = add (S n) m

add_t_add < trivial.
1 subgoal

  n : nat
  IHn : forall m : nat, add_t n m = add n m
  ============================
   forall m : nat, add_t (S n) m = add (S n) m

add_t_add < intro.
1 subgoal

  n : nat
  IHn : forall m : nat, add_t n m = add n m
  m : nat
  ============================
   add_t (S n) m = add (S n) m

add_t_add < simpl.
1 subgoal

  n : nat
  IHn : forall m : nat, add_t n m = add n m
  m : nat
  ============================
   add_t n (S m) = S (add n m)

add_t_add < rewrite IHn.
1 subgoal

  n : nat
  IHn : forall m : nat, add_t n m = add n m
  m : nat
  ============================
   add n (S m) = S (add n m)
```

```
add_t_add < apply add_S2.
No more subgoals.
```

Commentaires :

- notez comment on a résolu le *cas de base* de l'induction (lorsque `n` prend la valeur `0`) avec la tactique `trivial` : elle remplace ici la suite `intro. simpl. reflexivity.`
- notez que dans l'hypothèse d'induction, *la variable `m` est universellement quantifiée*. C'est ce qui permet d'utiliser cette hypothèse pour réécrire `(add_t n (S m))` en `(add n (S m))` : le `m` de l'hypothèse est instancié avec le `(S m)` de `(add_t n (S m))` ;
- pour conclure, on a utilisé la tactique `apply` plutôt qu'un `rewrite` car l'égalité à montrer est celle donnée par le lemme `add_S2`.

Autres définitions récursives Nous avons souligné (paragraphe «Type et terminaison» page 8) que la clause de définition `Fixpoint` applique un critère purement syntaxique pour s'assurer de la terminaison des fonctions définies. Cela interdit certaines formes de définitions récursives, pourtant arithmétiquement correctes comme :

```
Fixpoint add (n m:nat) : nat :=
  if (n =? 0) then m else (add (n-1) m).
```

Le système nous informe que

```
Error: Cannot guess decreasing argument of fix.
```

On peut comprendre cela car il est loin d'être immédiat de s'assurer que le `n-1` de l'appel récursif est un entier plus petit que le `n` d'origine : cela nécessite une *preuve* que

```
forall (n:nat), (not (n=0)) -> ((n-1) < n)
```

Par cas sur `n` :

- si `n`=0, alors la formule est trivialement vraie car la prémisse `(not (n=0))` est fausse ;
- sinon, `n` est de la forme `(S n)` et `(((S n) -1) < (S n))` se simplifie en `(n - 0) < (S n)`. Or `(n - 0) = n` (ce qui est un lemme à démotrer par récurrence sur `n`), il faut donc montrer que `n < (S n)`, ce qui demande d'utiliser la définition de `<` en terme de `<=` (`n < m` ssi `(S n) <= m`) et la réflexivité de la relation `<=`.

Nous verrons plus tard comment le système Coq fournit toutefois des outils pour poser des définitions récursives lorsque l'on sait donner les preuves de décroissance des arguments. Retenons simplement pour l'instant que l'on ne saura pas tout «programmer» avec la clause `Fixpoint`.