SU/FSI/MASTER/INFO/STL/MU5IN554
Spécification et Vérification de Programmes
Éléments d'une logique d'ordre supérieur avec le système Coq.

oct. 2020

1 Programmation récursive : récurrence structurelle

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

1.1 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 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*.

1.2 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`. 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) {
  for(; n > 0; n--) m++;
  return m;
}
```

en supposant que `n` est positif. Nous verrons une définition alternative de l'addition qui, quoique toujours fonctionnelle, est encore plus proche de cette définition en C.

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`.

Généralisons notre problème en notant $P($`n`$)$ pour `(add n 0) = n` et résumons la situation :

1. On sait montrer $P($`0`$)$.
2. On saurait montrer $P($`S n0`$)$ si l'on avait $P($`n0`$)$.

Nous avons là les deux ingrédients qui permettent de construire une règle de raisonnement valide pour les entiers naturelles : le *principe d'induction structurelle*. On peut le présenter ainsi : pour tote 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)`.

On peut justifier intuitivement ce principe en observant que si l'on a effectivement démontré (1) et (2), on saura montrer `(P 0)`, `(P (S 0))`, `(P (S (S 0)))`, `(P (S (S (S 0)))`, etc.

En effet, par (1), on a `(P 0)` ; de cela et de (2), on déduit `(P (S 0))` ; de cela et de (2), on déduit `(P (S (S 0)))` ; de clea et de (2), on déduit `(P (S (S (S 0))))`, etc.

Avec (1) et (2), on a donc un procédé pour montrer que pour tout valeur `v` de type `nat`, `(P v)`. En effet, par construction du type `nat`, pour toute expression `v:nat`, il exsite une expression de la forme `(S (S ... 0)...)` qui lui est égale. En itérant l'application de (2) autant de fois qu'il y a de `S` on finira toujours par avoir une démonstration de `(P (S (S ... 0)...))` ; toujours, car il y a un nombre fini de `S` dans l'écriture d'un entier. Ainsi, (1) et (2) suffisent à assurer la validité de (3) : `forall (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)) = 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`.

Propriété logique et fonction booléenne

L'ensemble des entiers naturels est muni d'une relation d'ordre (au sens large) que l'on définit souvent à l'aide de l'addition :

$$n \leq m \text{ si et seulement si il existe } k \text{ tel que } m = n + k$$

On pose cette définition dans le système Coq de la manière suivante :

```
Definition xle (n m:nat): Prop :=
  exists (k:nat), (m = (add n k)).
```

Commentaires

— on appelle `xle` notre relation d'ordre. Elle a 2 paramètres de type `nat` (c'est une relation *binaire* sur les entiers).

— l'expression `(xle 0 (S 0))` est de type `Prop`, qui est, dans le système Coq, le type des *valeurs de vérité logiques* – qu'il ne faut pas confondre avec le type des booléens.

Le type de `xle` elle-même est `nat -> nat -> Prop`. C'est, en quelques sorte, une *fonction de vérité*.

La relation `xle` est réflexive : pour tout `(n:nat)`, `(xle n n)`. En effet, si l'on *déplie* la définition de `xle`, il faut vérifier que *il existe* ***`(k:nat)`*** *tel que* ***`n = (add n k)`***. Ce `k` n'est pas difficile à trouver : il suffit de prendre `0`.

Transcrivons cette preuve dans le système Coq :

```
Coq < Lemma xle_refl : forall (n:nat), (xle n n).
1 subgoal

  ============================
   forall n : nat, xle n n

xle_refl < intro.
1 subgoal

  n : nat
  ============================
   xle n n

xle_refl < unfold xle.
1 subgoal

  n : nat
  ============================
   exists k : nat, n = add n k

xle_refl < exists 0.
1 subgoal

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

xle_refl < rewrite add_02.
1 subgoal

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

xle_refl < reflexivity.
No more subgoals.
```

Commentaires :

— la tactique `unfold` permet de *déplier* une définition ;

— la tactique `exists` permet d'indiquer quelle valeur prendre pour démontrer une formule existentielle ;

— notez l'utilisation de l'égalité `add_02`.

Exercice :

```
Lemma xle_0 : forall (n:nat), (xle 0 n).
```

Même squelette de preuve.

Intéressons nous maintenant à la propriété suivante : *pour tout* `(n m:nat)`, *si* `(xle n m)` *alors* `(xle (S n) (S m))`.

Pour établir cette propriété, on procède ainsi : supposons `(n:nat)`, `(m:nat)`, `H:(xle n m)` et montrons `(xle (S n) (S m))` ; c'est-à-dire, par définition : il existe `(k:nat)` tel que `(S m) = (add (S n) k)`. Par définition, l'hypothèse `H` nous donne qu'il existe `(k:nat)` tel que `m = (add n k)`. Appelons `x` le `k` que nous donne `H`. On a donc `(x:nat)` tel que `m = (add n x)`. Appelons `H0` l'hypothèse que `m = (add n x)`.
Pour montrer qu'il existe `(k:nat)` tel que `(S m) = (add (S n) k)`, il suffit de prendre ce `x` et montrer `(S m) = (add (S n) x)` ; c'est-à-dire, par évaluation : `(S m) = (S (add n x))`.
En utilisant l'hypothèse `H0`, cela revient à montrer que `(S (add n x)) = (S (add n x))` ; ce qui est donné par réflexivité de l'égalité.

Transcrivons cette preuve dans le système Coq :

```
Coq < Lemma xle_n_S : forall (n m:nat), (xle n m) -> (xle (S n) (S m)).
1 subgoal

  ============================
   forall n m : nat, xle n m -> xle (S n) (S m)

xle_n_S < unfold xle.
1 subgoal

  ============================
   forall n m : nat,
   (exists k : nat, m = add n k) -> exists k : nat, S m = add (S n) k

xle_n_S < intros.
1 subgoal

  n, m : nat
  H : exists k : nat, m = add n k
  ============================
   exists k : nat, S m = add (S n) k

xle_n_S < elim H.
1 subgoal

  n, m : nat
  H : exists k : nat, m = add n k
  ============================
   forall x : nat, m = add n x -> exists k : nat, S m = add (S n) k

xle_n_S < intros.
1 subgoal

  n, m : nat
  H : exists k : nat, m = add n k
```

```
  x : nat
  H0 : m = add n x
  ============================
   exists k : nat, S m = add (S n) k

xle_n_S < exists x.
1 subgoal

  n, m : nat
  H : exists k : nat, m = add n k
  x : nat
  H0 : m = add n x
  ============================
   S m = add (S n) x

xle_n_S < simpl.
1 subgoal

  n, m : nat
  H : exists k : nat, m = add n k
  x : nat
  H0 : m = add n x
  ============================
   S m = S (add n x)

xle_n_S < rewrite H0.
1 subgoal

  n, m : nat
  H : exists k : nat, m = add n k
  x : nat
  H0 : m = add n x
  ============================
   S (add n x) = S (add n x)

xle_n_S < reflexivity.
No more subgoals.
```

Commentaires :

- la tactique `intros` permet de répéter autant d'invocations de `intro` que possible ;
- pour «extraire» un *témoin* d'une hypothèse existentielle, on peut utiliser la tactique `elim`. Remarquez comment celle-ci modifie le but à prouver et comment nous enchaînons avec `intros` pour avoir nos nouvelles hypothèses.

Définition inductive de prédicat La définition que ne avons donnée de la relation d'ordre sur les entiers définit le symbole de prédicat binaire `xle` comme une *abréviation* d'une formule à deux variables libres. Le langage d'ordre supérieur du calcul des constructions nous autorise à la poser comme la définition d'une *fonction propositionnelle* de type `nat -> nat -> Prop`.

On peut aussi envisager une définition *axiomatique* de ce prédicat. C'est-à-dire que : on introduit un symbole pour désigner le prédicat (prenons `le`) ; on pose les propriétés logiques du symbole. Par exemple, en reprenant la définition donnée de la relation d'ordre dans la bibliothèque standard du système `Coq`, on pose les deux

énoncés suivants :

1. (`le` n n) pour tout entier n (la relation est *réflexive*).
2. Si (`le` n m) alors (`le` n $m+1$), pour tout entier n et m (la relation est close «à droite» pour la fonction successeur).

On peut facilement se convaincre que `le`, ainsi défini, contient tous les couples $(n, n+k)$, c'est-à-dire tous les couples d'entiers (n, m) tels que (`xle` n m). Nous le ferons formellement avec les lemmes `le_xle` et `xle_le`. On a donc bien caractérisé, avec nos deux énoncés, la relation d'ordre sur les entiers naturels.

Pour être complets, il faut rajouter une *clause de clôture* à notre définition qui fait de nos deux axiomes une condition nécessaire et suffisante pour être dans la relation `le`.

Techniquement, on peut poser ce type de définition dans le système `Coq` en utilisant une *définition inductive* similaire à celle que nus avions utilisée pour la type `nat`. De fait, la relation d'ordre sur les entiers est définie dans la bibliothèque standard du système Coq, de cette manière (module `Coq.Init.Peano`) :

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

On y retrouve bien nos deux «axiomes». Ils sont étiquetés par des symboles (`le_n` et `le_S`) que l'on appelle les *constructeurs* du prédicat `le`. Ceux-ci servent à construire les preuves de validité des expressions de la forme (`le` n_1 n_2) lorsque $n_1 \leq n_2$. Exemple :

```
Coq < Fact le_2_4 : (le 2 4).
1 subgoal

  ============================
   2 <= 4

le_2_4 < apply le_S.
1 subgoal

  ============================
   2 <= 3

le_2_4 < apply le_S.
1 subgoal

  ============================
   2 <= 2

le_2_4 < apply le_n.
No more subgoals.

le_2_4 < Qed.
apply le_S.
apply le_S.
apply le_n.

le_2_4 is defined
```

Commentaire :

— notez comment l'application du constructeur `le_S` avec la tactique `apply` engendre un nouveau but de preuve. Celui-ci est l'instance de la prémisse de l'implication qui définit le constructeur.

On a ici une réalisation dans le système Coq du principe de raisonnement par *modus ponens* : pour montrer B, si l'on sait $A \to B$, il suffit de montrer A. Dans notre exemple, A est `(le n m)` et B est `(le n (S m))`. La tactique `apply` est également capable de trouver les instances des variable `n` et `m` liées dans la formule qui définit `le_S`.

Ce qu'il est intéressant de regarder ici, c'est *la preuve* de l'énoncé `(le 2 4)` telle que nous l'affiche la commande `Print le_2_4` :

```
Coq < Print le_2_4.
le_2_4 = le_S 2 3 (le_S 2 2 (le_n 2))
     : 2 <= 4
```

Les noms `le_S` et `le_n` sont des *constructeurs* pour les preuves des énoncés de la forme `(le` n_1 n_2`)`. Ainsi se rejoignent le point de vue logique (axiomes) et le point de vue types inductifs (constructeurs). Les ≪valeurs≫ construites ici n'appartiennent pas à des ensembles de données (éléments du type `Set`) mais à des ensembles de *vérités* (éléments du type `Prop`). Le *terme* `(le_S 2 3 (le_S 2 2 (le_n 2)))` qui est de *type* `2 <= 4` est la représentation dans le système Coq de la *preuve* que `2 <= 4`.

Preuve par constructeurs Nous illustrons plus avant l'utilisation de ces constructeurs de prédicats, ainsi que l'utilisation de quelques résultats de la bibliothèque standard (`Coq.Arith.Le`) avec la preuve de :

```
Lemma xle_le : forall (n m:nat), (xle n m) -> (le n m).
```

Par induction sur `n`

- si `n`=0, il faut montrer `(le 0 m)`, ce qui nous est donné par le lemme de la bibliothèque standard `le_0_n`.
- si `n` est de la forme `(S n)`, on a comme hypothèse d'induction que (IHn) `forall (m:nat), (xle n m) -> (le n m)` et il faut montrer que si (H) `(xle (S n) m)` alors `(le (S n) m)`. Pour cela, nous procédons par cas sur `m` :
  - si `m`=0, il faut montrer que si `(xle (S n) 0)` alors `(le (S n) 0)`. Ce que l'on a, en vertu du principe *ex falso quodlibet* en montrant que l'hypothèse `(xle (S n) 0)` est contradictoire.
    En effet, si `(xle (S n) 0)` alors, il exist `k:nat` tel que `0 = (add (S n) k)`, c'est-à-dire, par évaluation symbolique, `0 = (S (add n k))`. C'est la contradiction recherchée.
  - si `m` est de la forme `S m`, on suppose (H)`(xle (S n) (S m))` et on montre `(le (S n) (S m))`. En utilisant le lemme `le_n_S`, reste à montrer que `(le n m)` ; en utilisant l'hypothèse de récurrence `IHn`, reste à montrer que `(xle n m)` ; en utilisant le lemme `xle_S_n` que nous avons démontré, reste à montrer que `(xle (S n) (S m))` ce qui nous est donné par l'hypothèse `H`.

Voici la transcription de cette preuve dans le système Coq :

```
Coq < Lemma xle_le : forall (n m:nat), (xle n m) -> (le n m).
1 subgoal

  ============================
   forall n m : nat, xle n m -> n <= m

xle_le < induction n.
2 subgoals

  ============================
   forall m : nat, xle 0 m -> 0 <= m

subgoal 2 is:
 forall m : nat, xle (S n) m -> S n <= m
```

```
xle_le < intros.
2 subgoals

  m : nat
  H : xle 0 m
  ============================
   0 <= m

subgoal 2 is:
 forall m : nat, xle (S n) m -> S n <= m

xle_le < apply le_0_n.
1 subgoal

  n : nat
  IHn : forall m : nat, xle n m -> n <= m
  ============================
   forall m : nat, xle (S n) m -> S n <= m

xle_le < destruct m.
2 subgoals

  n : nat
  IHn : forall m : nat, xle n m -> n <= m
  ============================
   xle (S n) 0 -> S n <= 0

subgoal 2 is:
 xle (S n) (S m) -> S n <= S m

xle_le < intro.
2 subgoals

  n : nat
  IHn : forall m : nat, xle n m -> n <= m
  H : xle (S n) 0
  ============================
   S n <= 0

subgoal 2 is:
 xle (S n) (S m) -> S n <= S m

xle_le < elim H. intros.
2 subgoals

  n : nat
  IHn : forall m : nat, xle n m -> n <= m
  H : xle (S n) 0
  ============================
   forall x : nat, 0 = add (S n) x -> S n <= 0
```

```
subgoal 2 is:
 xle (S n) (S m) -> S n <= S m

2 subgoals

  n : nat
  IHn : forall m : nat, xle n m -> n <= m
  H : xle (S n) 0
  x : nat
  H0 : 0 = add (S n) x
  ============================
   S n <= 0

subgoal 2 is:
 xle (S n) (S m) -> S n <= S m

xle_le < simpl in H0.
2 subgoals

  n : nat
  IHn : forall m : nat, xle n m -> n <= m
  H : xle (S n) 0
  x : nat
  H0 : 0 = S (add n x)
  ============================
   S n <= 0

subgoal 2 is:
 xle (S n) (S m) -> S n <= S m

xle_le < discriminate.
1 subgoal

  n : nat
  IHn : forall m : nat, xle n m -> n <= m
  m : nat
  ============================
   xle (S n) (S m) -> S n <= S m

xle_le < intro.
1 subgoal

  n : nat
  IHn : forall m : nat, xle n m -> n <= m
  m : nat
  H : xle (S n) (S m)
  ============================
   S n <= S m

xle_le < apply le_n_S.
1 subgoal
```

```
  n : nat
  IHn : forall m : nat, xle n m -> n <= m
  m : nat
  H : xle (S n) (S m)
  ============================
   n <= m

xle_le < apply IHn.
1 subgoal

  n : nat
  IHn : forall m : nat, xle n m -> n <= m
  m : nat
  H : xle (S n) (S m)
  ============================
   xle n m

xle_le < apply xle_S_n.
1 subgoal

  n : nat
  IHn : forall m : nat, xle n m -> n <= m
  m : nat
  H : xle (S n) (S m)
  ============================
   xle (S n) (S m)

xle_le < assumption.
No more subgoals.
```

Commentaires :

- les lemmes `le_0_n`, `le_n_S` et `O_S` sont donnés par la bibliothèque standard. Nous avons démontré le lemme `xle_S_n` ;
- la suite `elim H. intros.` nous donne le `x` contenu dans l'hypothèse `H`. La tactique `elim` a su déplier la définition de `xle` ;
- la tactique `discriminate` réalise, dans le système Coq le principe du *ex falso ...* qui permet de déduire ce que l'on veut d'une assertion fausse. Ici l'assertion fausse est l'hypothèse `H0` qui pose l'égalité entre `0` et `S (n + x).` ;
- notez comment la tactique `simpl in H0` permet d'appliquer le mécanisme d'évaluation à une hypothèse ;
- notez l'utilisation de `assumption` lorsque le but à prouver nous est donné directement par hypothèse.

Relation inductive et principe d'induction Comme pour le type des entiers, la relation `le` admet un principe d'induction. Intuitivement, si l'on a que `(le n m)` alors, par définition de `le`, on peut, et il suffit, d'envisager deux cas :

- ou bien `n`=`m`, ce qui correspond au premier «axiome» de la définition – constructeur `le_n` ;
- ou bien `m` est en fait de la forme `(S m)` et l'on peut supposer que `(xle n m)`, ce qui correspond à «l'axiome» `le_S`.

Nous allons voir l'utilisation de ce principe en démontrant la réciproque du lemme ci-dessus.

Supposons donc que `(le n m)` et montrons `(xle n m)`. Par *induction* sur `(le n m)` :

— il faut montrer `(xle n n)`. On peut utiliser ici le lemme `xle_refl` que nous avons démontré ;
— on suppose `(xle n m)` et il faut montrer que `(xle n (S m))` ; c'est-à-dire qu'il existe `k:nat` tel que `(S m) = n+k`.
Par hypothèse, on a `x:nat` tel que `m = n + x`. Il suffit alors de prendre `(S x)` pour valeur de `k`. Il faut alors montrer que `(S m) = n + (S x)`. Ce que l'on ramène à `(S m) = (S (n + x))` en utilisant le lemme `add_S2` que nous avons démontré. Et cette égalité nous est donnée par l'hypothèse que `m = n + x`.

Voici comment cela se transcrit dans le système Coq :

```
Coq < Lemma le_xle : forall (n m:nat), (le n m) -> (xle n m).
1 subgoal

  ============================
   forall n m : nat, n <= m -> xle n m

le_xle < intros.
1 subgoal

  n, m : nat
  H : n <= m
  ============================
   xle n m

le_xle < elim H.
2 subgoals

  n, m : nat
  H : n <= m
  ============================
   xle n n

subgoal 2 is:
 forall m0 : nat, n <= m0 -> xle n m0 -> xle n (S m0)

le_xle < apply xle_refl.
1 subgoal

  n, m : nat
  H : n <= m
  ============================
   forall m0 : nat, n <= m0 -> xle n m0 -> xle n (S m0)

le_xle < intros. unfold xle.
1 subgoal

  n, m : nat
  H : n <= m
  m0 : nat
  H0 : n <= m0
  H1 : xle n m0
  ============================
   xle n (S m0)
```

```
1 subgoal

  n, m : nat
  H : n <= m
  m0 : nat
  H0 : n <= m0
  H1 : xle n m0
  ============================
   exists k : nat, S m0 = add n k

le_xle < elim H1. intros.
1 subgoal

  n, m : nat
  H : n <= m
  m0 : nat
  H0 : n <= m0
  H1 : xle n m0
  ============================
   forall x : nat, m0 = add n x -> exists k : nat, S m0 = add n k

1 subgoal

  n, m : nat
  H : n <= m
  m0 : nat
  H0 : n <= m0
  H1 : xle n m0
  x : nat
  H2 : m0 = add n x
  ============================
   exists k : nat, S m0 = add n k

le_xle < exists (S x).
1 subgoal

  n, m : nat
  H : n <= m
  m0 : nat
  H0 : n <= m0
  H1 : xle n m0
  x : nat
  H2 : m0 = add n x
  ============================
   S m0 = add n (S x)

le_xle < rewrite add_S2.
1 subgoal

  n, m : nat
```

```
  H : n <= m
  m0 : nat
  H0 : n <= m0
  H1 : xle n m0
  x : nat
  H2 : m0 = add n x
  ============================
   S m0 = S (add n x)

le_xle < rewrite H2.
1 subgoal

  n, m : nat
  H : n <= m
  m0 : nat
  H0 : n <= m0
  H1 : xle n m0
  x : nat
  H2 : m0 = add n x
  ============================
   S (add n x) = S (add n x)

le_xle < reflexivity.
No more subgoals.
```

Commentaires :

— notez comment, pour raisonner par «induction» sur l'hypothèse `H:(le n m)`, nous avons utilisé la tactique `elim`.

Fonction booléenne Certaines fonctions utilisent l'ordre sur les entiers dans leurs calculs. La fonction d'insertion dans une liste ordonnée, par exemple. Il leur faut donc l'équivalent booléen de la relation d'ordre. c'est-à-dire,

> une fonction qui, étant donnés deux entiers n et m, donne le booléen `true` si $n \leq m$ et le booléen `false` sinon.

Une manière d'effectuer ce calcul est de décrémenter parallèlement n et m jusqu'à ce que l'un d'eux atteigne 0. On regarde alors lequel de n ou m a été annulé : si c'est n, c'est que $n \leq m$, sinon, c'est que $n \not\leq m$.

La définition récursive de ce calcul s'écrit ainsi :

```
Fixpoint leb (n m:nat) : bool :=
  match n,m with
      0, _ => true
    | _, 0 => false
    | (S n), (S m) => (leb n m)
  end.
```

Commentaires :

— notez le filtrage sur le couple `n,m` et l'utilisation du *motif anonyme* `_`.

Nous allons montrer que cette définition est *correcte vis-à-vis de sa spécification*. C'est-à-dire que si `(leb n m)=true` alors `(le n m)`.

La preuve s'obtient très facilement par induction sur `n` et par cas sur `m`. D'autant plus que nous allons utiliser les deux lemmes déjà vus : `le_0_n` et `le_n_S`.

Sans en faire un schéma intuitif préalable, donnons la preuve de notre résultat avec le système Coq :

```
Coq < Lemma leb_le : forall (n m:nat), (leb n m)=true -> (le n m).
1 subgoal

  ============================
   forall n m : nat, leb n m = true -> n <= m

leb_le < induction n.
2 subgoals

  ============================
   forall m : nat, leb 0 m = true -> 0 <= m

subgoal 2 is:
 forall m : nat, leb (S n) m = true -> S n <= m

leb_le < intros.
2 subgoals

  m : nat
  H : leb 0 m = true
  ============================
   0 <= m

subgoal 2 is:
 forall m : nat, leb (S n) m = true -> S n <= m

leb_le < apply le_0_n.
1 subgoal

  n : nat
  IHn : forall m : nat, leb n m = true -> n <= m
  ============================
   forall m : nat, leb (S n) m = true -> S n <= m

leb_le < destruct m.
2 subgoals

  n : nat
  IHn : forall m : nat, leb n m = true -> n <= m
  ============================
   leb (S n) 0 = true -> S n <= 0

subgoal 2 is:
 leb (S n) (S m) = true -> S n <= S m

leb_le < simpl.
2 subgoals

  n : nat
  IHn : forall m : nat, leb n m = true -> n <= m
```

```
  ============================
   false = true -> S n <= 0

subgoal 2 is:
 leb (S n) (S m) = true -> S n <= S m

leb_le < intro.
2 subgoals

  n : nat
  IHn : forall m : nat, leb n m = true -> n <= m
  H : false = true
  ============================
   S n <= 0

subgoal 2 is:
 leb (S n) (S m) = true -> S n <= S m

leb_le < discriminate H.
1 subgoal

  n : nat
  IHn : forall m : nat, leb n m = true -> n <= m
  m : nat
  ============================
   leb (S n) (S m) = true -> S n <= S m

leb_le < intro.
1 subgoal

  n : nat
  IHn : forall m : nat, leb n m = true -> n <= m
  m : nat
  H : leb (S n) (S m) = true
  ============================
   S n <= S m

leb_le < apply le_n_S.
1 subgoal

  n : nat
  IHn : forall m : nat, leb n m = true -> n <= m
  m : nat
  H : leb (S n) (S m) = true
  ============================
   n <= m

leb_le < apply IHn.
1 subgoal

  n : nat
```

```
  IHn : forall m : nat, leb n m = true -> n <= m
  m : nat
  H : leb (S n) (S m) = true
  ============================
   leb n m = true

leb_le < assumption.
No more subgoals.
```

Commentaire :

— dans la dernière utilisation de `assumption`, le système identifie de lui-même le but `(leb n m)=true` avec l'hypothèse `(leb (S n) (S m))=true` par évaluation symbolique.

Propriété de complétude, inversion de prédicat inductif Ce premier énoncé nous donne la *correction* de la fonction `leb` vis-à-vis de la spécification `le`. Cela signifie que la fonction `leb` ne fait pas de calcul erroné. On peut, et il est bon, de vérifier la proposition inverse : si `(le n m)` alors `(leb n m)=true`. Cette proposition énonce la *complétude* de la fonction vis-à-vis de sa spécification.

La preuve de cet énoncé va être l'occasion d'introduire une autre utilisation des définitions inductives de prédicats : *l'inversion*. Donnons dans un premier temps la version ≪papier-crayon≫ de la preuve de : pour tout `(n m:nat)`, si `(le n m)` alors `(leb n m)=true`.

ATTENTION : il y a une finesse technique dans cette preuve, aussi allons nous utiliser l'écriture formalisée des énoncés à démontrer. Ici : `forall (n m:nat), (le n m) -> (leb n m)=true`. C'est-à-dire : `forall (n:nat), forall (m:nat), (le n m) -> (leb n m)=true`. Formellement donc, nous montrons par induction sur `n:nat` la formule `forall (m:nat), (le n m) -> (leb n m)=true` :

— si `n=0`, il faut montrer `(leb 0 m)` sous hypothèse que `(le 0 m)`. Ce qui est trivial par évaluation.

— si `m` a la forme `(S m)`, notre hypothèse d'induction est `IHn:forall (m:nat), (le n m) -> (leb n m)=true` et il faut montrer que `forall (m:nat), (le (S n) m) -> (leb (S n) m)=true`. Supposons donc `H:(le (S n) m)` et montrons `(leb (S n) m)=true`.
  Par définition de `le`, de l'hypothèse `H:(le (S n) m)` on peut déduire que :
  — soit `(le (S n) m)` est justifié par le constructeur `le_n` et alors `m=(S n)` ;
  — soit `(le (S n))` est justifié par le constructeur `le_S`, auquel cas, `m=(S m0)` et `(le (S n) m0)`.
  Appliquons ce principe à la démonstration de `(leb (S n) m)=true` :
  — si `m=(S n)`, il faut montrer que `(leb (S n) (S n))`. C'est-à-dire, après évaluation symbolique, que `(leb n n)=true`. On peut appliquer l'hypothèse d'induction qui donne à montrer que `(le n n)`. Ce que l'on a par `le_n`.
  — si `m=(S m0)` avec `(le (S n) m0)`, il faut montrer que `(leb (S n) (S m0))=true`. C'est-à-dire, après évaluation symbolique : `(leb n m0)`. Par application de l'hypothèse d'induction, il reste à montrer que `(le n m0)` ; puis par application du lemme `(le_S_n)`, il reste à montrer que `(le (S n) (S m0))`. Ce qui est l'hypothèse `H` où l'on peut remplacer `m` par `(S m0)`, puisqu'ici, `m=(S m0)`.

Il faut noter comment il a été utilie ici que l'hypothèse d'induction soit universellement quantifiée sur `m:nat` car dans chacune des ces 2 utilisations, nous avons instancié cette variable avec des termes différents.

Transcription de la preuve avec le système Coq :

```
Coq < Lemma le_leb : forall (n m:nat), (le n m) -> (leb n m)=true.
1 subgoal

  ============================
   forall n m : nat, n <= m -> leb n m = true

le_leb < induction n.
```

```
2 subgoals

  ============================
   forall m : nat, 0 <= m -> leb 0 m = true

subgoal 2 is:
 forall m : nat, S n <= m -> leb (S n) m = true

le_leb < auto.
1 subgoal

  n : nat
  IHn : forall m : nat, n <= m -> leb n m = true
  ============================
   forall m : nat, S n <= m -> leb (S n) m = true

le_leb < intros.
1 subgoal

  n : nat
  IHn : forall m : nat, n <= m -> leb n m = true
  m : nat
  H : S n <= m
  ============================
   leb (S n) m = true

le_leb < inversion H.
2 subgoals

  n : nat
  IHn : forall m : nat, n <= m -> leb n m = true
  m : nat
  H : S n <= m
  H0 : S n = m
  ============================
   leb (S n) (S n) = true

subgoal 2 is:
 leb (S n) (S m0) = true

le_leb < apply IHn.
2 subgoals

  n : nat
  IHn : forall m : nat, n <= m -> leb n m = true
  m : nat
  H : S n <= m
  H0 : S n = m
  ============================
   n <= n
```

```
subgoal 2 is:
 leb (S n) (S m0) = true

le_leb < apply le_n.
1 subgoal

  n : nat
  IHn : forall m : nat, n <= m -> leb n m = true
  m : nat
  H : S n <= m
  m0 : nat
  H0 : S n <= m0
  H1 : S m0 = m
  ============================
   leb (S n) (S m0) = true

le_leb < apply IHn.
1 subgoal

  n : nat
  IHn : forall m : nat, n <= m -> leb n m = true
  m : nat
  H : S n <= m
  m0 : nat
  H0 : S n <= m0
  H1 : S m0 = m
  ============================
   n <= m0

le_leb < apply le_S_n.
1 subgoal

  n : nat
  IHn : forall m : nat, n <= m -> leb n m = true
  m : nat
  H : S n <= m
  m0 : nat
  H0 : S n <= m0
  H1 : S m0 = m
  ============================
   S n <= S m0

le_leb < rewrite H1.
1 subgoal

  n : nat
  IHn : forall m : nat, n <= m -> leb n m = true
  m : nat
  H : S n <= m
  m0 : nat
  H0 : S n <= m0
```

```
  H1 : S m0 = m
  ============================
   S n <= m

le_leb < assumption.
No more subgoals.
```

Commentaires :

- la tactique `auto` est un peu plus puissante que la tactique `trivial` ;
- vous aurez noté l'utilisation de `inversion H` où `H` est l'hypothèse `(S n) <= m` qui donne un but où `m = (S n)` et l'autre où `m = (S m0)` et `(S n) <= m0`.

L'arithmétique formelle

La définition du type inductif `nat` fournit la possibilité de définir des fonctions et de raisonner par récurrence structurelle sur les entiers. Cette simple base suffit, en théorie, à reconstruire l'ensemble de l'arithmétique. Toutefois certaines fonctions ne se prêtent pas facilement à une définition par récurrence structurelle avec la clause `Fixpoint`. C'est le cas de la division entière (le *quotient*).

En effet, l'algorithme qui permet d'obtenir le quotient de n par m consiste à compter le nombre de fois où l'on peut retrancher m de n. En C, on écrirait

```
int div(int n, int m)
  int q;
  for (q = 0; n > 0; n = n - m, q++);
  return q;
```

Ce que l'on transcrirait

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

Mais cette définition souffre d'un défaut : il n'est pas possible de montrer que la valeur de `(div n m)` est définie pour tout `(n m:nat)`. En effet, `(div n 0)` n'est pas définie – puique `n-0=n` et ≪la fonction boucle≫. Quand bien même nous tenterions d'éviter ce cas en posant

```
Fixpoint div (n m:nat) : nat :=
  match n, m with
    0, _ => 0
  | _, 0 => 0
  | (S n), (S m) => S (div (n - m) m)
  end.
```

Le système nous refusera cette définition au prétexte que *Error : Cannot guess decreasing argument of fix* : il ne sait pas reconnaître la décroissance du premier paramètre de `div` dans l'appel récursif. Par tant, il n'est pas capable d'assurer que la fonction `div` *termine*. Or, c'est une exigence du système Coq que toutes les fonctions que l'on y définit terminent. Nous reviendrons ultérieurement sur ce point.

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.
```

Dans notre définition, le paramètre `A` du type `list` est n'importe quel type de type `Set` (comme les booléens ou les entiers). Ainsi, l'expression `(cons true nil)` est de type `(list bool)` ; l'expression `(cons 1 nil)` est de type `(list nat)` ; etc.

Stricto sensu les constructeurs `nil` et `cons` sont, respectivement, de type `forall (A:Set), (list A)` et `forall (A:Set), A -> (list A) -> (list A)`. Ce qui implique que, par exemple, on distingue la liste de booléens vide de la liste d'entiers vide en écrivant `(nil bool)` dans le premier cas et `(nil nat)` dans le second. Toutefois, dans un cas aussi simple que celui des paramètres des listes, le système est en gégnéral capable *d'inférer* la valeur du paramètre de type des listes. Aussi, le système Coq est-il muni d'un mécanisme de *paramètres implicites* qui permet de dispenser l'utilisateur de mentionner le type des éléments d'une liste dans l'application des constructeurs mais également lors de la définition de fonctions manipulant des listes. Nous verrons l'utilisation de ce mécanisme pour définir des fonctions ainsi que d'autres structures de données paramétéres, comme les arbres binniares.

Définition récursives et induction structurelle La définition inductive du type des listes ouvre la possibilité de raisonner et de définir des fonctions sur les listes par récurrence structurelle.

Par exemple, 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 => 0
  | (cons x xs) => (S (length xs))
  end.
```

Dans la définition de la fonction, l'argument implicite est indiqué entre accolades. Il n'est plus besoin de le mentionner lorsque que l'on applique la fonction : on écrit simplement `(length xs)` là où il eût fallu écrire `(length A xs)`.

Par analogie avec le type `nat` on peut dire que `nil` est le 0 des listes et le constructeur `cons` sa fonction successeur.

Par analogie, la fonction de *concaténation* des listes est sa fonction d'addition. Comme en témoigne sa définition récursive :

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

On va retrouver, par exemple que `nil` est élément neutre, à gauche et à droite pour la concaténation. On a immédiatement :

```
Coq < Fact app_nil : forall (A:Set) (xs:list A), (app nil xs) = xs.
1 subgoal
```

```
  ============================
   forall (A : Set) (xs : list A), app nil xs = xs

app_nil < trivial.
No more subgoals.
```

En revanche, la neutralité à droite – `(app xs nil)=xs` – ne vient pas immédiatement et nécessite une *preuve par induction structurelle* sur les listes. Le principe d'induction structurelle pour les listes est le suivant : pour tout prédicat `P : (list A) -> Prop`, avec `(A:Set)` :

1. si `(P nil)`
2. si `(P xs)` entraîne `(P (cons x xs))`, pour tout `x:A` et `(xs:list A)`
3. alors, pour tout `xs:list A`, `(P xs)`

Appliquons ce principe pour montrer `(app xs nil)=xs`, avec `A:Set` et `xs:list A` :

- si `xs` est `nil`, il faut montrer `(app nil nil)=nil`, ce qui est immédiat ;
- si `xs` est de la forme `(cons x xs)`, avec `x:A` et `xs:list A`, on suppose, par hypothèse de récurrence, que `(app xs nil)=xs` et il faut montrer que `(app (cons x xs) nil)=(cons x xs)`. Par évaluation symbolique, cela revient à montrer que `(cons x (app xs nil))=(cons x xs)`. Par hypothèse de récurrence, on peut remplacer `(app xs nil)` par `xs`. Reste à montrer que `(cons x xs)=(cons x xs)`, ce qui est trivial.

Transcrivons cette preuve dans le système Coq :

```
Coq < Lemma app_nil2 : forall (A:Set) (xs:list A), (app xs nil)=xs.
1 subgoal

  ============================
   forall (A : Set) (xs : list A), app xs nil = xs

app_nil2 < induction xs.
2 subgoals

  A : Set
  ============================
   app nil nil = nil

subgoal 2 is:
 app (a :: xs) nil = (a :: xs)%list

app_nil2 < trivial.
1 subgoal

  A : Set
  a : A
  xs : list A
  IHxs : app xs nil = xs
  ============================
   app (a :: xs) nil = (a :: xs)%list

app_nil2 < simpl.
1 subgoal

  A : Set
```

```
  a : A
  xs : list A
  IHxs : app xs nil = xs
  ============================
   (a :: app xs nil)%list = (a :: xs)%list

app_nil2 < rewrite IHxs.
1 subgoal

  A : Set
  a : A
  xs : list A
  IHxs : app xs nil = xs
  ============================
   (a :: xs)%list = (a :: xs)%list

app_nil2 < reflexivity.
No more subgoals.
```

Commentaires :

- notez la variante syntaxique : `(a :: xs)%list` plutôt que `(cons a xs)` ;
- comme avec les entiers (type `nat`) on applique aux listes le principe d'induction structurelle avec la tactique `induction`.

La concaténation de listes, comme l'addition, est une opération associative. Mais l'analogie s'arrête là : la concaténation, contrairement à l'addition, n'est pas commutative. Par exemple, on n'a pas, l'équivalent de `add_S2`, à savoir : `(app xs (cons y ys))=(cons y (app xs ys))`.

Fonctions partielles

On peut utiliser les structures de listes pour *modéliser* une structure séquentielle de tableaux. Pour cela, on voudra définir l'accès à un élément du tableau par son indice. C'est-à-dire, par sa position dans la liste. On sait toutefois que cette fonction n'est pas partout définie. par exemple, en Java : `IndexOutOfBoundsException`. Or, dans le système Coq, il nous faut toujours des fonctions totales.

Une astuce consiste, dans notre cas et dans bien d'autres, à encapsuler le résultat attendu, dans un type qui, intuitivement dit «il n'y a pas de valeur» ou «il y a une valeur et la voici». C'est le type paramétré `option` prédéfini en Coq :

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

Commentaire :

- notez que les valeurs du type `option` ne sont pas simplement des `Set`, mais des `Type`. On peut avoir des listes de booléens alors que `bool` est un `Set`, car dans la téhorie des type du système Coq, `Set` est un sous-type de `Type`.
- le type `A` est implicite.

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 mais 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.
```

Remarque :

— la réponse du système Coq à cette définition est : `nth is recursively defined (decreasing on 2nd argument)`. Cela a pour conséquence que, dans le processus d'évaluation symbolique, le second argument (la liste) est privilégié par rapport au premier (l'indice). En fait, le système Coq a interprété notre définition par double filtrage comme l'imbrication de filtrage suivante :

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

Relations logiques et fonctions propositionnelles, égalité

La relation d'appartenance d'un élément à une liste est d'usage courant. Un élément `x` appartient à la liste, non vide, `xs` si et seulement si `x` est le premier élément de `xs` ou si `x` appartient à la suite de `xs`.

On peut transcrire cette définition par la relation inductive :

```
Inductive In {A:Set} : A -> (list A) -> Prop :=
  mem_hd : forall (x:A) (xs:list A), (In x (cons x xs))
| mem_tl : forall (x y:A), (In x xs) -> (In x (cons y xs)).
```

Une alternative qu'offre le système Coq pour définir des prédicats inductifs est d'adopter un style proche de la définition de fonctions. En effet, le système Coq permet de produire aussi aisément les valeurs booléennes que les valeurs propositionnelles. Ainsi, on peut définir le prédicat logique d'appartenance comme on le ferait de la fonction booléenne (récursive) qui teste l'appartenance. C'est d'ailleurs cette définition que l'on trouve dans la bibliothèque standard du système Coq (module `List`) :

```
Fixpoint In (a:A) (l:list A) : Prop :=
    match l with
      | [] => False
      | b :: m => b = a \/  In a m
    end.
```

Égalité La *fonction propositionnelle* `In` utilise *l'égalité propositionnelle* prédéfinie du système Coq. On obtient ainsi une définition *polymorphe* : le type `A` reste indéterminé.

Si l'on veut définir la *fonction booléenne* qui teste l'appartenance, il faut résoudre une petite difficulté : le test d'égalité entre valeurs. En effet, il n'existe pas de fonction booléenne testant l'égalité pour tous les types de données. Il faut définir ces fonctions au cas par cas. Il y a une excellente raison à cela, c'est que, en toute généralité, *l'égalité est indécidable* : il n'existe pas d'algorithme capable de calculer, pour tout type de valeur, si deux valeurs (d'un même type) sont égales ou non.

On peut toutefois contourner cette impossibilité en définissant une fonction qui prend en paramètre une fonction booléenne.

```
Fixpoint mem A:Set (eqb:A -> A -> bool) (x:A) (xs:list A) : bool :=
  match xs with
    nil => false
  | (cons y xs) => (orb (eqb x y) (mem eqb x xs))
  end.
```

Naturellement, cette fonction donnera le résultat attendu à la condition que la valeur d'appel de `eqb` soit une fonction de test d'égalité telle que `(eqb x y)=true <-> (eq x y)`.

1.4 Propositions, formules et fonctions

Étudions la preuve de la proposition qui établit que si un élément appartient à la concaténation de deux listes, alors il appartient à une liste ou l'autre (ce résultat est connu de la bibliothèque standard sous le nom de `in_app_or`) :

```
Lemma in_app_or :
  forall (A:Set) (z:A) (xs ys:list A),
    (In z (app xs ys)) -> (In z xs) \/  (In z ys).
```

Par induction sur `xs` :

- si `xs` est la liste vide `nil`, il faut montrer, après simplification, que `(In z ys) -> (In z nil) \/ (In z ys)`. Ce qui est une tautologie du calcul de proposition ;
- si `xs` a la forme `(cons x xs)`, sous l'hypothèse d'induction que `(In z (app xs ys)) -> (In z xs) \/ (In z ys)`, il faut montrer que si (H) `(In z (app (cons x xs) ys))` alors `(In z (cons x xs)) \/ (In z ys)`. Par simplification, cela revient à montrer que `(z=x \/ (In x xs)) \/ (In x ys)`. Pour montrer une disjonction, il suffit de montrer l'un ou l'autre de ses termes.
  Par hypothèse (H), après simplification, on a que `z=x \/ (In z (app xs ys))`. Ceci nous permet de *raisonner pas cas* selon que `x=z` ou `(In z (app xs ys))`.
  - dans le premier cas, on a terminé ;
  - dans le second cas, de l'hypothèse d'induction, on peut déduire que `(In z xs)` ou `(In z ys)` et l'on a égalemnt terminé.

Nous allons maintenant formaliser cette preuve avec le système Coq et nous verrons, en particulier comment raisonner par cas selon les termes d'une disjonction ainsi qu'une manière synthétique de réaliser dans le système l'étape de preuve : ≪*de l'hypothèse d'induction, on peut déduire que* ***(In z xs)*** *ou* ***(In z ys)***≫.

```
Coq < Lemma in_app_or :
  forall (A:Set) (z:A) (xs ys:list A),
    (In z (app xs ys)) -> (In z xs) \/  (In z ys).
Coq < Coq < 1 subgoal

  ============================
   forall (A : Set) (z : A) (xs ys : list A),
   In z (xs ++ ys) -> In z xs \/  In z ys

in_app_or < induction xs.
2 subgoals

  A : Set
  z : A
  ============================
```

```
   forall ys : list A, In z (nil ++ ys) -> In z nil \/  In z ys

subgoal 2 is:
 forall ys : list A, In z ((a :: xs) ++ ys) -> In z (a :: xs) \/  In z ys

in_app_or < tauto.
1 subgoal

  A : Set
  z, a : A
  xs : list A
  IHxs : forall ys : list A, In z (xs ++ ys) -> In z xs \/  In z ys
  ============================
   forall ys : list A, In z ((a :: xs) ++ ys) -> In z (a :: xs) \/  In z ys

in_app_or < simpl.
1 subgoal

  A : Set
  z, a : A
  xs : list A
  IHxs : forall ys : list A, In z (xs ++ ys) -> In z xs \/  In z ys
  ============================
   forall ys : list A,
   a = z \/  In z (xs ++ ys) -> (a = z \/  In z xs) \/  In z ys

in_app_or < intros.
1 subgoal

  A : Set
  z, a : A
  xs : list A
  IHxs : forall ys : list A, In z (xs ++ ys) -> In z xs \/  In z ys
  ys : list A
  H : a = z \/  In z (xs ++ ys)
  ============================
   (a = z \/  In z xs) \/  In z ys

in_app_or < elim H.
2 subgoals

  A : Set
  z, a : A
  xs : list A
  IHxs : forall ys : list A, In z (xs ++ ys) -> In z xs \/  In z ys
  ys : list A
  H : a = z \/  In z (xs ++ ys)
  ============================
   a = z -> (a = z \/  In z xs) \/  In z ys

subgoal 2 is:
```

```
 In z (xs ++ ys) -> (a = z \/  In z xs) \/  In z ys

in_app_or < tauto.
1 subgoal

  A : Set
  z, a : A
  xs : list A
  IHxs : forall ys : list A, In z (xs ++ ys) -> In z xs \/  In z ys
  ys : list A
  H : a = z \/  In z (xs ++ ys)
  ============================
   In z (xs ++ ys) -> (a = z \/  In z xs) \/  In z ys

in_app_or < intro.
1 subgoal

  A : Set
  z, a : A
  xs : list A
  IHxs : forall ys : list A, In z (xs ++ ys) -> In z xs \/  In z ys
  ys : list A
  H : a = z \/  In z (xs ++ ys)
  H0 : In z (xs ++ ys)
  ============================
   (a = z \/  In z xs) \/  In z ys

in_app_or < elim (IHxs ys H0).
2 subgoals

  A : Set
  z, a : A
  xs : list A
  IHxs : forall ys : list A, In z (xs ++ ys) -> In z xs \/  In z ys
  ys : list A
  H : a = z \/  In z (xs ++ ys)
  H0 : In z (xs ++ ys)
  ============================
   In z xs -> (a = z \/  In z xs) \/  In z ys

subgoal 2 is:
 In z ys -> (a = z \/  In z xs) \/  In z ys

in_app_or < tauto.
1 subgoal

  A : Set
  z, a : A
  xs : list A
  IHxs : forall ys : list A, In z (xs ++ ys) -> In z xs \/  In z ys
  ys : list A
```

```
  H : a = z \/  In z (xs ++ ys)
  H0 : In z (xs ++ ys)
  ============================
   In z ys -> (a = z \/  In z xs) \/  In z ys

in_app_or < tauto.
No more subgoals.
```

Commentaires :

- notez l'utilisation de la tactique `tauto` qui utilise une *procédure de décision* pour construire les preuves des tautologies du calcul des propositions ;
- nous avons utilisé par deux fois la tactique `elim` pour pouvoir raisonner par cas selon les termes d'une hypothèse nous donnant une disjonction :
  - la première fois, nous avons utilisé la tactique `elim H` sur l'hypothèse `H : (a = z) \/ (In z (app xs ys))` qui est directement une disjonction ;
  - la seconde fois, nous avons utilisé la tactique `elim` avec l'hypothèse d'induction `IHxs` dont la disjonction n'est pas directement accessible. Toutefois, on peut *déduire* cette disjonction en *appliquant* l'hypothèse `IHxs` à `ys: list A` et à l'hypothèse `H0 : (In z (app xs ys))`.
    Dans un système comme Coq, l'hypothèse `IHxs` est comme une fonction *de type* `forall ys : list A, (In z (app xs ys)) -> (In z xs) \/ (In z ys)`. Si on lui applique une liste (ici, `ys`), l'expression `(IHxs ys)` est quelque chose *de type* `(In z (app xs ys)) -> (In z xs) \/ (In z ys)` ; et si on applique à cela l'hypothèse `H0` qui est de type `(In z (app xs ys))`, l'expression `(IHxs ys H0)` nous donne la disjonction `(In z xs) \/ (In z ys)`.
    En fait, l'expression applicative `(IHxs ys H0)` est, dans le système Coq (et dans ce contexte), *une preuve* de la formule `(In z xs) \/ (In z ys)`. De manière générale, dans le système Coq, les preuves sont des fonctions dont le *type* est la formule qu'elles démontrent.

2 L'art de la spécification

Considérons la fonction suivante :

```
Fixpoint get_prefix {A:Set} (n:nat) (xs:list A) :=
  match n, xs with
    0, _ => nil
  | (S n), nil => nil
  | (S n), (cons x xs) => (cons x (get_prefix n xs))
  end.
```

Comme son nom le suggère, cette fonction calcule un préfixe de `xs`. On attend de surcroît que le résultat soit une liste de longueur `n`. Ce que l'on pourrait exprimer par la formule :

```
forall (A:Set) (n:nat) (xs:list A), (length (get_prefixe n xs)) = n.
```

Toutefois, cette formule n'est pas vraie en général. Par exemple, il est faux que `(length (get_prefixe 1 nil)) = 1` ; puisque `(get_prefixe 1 nil)` est égal à `nil` et `(length nil)` est égal à `0`.

L'égalité de la longueur du résultat et de l'argument `n` n'est vraie que si `n` n'est pas plus grand que la longueur de la liste dont on veut extraire le préfixe (*i.e.* `n` est inférieur ou égal à la longueur de la liste). Il y a donc une (pré)condition à satisfaire pour obtenir l'égalité attendue :

```
forall (A:Set) (n:nat) (xs:list A),
  (n <= (length xs)) -> (length (get_prefix n xs)) = n.
```

La propriété de longueur n'est bien entendue pas suffisante pour garantir le correction de la fonction `get_prefix`. Il faut également garantir que la valeur de l'application `(get_prefix n xs)` calcule effectivement un préfixe de la liste `xs`. Se pose alors la question de *formaliser* l'énoncé ≪*`(get_prefix n xs)` est un préfixe de `xs`*≫.

Nous allons envisager trois possibilités de définition de cette relation, correspondant à trois visions différentes des structures de listes. Nous montrerons l'équivalence de ces trois définitions. Nous prouverons ensuite la correction de la fonction `get_prefix`.

2.1 La relation ≪est préfixe de≫

Informellement, une liste `zs` est un préfixe de la liste `xs` si `xs` ≪commence≫ par `zs`. Reste à traduire formellement sur les listes la notion de ≪commence par≫.

Les listes comme suites indicées Une liste est une *séquence* de valeurs dans laquelle chacune a une position, comme dans les structures de tableau. La fonction `nth` (définie en 1.3), nous donne, lorsqu'elle existe, la valeur présente à une certaine position dans une liste : `(nth i xs)` désigne, lorsqu'elle existe, la valeur en position `i` dans `xs`.

Avec cette vision, dire que `xs` commence par `zs`, c'est dire que pour tout indice `i` dans la liste `zs`, `(nth i zs) = (nth i xs)`. On garantit que `i` est ≪un indice dans la liste `zs`≫ en posant la condition que `i < (length zs)`. Ces éléments nous donnent une première définition de la relation cherchée :

```
Definition Is_prefix A:Set (zs xs:list A) : Prop :=
  forall (i:nat), (i < length zs) -> (nth i zs) = (nth i xs).
```

Cette définition est cependant incomplète : si elle garantit (par la prémisse `(i < (length zs))`) que `(nth i zs)` est bien défini, elle ignore le cas où `(nth i xs)` ne l'est pas. Pour obtenir une définition correcte, il faut poser :

```
Definition Is_prefix1 {A:Set} (zs xs:list A) : Prop :=
  (length zs <= length xs) /\
  forall (i:nat), (i < length zs) -> (nth i zs) = (nth i xs).
```

Si elle est correcte, cette définition est, en fait, d'un emploi assez pénible pour raisonner avec les structures inductives de listes. Nous le verrons lorsque nous montrerons que cette première définition implique la troisième.

Le *monoïde* des listes[1] Il existe une vision des suite de valeurs, et donc des listes, dans laquelle c'est l'opération de concaténation qui est primitive.

Dans cette perspective, dire que `xs` ≪commence≫ par `zs`, c'est dire que `xs` est construite en concaténant `zs` avec une liste `ys`. En d'autre termes, c'est dire qu'il existe une liste `ys` telle que `xs` est égale à la concaténation de `zs` et `ys`. On peut écrire (formellement) cette égalité : `(app zs ys) = xs`. La *relation* ≪`zs` est un préfixe de `xs`≫, s'écrit alors à l'aide d'un quantificateur existentiel : `exists (ys:list A), (app zs ys) = xs`. Et nous pouvons poser comme définition de la relation ≪être préfixe de≫ :

```
Definition Is_prefix2 {A:Set} (zs xs : list A) : Prop :=
  exists (ys:list A), (app zs ys) = xs.
```

Cette formulation est assez limpide et plus aisée d'utilisation que la précédente bien que le traitement de l'existentiel reste un peu délicat.

1. Structure algébrique munie d'une loi de composition interne associative et qui possède un élément neutre.

Relation inductive Cette autre manière de définir la relation `Is_prefix` repose sur la définition inductive des listes.

Pour que `zs` soit un préfixe de `xs`, il suffit de vérifier l'une ou l'autre des deux conditions suivante :

1. soit `zs` est la liste vide `nil`.
2. soit `zs` et `xs` commencent par une même valeur et `zs` se prolonge en un préfixe de `xs`.

C'est le second terme de la conjonction de la seconde condition fait de cette définition une *définition inductive*.

On peut préciser plus formellement cette seconde condition : l'énoncé «`zs` et `xs` commencent par une même valeur» signifie que l'on peut écrire `zs` sous la forme `(cons a zs)` et `xs` sous la forme `(cons a xs)`, pour un `a` quelconque ; l'énoncé «`zs` se prolonge en un préfixe de `xs`» devint alors simplement : `zs` est un préfixe de `xs`. Plus formellement : pour toute valeur `a` et toutes listes `zs` et `xs`, si `zs` est un préfixe de `xs` alors `(cons a zs)` est un préfixe de `(cons a xs)`.

Ce qui s'exprime dans le système Coq par la définition inductive suivante :

```
Inductive Is_prefix {A:Set} : (list A) -> (list A) -> Prop :=
  is_refix_nil : forall (xs:list A), (Is_prefix nil xs)
| is_refix_cons : forall (a:A) (zs xs:list A),
    (Is_prefix zs xs) -> (Is_prefix (cons a zs) (cons a xs)).
```

2.2 Équivalence des définitions

Pour montrer l'équivalence de ces trois définitions, nous montrons leur implication circulaire. C'est-à-dire que `Is_prefix1` implique `Is_prefix2` qui implique `Is_prefix` qui implique à son tour `Is_prefix1`. Ainsi, chacune des définitions implique, directement ou par transitivité, les deux autres.

`Is_prefix1` **implique** `Is_prefix2` Soit à montrer que pour toutes listes `zs` et `xs`, si `(Is_prefix1 zs xs)` alors `(Is_prefix2 zs xs)`.

Par induction sur `zs` :

— si `zs=nil`, il faut montrer qu'il existe `ys` telle que `(app nil ys)=xs`. Il suffit de prendre `xs` pour valeur de `ys`.

— si `zs` est de la forme `(cons z zs)`, notre hypothèse d'induction est (HR) : pour toute liste `xs`, si `(Is_prefix1 zs xs)` alors `(Is_prefix2 zs xs)`. Il faut montrer que pour toute liste `xs`, si `(Is_prefix1 (cons z zs) xs)` alors `(Is_prefix2 (cons z zs) xs)`.

On raisonne par cas sur `xs` :

— si `xs=nil`, il faut montrer que si `(Is_prefix1 (cons z zs) nil)` alors `(Is_prefix2 (cons z zs))`. Ce que l'on obtient *ex falso* puisque `(Is_prefix1 (cons z zs) nil)` est faux car `(length (cons z zs)) <= 0` est faux.

— si `xs` a la forme `(cons x xs)`, on suppose (H) `(Is_prefix1 (cons z zs) (cons x xs))` – c'est-à-dire (H1) `(length (cons z zs)) <= (length (cons x xs))` et (H2) pour tout `i <= (length (cons z zs))`, `(nth i (cons z xs))=(nth i (cons x xs))` – et montrons qu'il existe `ys` tel que `(app (cons z zs) ys)=(cons x xs)`.

Avant de poursuivre, établissons deux propriétés de `Is_prefix1` :

(1) si `(Is_prefix1 (cons z zs) (cons x xs))` alors `z=x`.

En effet, par définition, si `(Is_prefix1 (cons z zs) (cons x xs))`, alors pour tout `i <= (length (cons z zs))`, `(nth i (cons z zs))=(nth i (cons x xs))`. En particulier, puisque `0 <= (length (cons z zs))`, on obtient `(nth 0 (cons z zs))=(nth 0 (cons x xs))`, c'est-à-dire : `z=x`.

(2) si `(Is_prefix1 (cons z zs) (cons x xs))` alors `(Is_prefix1 zs xs)`.
Si `(Is_Prefix1 (cons z zs) (cons x xs))` alors (Ha) `(length (cons z zs)) <= (length (cons x xs))` et (Hb) pour tout `i <= (length (cons z zs))`, `(nth i (cons z zs))=(nth i (cons x xs))`. Sous ces hypothèses, il faut montrer que `(Is_prefix1 zs xs)`, c'est-à-dire que (a) `(length zs) <= (length xs)` et que, (b) pour tout `i <= (length zs)` quelconque, `(nth i zs)=(nth i xs)`.
On obtient (a) de l'hypothèse (Ha).
Pour obtenir (b), on se donne un `i` tel que `i <= (length zs)`. En instanciant le `i` de `Hb`, par `(S i)`, on obtient que si `(S i) <= (length (cons z zs))` alors `(nth (S i) (cons z zs))=(nth (S i) (cons x xs))` ; ce qui, par évaluation symbolique donne que si `i <= (length zs)` alors `(nth i zs)=(nth i xs)`. Or on a supposé `i <= (length zs)` donc `(nth i zs)=(nth i xs)`. Ce qui achève la démonstration de (2).
Par (1), nous avons que `z=x`, il nous faut donc montrer qu'il existe `ys` tel que `(app (cons x zs) ys)=(cons x xs)`, c'est-à-dire : `(cons x (app zs ys))=(cons x xs)`. Il suffit pour cela de trouver un `ys` tel que `(app zs ys)=xs`.
De l'hypothèse (H) et de (2), on déduit que `(Is_prefix1 zs xs)` ; de cela et de (HR), on déduit qu'il existe un `ys` tel que `(app zs ys)=xs` qui est ce que nous voulions.

`Is_prefix2` **implique** `Is_prefix` Soit à montrer que pour toutes listes `zs` et `xs`, si `(Is_prefix2 zs xs)` alors `(Is_prefix zs xs)`.
Par induction sur `zs` :
— si `zs=nil`, on a `(Is_prefix nil xs)` par `is_prefix_nil`.
— si `zs` a la forme `(cons z zs)`, on suppose (HR) pour tout `xs`, si `(Is_prefix2 zs xs)` alors `(Is_prefix zs xs)` et on montre que si `(Is_prefix2 (cons z zs) xs)` alors `(Is_prefix (cons z zs) xs)` par cas sur `xs` :
  — si `xs=nil`, on montre que `(Is_prefix2 (cons z zs))` est faux car `(cons z (app zs ys))=nil` est faux.
  — si `xs` a la forme `(cons x xs)`, on suppose (H) `(Is_prefix2 (cons z zs) (cons x xs))` et on montre `(Is_prefix (cons z zs) (cons x xs))`.
  (1) De (H), on déduit que `z=x`. En effet, (H) nous donne un `ys` tel que `(app (cons z zs) ys)=(cons x xs)`, c'est-à-dire `(cons z (app zs ys))=(cons x xs)` et, par injectivité de `cons`, on a que `z=x` (et `(app zs ys)=xs`).
  (2) De (H) on déduit que `(Is_prefix2 zs xs)`. En effet, nous venons de voir comment (H) nous donne un `ys` tel que `(app zs ys)=xs` ; c'est-à-dire qu'il existe `ys` tel que `(app zs ys)=xs` ; c'est-à-dire `(Is_prefix2 zs xs)`.
  Par (1), il suffit de montrer que `(Is_Prefix (cons x zs) (cons x xs))`. Par application de `is_prefix_cons`, reste à montrer que `(Is_prefix zs xs)`. Ce que l'on obtient par (HR) et (2).
Notez que dans les deux preuves précédentes nous avons eu besoin de l'égalité des premiers éléments des listes ainsi que du fait que la relation ≪préfixe≫ passe aux suites de listes.

`Is_prefix` **implique** `Is_prefix1` Soit à montrer que pour toutes listes `zs` et `xs`, si `(Is_prefix zs xs)` alors `(Is_prefix1 zs xs)`.
Par induction sur `(Is_prefix zs xs)` :
— si `(Is_prefix nil xs)`, il faut montrer `(Is_prefix1 nil xs)`, c'est-à-dire (a) `0 <= (length xs)` et (b) pour tout `i < 0`, `(nth i nil)=(nth i xs)`.
  (a) est un théorème arithmétique.
  (b) est vrai car `i < 0` est faux.
— si `(Is_prefix (cons a zs) (cons a xs))`, supposons (HR) `(Is_prefix1 zs xs)` et montrons `(Is_prefix1 (cons a zs) (cons a xs))` ; c'est-à-dire (a) `(length (cons a zs)) <= (length (cons a xs))` et (b) pour tout `i < (length (cons z zs))`, `(nth i (cons a zs))=(nth i (cons a xs))`.

En développant `Is_prefix1` dans (HR), on a (Ha) `(length zs) <= (length xs)` et (Hb) pour tout `i < (length zs)`, `(nth i zs)=(nth i xs)`.
(a) est donné par (Ha).
(b) est donné par cas sur `i`
— si `i=0`, il faut montrer que `(nth 0 (cons a zs))=(nth 0 (cons a xs))`, ce qui est immédiat.
— si `i` est de la forme `(S i)`, on suppose `(S i) < (length (cons a zs))` et il faut montrer que `(nth (S i) (cons a zs))=(nth (S i) (cons a xs))` ; c'est-à-dire `(nth i zs)=(nth i xs)`. Ce que nous donne (Hb) sachant que `(S i) < (length (cons a zs))` nous donne que `i < (length zs)`.

Transcription en Coq

Si l'on analyse les trois preuves ci-dessus, on peut voir qu'elles utilisent toutes peu ou prou quatre propriétés de la relation «est préfixe de» :

1. `nil` est préfixe de toute liste.
2. `(cons z zs)` n'est pas préfixe de `nil`.
3. si `(cons z zs)` est préfixe de `(cons x xs)` alors `z=x`.
4. si `(cons z zs)` est préfixe de `(cons x xs)` alors `zs` est préfixe de `xs`.

Nous n'allons pas reprendre la totalité des preuves d'équivalence pour les réaliser dans le système Coq, mais nous allons montrer comment prouver les trois propriétés mentionnées ci-dessus.

Propriété 1 On obtient que `nil` est préfixe de toute liste avec `Is_prefix1` car

1. `0` (qui est la longueur de `nil`) est le plus petit des entiers.
2. il n'y a pas d'indice inférieur (strictement) à `0`.

```
Coq < Lemma prefix1_nil :
Coq <   forall (A:Set)(xs:list A), (Is_prefix1 nil xs).
1 subgoal

  ============================
   forall (A : Set) (xs : list A), Is_prefix1 nil xs

prefix1_nil < intros.
1 subgoal

  A : Set
  xs : list A
  ============================
   Is_prefix1 nil xs

prefix1_nil < split.
2 subgoals

  A : Set
  xs : list A
  ============================
   length nil <= length xs

subgoal 2 is:
```

```
 forall i : nat, i < length nil -> nth i nil = nth i xs

prefix1_nil < simpl.
2 subgoals

  A : Set
  xs : list A
  ============================
   0 <= length xs

subgoal 2 is:
 forall i : nat, i < length nil -> nth i nil = nth i xs

prefix1_nil < auto with arith.
1 subgoal

  A : Set
  xs : list A
  ============================
   forall i : nat, i < length nil -> nth i nil = nth i xs

prefix1_nil < intros.
1 subgoal

  A : Set
  xs : list A
  i : nat
  H : i < length nil
  ============================
   nth i nil = nth i xs

prefix1_nil < inversion H.
No more subgoals.
```

Commentaires :

- la tactique `split` décompose en 2 sous-buts les termes d'un but qui est une conjonction.
- la tactique `auto` peut être paramétrée par un ensemble de résultats particuliers. ici on l'utilise avec une base de faits arithmétiques : `auto with arith`. Pour avoir accès à cette base, nous avons exécuté la commande : `Require Import Arith.`
- notez comment on conclut la preuve avec `inversion H` où `H: i <= (length nil)`, c'est-à-dire, par définition, `(S i) <= 0`.
  Nous avons vu comment la tactique d'inversion, comme son nom l'indique, nous a servi à déduire les conditions suffisantes d'un cas de prédicat inductif (preuve de `le_leb`). Nous en voyons ici une autre application : elle permet de montrer l'absurdité d'un cas de prédicat inductif, ici, `(S i) <= 0`. Ce cas n'est jamais réalisé car, dans la définition inductive de `<=`, aucune clause n'a pour conclusion une formule de la forme `(S i) <= 0`. Ce qui exclut, par définition, les couples d'entiers de la forme `(S i, 0)` de la relation `<=`.

On obtient que `nil` est préfixe de toute liste, au sens de `Is_prefix2` car `nil` est neutre pour la concaténation.

```
Coq < Lemma prefix2_nil :
  forall (A:Set)(xs:list A), (Is_prefix2 nil xs).
Coq < 1 subgoal
```

```
  ============================
   forall (A : Set) (xs : list A), Is_prefix2 nil xs

prefix2_nil < intros.
1 subgoal

  A : Set
  xs : list A
  ============================
   Is_prefix2 nil xs

prefix2_nil < unfold Is_prefix2.
1 subgoal

  A : Set
  xs : list A
  ============================
   exists ys : list A, app nil ys = xs

prefix2_nil < exists xs.
1 subgoal

  A : Set
  xs : list A
  ============================
   app nil xs = xs

prefix2_nil < trivial.
No more subgoals.
```

Enfin, le résultat cherché pour `Is_prefix` est simplement un cas de sa définition.

```
Coq < Lemma prefix_nil :
  forall (A:Set)(xs:list A), (Is_prefix nil xs).
Coq < 1 subgoal

  ============================
   forall (A : Set) (xs : list A), Is_prefix nil xs

prefix_nil < intros.
1 subgoal

  A : Set
  xs : list A
  ============================
   Is_prefix nil xs

prefix_nil < apply is_prefix_nil.
No more subgoals.
```

Propriété 2 Dans le cas de `Is_prefix1`, la contradiction vient du premier terme de la conjonction qui définit `Is_prefix1` : on n'a jamais `(S n) <= 0`.

```
Coq < Lemma not_prefix1_nil :
  forall (A:Set) (x:A) (xs:list A), not (Is_prefix1 (cons x xs) nil).
Coq < 1 subgoal

  ============================
   forall (A : Set) (x : A) (xs : list A), ~ Is_prefix1 (x :: xs) nil

not_prefix1_nil < intros.
1 subgoal

  A : Set
  x : A
  xs : list A
  ============================
   ~ Is_prefix1 (x :: xs) nil

not_prefix1_nil < intro.
1 subgoal

  A : Set
  x : A
  xs : list A
  H : Is_prefix1 (x :: xs) nil
  ============================
   False

not_prefix1_nil < elim H.
1 subgoal

  A : Set
  x : A
  xs : list A
  H : Is_prefix1 (x :: xs) nil
  ============================
   length (x :: xs) <= length nil ->
   (forall i : nat, i < length (x :: xs) -> nth i (x :: xs) = nth i nil) ->
   False

not_prefix1_nil < intros.
1 subgoal

  A : Set
  x : A
  xs : list A
  H : Is_prefix1 (x :: xs) nil
  H0 : length (x :: xs) <= length nil
  H1 : forall i : nat, i < length (x :: xs) -> nth i (x :: xs) = nth i nil
  ============================
   False
```

```
not_prefix1_nil < inversion H0.
No more subgoals.
```

Commentaires :

— En Coq, la négation `not F` est définie commme l'implication `F -> False`.
Dans notre script, les premiers `intros` fait passer les quantifications en hypothèse ; le deuxiéme `intro` fait passer la formule niée en hypothèse est donne `False` comme but de preuve. C'est une instance du raisonnement pas l'absurde : pour montrer `not F`, on suppose `F` et on montre `False`.

Dans le cas `Is_prefix2`, la contradiction vient de l'égalité supposée entre `(cons x (app xs ys))` et `nil` :

```
Coq < Lemma not_prefix2_nil :
  forall (A:Set) (x:A) (xs:list A), not (Is_prefix2 (cons x xs) nil).
Coq < 1 subgoal

  ============================
   forall (A : Set) (x : A) (xs : list A), ~ Is_prefix2 (x :: xs) nil

not_prefix2_nil < intros.
1 subgoal

  A : Set
  x : A
  xs : list A
  ============================
   ~ Is_prefix2 (x :: xs) nil

not_prefix2_nil < intro.
1 subgoal

  A : Set
  x : A
  xs : list A
  H : Is_prefix2 (x :: xs) nil
  ============================
   False

not_prefix2_nil < elim H.
1 subgoal

  A : Set
  x : A
  xs : list A
  H : Is_prefix2 (x :: xs) nil
  ============================
   forall x0 : list A, app (x :: xs) x0 = nil -> False

not_prefix2_nil < intros.
1 subgoal

  A : Set
  x : A
```

```
  xs : list A
  H : Is_prefix2 (x :: xs) nil
  x0 : list A
  H0 : app (x :: xs) x0 = nil
  ============================
   False

not_prefix2_nil < discriminate H0.
No more subgoals.
```

Commentaire :
— C'est `discriminate` qui permet de conclure ici.

Dans le cas de la définition inductive `Is_prefix`, la preuve est quasi immédiate avec l'inversion.

```
Coq < Lemma not_prefix_nil :
  forall (A:Set) (x:A) (xs:list A), not (Is_prefix (cons x xs) nil).
Coq < 1 subgoal

  ============================
   forall (A : Set) (x : A) (xs : list A), ~ Is_prefix (x :: xs) nil

not_prefix_nil < intros.
1 subgoal

  A : Set
  x : A
  xs : list A
  ============================
   ~ Is_prefix (x :: xs) nil

not_prefix_nil < intro.
1 subgoal

  A : Set
  x : A
  xs : list A
  H : Is_prefix (x :: xs) nil
  ============================
   False

not_prefix_nil < inversion H.
No more subgoals.
```

Propriété 3 Dans le cas de `Is_prefix1`, l'égalité des éléments de têtes est donnée par l'égalité en position 0.

```
Coq < Lemma prefix1_hd :
  forall (A:Set) (z x:A) (zs xs:list A),
    (Is_prefix1 (cons z zs) (cons x xs)) -> (z=x).
Coq < 1 subgoal
```

```
  ============================
   forall (A : Set) (z x : A) (zs xs : list A),
   Is_prefix1 (z :: zs) (x :: xs) -> z = x

prefix1_hd < intros.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  ============================
   z = x

prefix1_hd < elim H.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  ============================
   length (z :: zs) <= length (x :: xs) ->
   (forall i : nat, i < length (z :: zs) -> nth i (z :: zs) = nth i (x :: xs)) ->
   z = x

prefix1_hd < intros.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : length (z :: zs) <= length (x :: xs)
  H1 : forall i : nat,
       i < length (z :: zs) -> nth i (z :: zs) = nth i (x :: xs)
  ============================
   z = x

prefix1_hd < specialize H1 with (i:=0).
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : length (z :: zs) <= length (x :: xs)
  H1 : 0 < length (z :: zs) -> nth 0 (z :: zs) = nth 0 (x :: xs)
  ============================
   z = x
```

```
prefix1_hd < simpl in H1.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : length (z :: zs) <= length (x :: xs)
  H1 : 0 < S (length zs) -> Some z = Some x
  ============================
   z = x

prefix1_hd < injection H1.
2 subgoals

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : length (z :: zs) <= length (x :: xs)
  H1 : 0 < S (length zs) -> Some z = Some x
  ============================
   z = x -> z = x

subgoal 2 is:
 0 < S (length zs)

prefix1_hd < trivial.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : length (z :: zs) <= length (x :: xs)
  H1 : 0 < S (length zs) -> Some z = Some x
  ============================
   0 < S (length zs)

prefix1_hd < auto with arith.
No more subgoals.
```

Commentaire :

- l'égalité en position `0` est donnée par l'instanciation de `H1` avec `0` par la tactique `specialize` ;
- notez que `nth` ne donne par directement un élément de la liste, mais une valeur de type `option`. On récupère l'égalité des éléments qui sont argument du constructeur `Some` par propriété d'*injectivité des constructeurs*. On invoque cette propriété avec la tactique `injection`.

Dans le cas de `Is_prefix2`, on obtient l'égalité des éléments de tête par injectivité du constructeur de liste `cons` (après évaluation symbolique de `app`).

```
Coq < Lemma prefix2_hd :
  forall (A:Set) (z x:A) (zs xs:list A),
```

```
    (Is_prefix2 (cons z zs) (cons x xs)) -> (z=x).
Coq < 1 subgoal

  ============================
   forall (A : Set) (z x : A) (zs xs : list A),
   Is_prefix2 (z :: zs) (x :: xs) -> z = x

prefix2_hd < intros.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix2 (z :: zs) (x :: xs)
  ============================
   z = x

prefix2_hd < elim H.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix2 (z :: zs) (x :: xs)
  ============================
   forall x0 : list A, app (z :: zs) x0 = (x :: xs)%list -> z = x

prefix2_hd < intros.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix2 (z :: zs) (x :: xs)
  x0 : list A
  H0 : app (z :: zs) x0 = (x :: xs)%list
  ============================
   z = x

prefix2_hd < injection H0.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix2 (z :: zs) (x :: xs)
  x0 : list A
  H0 : app (z :: zs) x0 = (x :: xs)%list
  ============================
   app zs x0 = xs -> z = x -> z = x
```

```
prefix2_hd < trivial.
No more subgoals.
```

Dans le cas de la définition inductive de `Is_prefix`, l'égalité des éléments de tête est donnée, en quelque sorte, *par définition*.

```
Coq < Lemma prefix_hd :
  forall (A:Set) (z x:A) (zs xs:list A),
    (Is_prefix (cons z zs) (cons x xs)) -> (z=x).
Coq < 1 subgoal

  ============================
   forall (A : Set) (z x : A) (zs xs : list A),
   Is_prefix (z :: zs) (x :: xs) -> z = x

prefix_hd < intros.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix (z :: zs) (x :: xs)
  ============================
   z = x

prefix_hd < inversion H.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix (z :: zs) (x :: xs)
  a : A
  zs0, xs0 : list A
  H1 : Is_prefix zs xs
  H0 : a = z
  H2 : zs0 = zs
  H3 : z = x
  H4 : xs0 = xs
  ============================
   x = x

prefix_hd < trivial.
No more subgoals.
```

Commentaire :

— la tactique `inversion` force l'égalité de `x` et `z` puisque le seul cas applicable à `(Is_prefix (z :: zs) (x :: xs))` est `is_prefix_cons`.

Propriété 4 Dans le cas de `Is_prefix1`, on obtient la suite du préfixe par l'égalité des éléments en position supérieure à `0` dont l'indice s'écrit `(S i)`.

```
Coq < Lemma prefix1_tl :
  forall (A:Set) (z x:A) (zs xs:list A),
    (Is_prefix1 (cons z zs) (cons x xs)) -> (Is_prefix1 zs xs).
Coq < 1 subgoal

  ============================
   forall (A : Set) (z x : A) (zs xs : list A),
   Is_prefix1 (z :: zs) (x :: xs) -> Is_prefix1 zs xs

prefix1_tl < intros.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  ============================
   Is_prefix1 zs xs

prefix1_tl < elim H.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  ============================
   length (z :: zs) <= length (x :: xs) ->
   (forall i : nat, i < length (z :: zs) -> nth i (z :: zs) = nth i (x :: xs)) ->
   Is_prefix1 zs xs

prefix1_tl < intros.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : length (z :: zs) <= length (x :: xs)
  H1 : forall i : nat,
       i < length (z :: zs) -> nth i (z :: zs) = nth i (x :: xs)
  ============================
   Is_prefix1 zs xs

prefix1_tl < split.
2 subgoals

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
```

```
  H0 : length (z :: zs) <= length (x :: xs)
  H1 : forall i : nat,
       i < length (z :: zs) -> nth i (z :: zs) = nth i (x :: xs)
  ============================
   length zs <= length xs

subgoal 2 is:
 forall i : nat, i < length zs -> nth i zs = nth i xs

prefix1_tl < simpl in H0.
2 subgoals

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : S (length zs) <= S (length xs)
  H1 : forall i : nat,
       i < length (z :: zs) -> nth i (z :: zs) = nth i (x :: xs)
  ============================
   length zs <= length xs

subgoal 2 is:
 forall i : nat, i < length zs -> nth i zs = nth i xs

prefix1_tl < auto with arith.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : length (z :: zs) <= length (x :: xs)
  H1 : forall i : nat,
       i < length (z :: zs) -> nth i (z :: zs) = nth i (x :: xs)
  ============================
   forall i : nat, i < length zs -> nth i zs = nth i xs

prefix1_tl < intros.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : length (z :: zs) <= length (x :: xs)
  H1 : forall i : nat,
       i < length (z :: zs) -> nth i (z :: zs) = nth i (x :: xs)
  i : nat
  H2 : i < length zs
  ============================
```

```
   nth i zs = nth i xs

prefix1_tl < specialize H1 with (S i).
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : length (z :: zs) <= length (x :: xs)
  i : nat
  H1 : S i < length (z :: zs) -> nth (S i) (z :: zs) = nth (S i) (x :: xs)
  H2 : i < length zs
  ============================
   nth i zs = nth i xs

prefix1_tl < apply H1.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : length (z :: zs) <= length (x :: xs)
  i : nat
  H1 : S i < length (z :: zs) -> nth (S i) (z :: zs) = nth (S i) (x :: xs)
  H2 : i < length zs
  ============================
   S i < length (z :: zs)

prefix1_tl < simpl.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix1 (z :: zs) (x :: xs)
  H0 : length (z :: zs) <= length (x :: xs)
  i : nat
  H1 : S i < length (z :: zs) -> nth (S i) (z :: zs) = nth (S i) (x :: xs)
  H2 : i < length zs
  ============================
   S i < S (length zs)

prefix1_tl < auto with arith.
No more subgoals.
```

Commentaires :

— la tactique `elim` appliquée à une conjonction en hypothèse (ici `H : Is_prefix1 (z :: zs) (x :: xs)`) déconstruit la conjonction en ses deux termes. Notez comment on utilise `intros` pour ramener les termes de la conjonction dans les hypothèses ;

Dans le cas de `Is_prefix2`, la suite du préfixe est simplement donnée par injectivité du constructeur `cons` (après évaluation symbolique de `app`).

```
Coq < Lemma prefix2_tl :
  forall (A:Set) (z x:A) (zs xs:list A),
    (Is_prefix2 (cons z zs) (cons x xs)) -> (Is_prefix2 zs xs).
Coq < 1 subgoal

  ============================
   forall (A : Set) (z x : A) (zs xs : list A),
   Is_prefix2 (z :: zs) (x :: xs) -> Is_prefix2 zs xs

prefix2_tl < intros.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix2 (z :: zs) (x :: xs)
  ============================
   Is_prefix2 zs xs

prefix2_tl < elim H.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix2 (z :: zs) (x :: xs)
  ============================
   forall x0 : list A, app (z :: zs) x0 = (x :: xs)%list -> Is_prefix2 zs xs

prefix2_tl < intros ys H0.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix2 (z :: zs) (x :: xs)
  ys : list A
  H0 : app (z :: zs) ys = (x :: xs)%list
  ============================
   Is_prefix2 zs xs

prefix2_tl < exists ys.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix2 (z :: zs) (x :: xs)
  ys : list A
```

```
  H0 : app (z :: zs) ys = (x :: xs)%list
  ============================
   app zs ys = xs

prefix2_tl < injection H0.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix2 (z :: zs) (x :: xs)
  ys : list A
  H0 : app (z :: zs) ys = (x :: xs)%list
  ============================
   app zs ys = xs -> z = x -> app zs ys = xs

prefix2_tl < trivial.
No more subgoals.
```

Commentaire :

— notez comment dans `intros ys H0` on nomme les hypothèses introduites.

Le cas de `Is_prefix` est encore donné par définition.

```
Coq < Lemma prefix_tl :
  forall (A:Set) (z x:A) (zs xs:list A),
    (Is_prefix (cons z zs) (cons x xs)) -> (Is_prefix zs xs).
Coq < 1 subgoal

  ============================
   forall (A : Set) (z x : A) (zs xs : list A),
   Is_prefix (z :: zs) (x :: xs) -> Is_prefix zs xs

prefix_tl < intros.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix (z :: zs) (x :: xs)
  ============================
   Is_prefix zs xs

prefix_tl < inversion H.
1 subgoal

  A : Set
  z, x : A
  zs, xs : list A
  H : Is_prefix (z :: zs) (x :: xs)
  a : A
  zs0, xs0 : list A
  H1 : Is_prefix zs xs
```

```
  H0 : a = z
  H2 : zs0 = zs
  H3 : z = x
  H4 : xs0 = xs
  ============================
   Is_prefix zs xs

prefix_tl < assumption.
No more subgoals.
```

Remarques Les quatre propriétés utilisées dans nos preuves d'équivalences sont ou bien des cas de définition (pour la première) ou bien des énoncés (possiblement négatifs) d'inversion de la définition inductive. En atteste les preuves des propriété dans le cas `Is_prefix`.

Dans le système Coq, une définition inductive sera donc préférable, dans la mesure où elle fournit, par elle-même, les propriétés utiles du prédicat (par inversions) et la rend en cela plus maniable. Cela étant, dans le contexte de la spécification, il peut-être utile de poser une définition qui, si elle est moins pratique d'utilisation, sera plus intuitive. En particulier, saura recueillir l'assentiment du «client». C'est au vérificateur, technicien de la preuve, d'établir l'équivalence entre la définition acceptée par le client et sa version inductive plus maniable ; comme nous l'avons fait avec nos 3 versions de la relation «préfixe».

Une spécification dicte le comportement attendu d'un composant logiciel. Les méthodes formelles permettent de valider l'adéquation du programme (de la fonction) vis-à-vis de la spécification. Mais qui valide la spécification ? Pour prendre un exemple trivial, si la spécification contient une contradiction, n'importe quel programme saura la satisfaire.
Formuler des versions différentes d'une spécification et vérifier formellement leur équivalence est un moyen de réduire les risques d'une spécification vide ou lacunaire.

2.3 Preuve de programme

Possédant maintenant un programme (la fonction `get_prefix`) et sa spécification (le prédicat `Is_prefix`), nous sommes en mesure de vérifier (formellement) la *correction* du programme vis-à-vis de sa spécification.

Nous avons identifié deux propriétés attendues de la fonction `get_prefix`
- `(get_prexfix n xs)` calcule un préfixe de la liste `xs`
- `(get_prefix n xs)` calcule une liste de longueur `n` à la condition que `n` soit inférieur ou égal à la longueur de `xs`

Vérifier de correction de `get_prefix`, c'est démontrer le *théorème*

```
Theorem get_prefix_correct :
  forall (A:Set) (n:nat) (xs:list A),
    (le n (length xs))
    -> (Is_prefix (get_prefix n xs) xs) /\  (length (get_prefix n xs)) = n.
```

Comme il s'agit de démontrer une conjonction, on peut formuler et démontrer séparément les deux termes de la conjonction sous forme de deux lemmes :

```
Lemma Is_prefix_get_prefix :
  forall (A:Set) (n:nat) (xs:list A), (Is_prefix (get_prefix n xs) xs).
```

et

```
Lemma length_get_prefix :
  forall (A:Set) (n:nat) (xs:list A),
```

```
(le n (length xs)) -> (length (get_prefix n xs))=n.
```

Les preuves de ces deux lemmes viennent facilement par induction sur `n`, puis par cas sur `xs` lorsque `n` est un successeur. La preuve du théorème en découle immédiatement.

Complétude L'énoncé de correction garanti que la fonction ne produit pas de résultats erronés vis-à-vis de la correction ; elle produit toujours un préfixe. Toutefois, on peut exiger plus de la fonction, à savoir : est-elle capable de produire *tous les préfixes* d'une liste ?

Formellement, cela revient à poser la question : étant donné un préfixe `zs` de `xs`, existe-t-il un `n` tel que `(get_prefix n xs)=zs`. La réponse est «oui» car on montre

```
Theorem get_prefix_complete :
  forall (A:Set) (zs xs:list A),
    (Is_prefix zs xs) -> exists (n:nat), zs = (get_prefix n xs).
```

par induction sur `(Is_prefix zs xs)`.

3 Un algorithme de tri : spécification, définition, preuve

Étant donné une liste, on veut obtenir une seconde liste dont les éléments sont exactement ceux de la première, mais rangés selon un ordre déterminé. Un algorithme de tri est une manière de résoudre le passage d'une liste à l'autre.

La littérature abonde sur ce sujet. Nous allons étudier l'une des solutions possibles dans un cadre purement fonctionnel. Pour simplifier, nous nous limitons au problème de tri des listes d'entiers, mais la solution proposée est généralisable.

Nous devons nous pencher sur trois éléments :

1. *spécifier* la nature du résultat attendu. Pour le problème du tri, cette spécification doit définir deux choses : qu'est-ce qu'une liste triée ? qu'est-ce qu'une liste qui contient exactement les mêmes éléments qu'une autre liste ?
2. *programmer* la fonction de tri. Avec l'algorithme choisi, nous serons amenés à définir également la structure d'arbre binaire de recherche.
3. *vérifier* que le résultat de la fonction définie satisfait les éléments de la spécification. Il y aura donc deux propriétés à vérifier concernant nos fonctions : on obtient une liste ordonnée ; tous les éléments de la liste à trier sont bien présents dans le résulat.
   Si le schéma général de la preuve est assez évident, il nous faudra passer par l'énoncé et la preuve d'un bon nombre de résulats intermédiares.

3.1 Listes triées

Intuitivement, selon la vision des listes comme une séquence de valeurs, une liste est dite *triée*, selon un ordre donné sur les éléments de la liste, lorsque «*deux éléments placés l'un après l'autre dans la liste sont dans la relation d'ordre*» : si `y` est placé aprsè `x` alors `x` est plus petit que `y`. Cette définition intuitive doit être précisée pour obtenir une spécification formelle.

On peut préciser «deux éléments placés l'un après l'autre» en mentionnant leur position. On obtiendrait ici une spécification à la manière des suites indicées : *`xs` est triée si et seulement si pour tout indice `i` et `j` de `xs`, si `i` est inférieur à `j` alors `(nth i xs)` $\leq$ `(nth j xs)`*[2].

2. On oublie ici que `nth` donne une valeur de type `option` et non directement un élément de la liste.

On peut affiner cette définition en remarquant qu'il suffit d'assurer que l'ordre est respecté par toute paire d'éléments *consécutifs* de la liste – une relation d'ordre étant transitive, ce nouveau critère implique le précédent. Deux éléments sont consécutifs dans une liste lorsque leurs indices le sont. Deux indices sont consécutifs lorsque le second est successeur du premier. On pose alors que `(nth i xs)` $\leq$ `(nth (S i) xs)` en prenant soin que `i` et `(S i)` soient des indices de `xs`.

Nous avons toutefois observé, avec l'example de la relation *préfixe* que la vision ≪suite indicée≫ donne des définitions qu'il n'est pas aisé de manipuler avec le système Coq. On cherchera donc à exprimer la propriété ≪est triée≫ comme une relation inductive.

Un candidat assez naturel pour une telle définition est le suivant :

1. `nil` est triée.
2. si `xs` est triée et si `x` est inférieur à tout élément de `xs` alors `(cons x xs)` est triée.

Mais, ici encore, on peut affiner la définition en observant qu'il suffit, lorsque `xs` est supposée triée, d'assurer que `x` est inférieur au premier élément de `xs`. Il faut, bien entendu que `xs` ne soit pas vide. cela nous donne la défintion suivante :

1. `nil` est triée.
2. pour tout `x`, `(cons x nil)` est triée.
3. pour tout `x1`, `x2`, pour toute liste `xs`, si `(cons x2 xs)` et triée et si `x1` $\leq$ `x2` alors `(cons x1 (cons x2 xs))` est triée.

Ce qui se transcrit, en nous limitant, pour simplifier, aux listes d'entiers [3] :

```
Inductive Sorted : (list nat) -> Prop :=
  sorted_nil : (Sorted nil)
| sorted_cons1 : forall (x:nat), (Sorted (cons x nil))
| sorted_cons2 :
    forall (x1 x2:nat) (xs:list nat),
      (le x1 x2) -> (Sorted (cons x2 xs)) -> (Sorted (cons x1 (cons x2 xs))).
```

On retrouve la définition précédente sous forme de lemme :

```
Lemma sorted_cons :
  forall (x:nat) (xs:list nat),
    (Sorted xs)
    -> (forall (y:nat), (In y xs) -> (x <= y))
    -> (Sorted (cons x xs)).
```

Ce qui se prouve simplement par cas sur `xs`.

Voici une autre propriété des listes triées qui motive l'algorithme de tri que nous allons étudier : si deux listes sont triées et que les éléments de la première sont inférieurs à ceux de la seconde, alors la concaténation de la première et de la seconde liste est une liste triée. Formellement :

```
Lemma sorted_app :
  forall (xs ys:list nat),
    (Sorted xs) -> (Sorted ys)
    -> (forall (x:nat), (In x xs) -> (forall (y:nat), (In y ys) -> (le x y)))
    -> (Sorted (app xs ys)).
```

Preuve par induction sur `(Sorted xs)` :

— cas `sorted_nil` : dans ce cas, `xs` est la liste vide et `(app xs ys)` est égal à `ys` qui est triée par hypothèse ;

3. Cette définition est une instance de la définition plus générale `LocallySorted` que l'on trouve dans le module `Coq.Sorting.Sorted` de la bibliothèque standard.

— cas `sorted_cons1` : dans ce cas, `xs` a la forme `(cons x nil)` et `(app (cons x nil) ys)` est égal à `(cons x ys)`. En appliquant le lemme `sorted_cons`, reste à montrer que (1) `(Sorted ys)` et que (2) `forall (y:nat), (In y ys) -> (le x y)`.
(1) est donné par hypothèse ;
(2) est donné par l'hypothèse que tous les éléments de `xs` (c'est-à-dire, dans notre cas `(cons x nil)`) sont inférieurs à ceux de `ys`.
— cas `sorted_cons2` : dans ce cas, `xs` a la forme `(cons x1 (cons x2 xs))`. Il faut montrer que `(Sorted (app (cons x1 (cons x2 xs)) ys))`, c'est-à-dire que `(Sorted (cons x1 (cons x2 (app xs ys))))`.
Les hypothèses attachées à ce cas sont :
H1 `x1 <= x2`
H2 `(Sorted (cons x2 xs))`
HR si `(Sorted ys)` et si les éléments de `(cons x2 xs)` sont tous inférieurs à ceux de `ys` alors `(Sorted (app (cons x2 xs) ys))`
Pour montrer `(Sorted (cons x1 (cons x2 (app xs ys))))` on suppose également que
H2 `(Sorted ys)`
H3 tous les éléments de `(cons x1 (cons x2 xs))` sont inférieurs à ceux de `ys`
En appliquant `sorted_cons2`, reste à montrer que (1) `(x1 <= x2)` et que (2) `(Sorted cons x2 (app xs ys))`.
(1) nous est donné par hypothèse.
(2) en appliquant l'hypothèse d'induction HR, reste à montrer que (a) `(Sorted ys)` et (b) tous les éléments de `(conx x2 xs)` sont inférieurs à ceux de `ys`.
(a) est donné par hypothèse.
(b) on suppose un `x` quelconque élément de `(cons x2 xs)` et on montre que `(x <= y)` pour tout élément `y` de `ys`. En appliquant l'hypothèse H3, reste à montrer que (i) `x` est élément de `(cons x1 (cons x2 xs))` et que (ii) `y` est élément de `ys`.
(ii) nous est donné par hypothèse.
(i) se déduit de l'hypothèse que `x` est élément de `(cons x2 xs)`.

3.2 Permutation de liste

Pour définir ce qu'est une liste qui contient *exactement* les mêms éléments qu'une autre liste, il faut prendre garde à ce qu'une liste peut contenir *plusieurs occurences* d'un même éléments. Par exemple, les listes `[1]` et `[1;1]` contiennent le même éléments, mais pas *exactement*.

Cette précaution prise, on peut poser qu'une liste `xs` contient *exactement* les mêmes éléments qu'une liste `ys` si et seulement si le nombre d'occurrences d'un élément `z` dans `xs` est égal au nombre d'occurences de `z` dans `ys`. Avec cette définition, `[1]` et `[1;1]` ne contiennent pas *exactement* les mêmes éléments.

Le nombre d'occurences d'un élément dans une liste n'est pas difficile à définir, soit par une fonction récursive sur les listes, soit par une relation inductive. Optons pour une fonction `nbocc` telle que `(nbocc x xs)` donne le nombre d'occurences de `x` dans `xs` – possiblement nul. On peut alors définir que `xs` contient *exactement* les même élements que `ys` par : *pour tout élément `z`, `(nbocc z xs)=(nbocc z ys)`.*

Mais, on peut envisager le problème sous un angle d'abord plus pragmatique qui nous amènera à des notions plus théoriques.

Oublions un temps la notion de tri de liste pour celle de tri des éléments d'un tableau. Les programmes de tri sur les tableaux font en général appel à des *échangent de places* des éléments du tableau ; c'est-à-dire, à des *permutations* d'éléments dans le tableau. Ainsi le tableau trié est obtenu comme une permutation du tableau original. Si l'on revient aux listes, pour assurer que la liste triée contient *exactement* les mêmes éléments que la liste originale, on peut s'assurer que le résultat de la fonction de tri est une *permutation* de la liste à trier.

La relation ≪*xs est une permutation de ys*≫ est définie dans la bibliothèque standard du systhème Coq (`Coq.Sorting.Permutation`) par les quatre clauses inductives suivantes :

1. `nil` est une permutation de `nil`.
2. si `xs` est une permutation de `ys` alors, pour tout `z`, `(cons z xs)` est une permutation de `(cons z ys)`.
3. `(cons x1 (cons x2 xs))` est une permutation de `(cons x2 (cons x1 xs))`.
4. la relation est *transitive* : si `xs` est une permutation de `ys` et si `ys` est une permutatuion de `zs` alors `xs` est une permutation de `zs`.

Intuitivement, cette définition repose sur le fait que toute permutation d'une liste est obtenue par itération de la permutation de deux éléments en tête de liste : `(cons x1 (cons x2 xs))` est une permutation de `(cons x2 (cons x1 xs))`. La clause 2 permet d'opérer la permutation de deux éléments consécutifs de la liste, quelque soit leurs positions. la clause 4 permet, par permutation successives d'éléments consécutifs, d'obtenir la permutation d'éléments non consécutifs.

Nous adopterons cette définition inductive de la permutation [4].

3.3 Tri par arbre binaire de recherche

On peut réaliser, en style purement fonctionnel, une analogue de *Quick sort*. Plutôt que travailler en place sur la liste à trier, on passe par une structure intermédiaire d'*arbre binaire de recherche* pour en extraire la liste des éléments triés.

L'algorithme procède en deux phases :

1. Construire un arbre binaire de recherche contenant tous les éléments de la liste à trier.
2. Reconstituer une liste par parcours infixe le l'arbre construit.

La correction de l'algorithme repose sur

1. la propriété des arbres binaires de recherche de ranger les éléments insérés de manière à ce que les éléments stockés dans le sous-arbre gauche soient inférieurs à la racine et que celle-ci soit inférieure aux éléments stockés dans le sous-arbre droit (et ce, récursivement).
2. le lemme `sorted_app` démontré au paragraphe précédent.

Si `btree_of_list` est la fonction de construction de l'arbre binaire (de recherche) et `list_of_btree` la fonction d'extraction de la liste des éléments d'un arbre binaire (de recherche), la fonction de tri de la liste `xs` est définie par la composition `(list_of_btree (btree_of_list xs))`. On pose :

```
Definition bst_sort (xs:list nat) :=
  (list_of_btree (btree_of_list xs)).
```

La correction de la fonction repose sur deux résultats

1. `btree_of_list` construit un arbre binaire de recherche : pour toute liste (d'entiers) `xs`, `(btree_of_list xs)` est un arbre binaire de recherche.
2. `list_of_btree` extrait une liste triée d'un arbre binaire de recherche : pour tout arbre binaire `bt`, si `bt` est un arbre binaire de recherche alors `(list_of_btree bt)` est une liste triée.

4. On peut montrer qu'elle est équivalente à la définition donnée en termes de nombres d'occurrences. Mais la preuve de cette équivalence est assez ardue : nous ne l'aborderons pas.

Arbres binaires de recherche On se donne une structure générique d'arbres binaires

```
Inductive btree (A:Type) : Type :=
  empty : (btree A)
| node : (btree A) -> A -> (btree A) -> (btree A).

Arguments empty [A].
Arguments node [A] _ _ _.
```

Commentaire
— notez l'usage du *pragma* `Arguments` pour indiquer l'argument implicite des constructeurs ;

Pour définir la propriété caractéristique des arbres binaires de recherche, il faut poser la notion d'appartenance d'un élément à un arbre. On peut calquer cette définition sur celle du prédicat `In` des listes :

```
Fixpoint In_btree A:Set (x:A) (bt:btree A) : Prop :=
  match bt with
      empty => False
    | (node bt1 y bt2) => (x=y) \/ (In_btree x bt1) \/ (In_btree x bt2)
  end.
```

Les clauses (inductives) de la définition d'un arbre binaire de recherche sont :

1. l'arbre vide `empty` est un arbre binaire de recherche
2. si `bt1` et `bt2` sont des arbres binaires de recherche, si `x` est plus grand que les éléments de `bt1` et plus petit que les éléments de `bt2` alors, `(node bt1 x bt2)` est un arbre binaire de recherche.

Ce qui se transcrit :

```
Inductive Bstree : (btree nat) -> Prop :=
  bstree_empty : (Bstree empty)
| bstree_node : forall (x:nat) (bt1 bt2:btree nat),
                  (Bstree bt1) -> (forall (y:nat), (In_btree y bt1) -> (le y x))
                  -> (Bstree bt2) -> (forall (y:nat), (In_btree y bt2) -> (le x y))
                  -> (Bstree (node bt1 x bt2)).
```

Construction de l'arbre binaire de recherche On construit un arbre binaire à partir d'une liste par itération de la fonction d'insertion suivante :

```
Fixpoint ins_btree (x:nat) (bt:(btree nat)) : (btree nat) :=
  match bt with
      empty => (node empty x empty)
    | (node bt1 y bt2) =>
      if (leb x y) then (node (ins_btree x bt1) y bt2)
      else (node bt1 y (ins_btree x bt2))
  end.
```

L'itération est définie par

```
Fixpoint btree_of_list (xs:list nat) : (btree nat) :=
  match xs with
      nil => empty
    | (cons x xs) => (ins_btree x (btree_of_list xs))
  end.
```

3.4 Correction de `btree_of_list`

Pour s'assurer que la fonction `btree_of_list` construit effectivement une arbre binaire de recherche, il faut démontrer :

```
Lemma bstree_of_list :
  forall (xs:list nat), (Bstree (btree_of_list xs)).
```

Avant de nous lancer dans la preuve ce cet énoncé, demandons nous pourquoi la fonction `btree_of_list` est correcte. La fonction `btree_of_list` procède par itération de la fonction `ins_btree` sur les éléments de la liste passée en argument. Par exemple, `(btree_of_list (cons x1 (cons x2 nil)))` est symboliquement égale à `(ins_btree x1 (ins_btree x2 empty))`. Montrer `(Bstree (btree_of_list (cons x1 (cons x2 nil))))` revient donc à montrer `(Bstree (ins_btree x1 (ins_btree x2 empty)))`.

Notons que `(ins_btree x2 empty)` qui est égal à `(node empty x2 empty)` est bien un arbre binaire de recherche : `empty` est, par définition un arbre binaire de recherche et `x2` est à la fois trivialement plus grand et plus petit que les éléments de `empty` puisque celui n'en contient pas. Nous avons donc l'arbre binaire de recherche `(node empty x2 empty)` et nous voulons vérifier que l'ajout de `x1` (avec la fonction `ins_btree`) produit un arbre binaire de recherche. Cela est vrai par définition de `ins_btree` qui placera `x1` soit à gauche, soit à droite de la racine `x2` selon que `x1` est inférieur (au sens large) ou non à `x2`.

- si `x1` est inférieur ou égal à `x2`, l'arbre produit est `(node (node empty x1 empty) x2 empty)` et c'est un arbre binaire de recherche car
  1. `(node empty x1 empty)`
  2. tous les éléments de `(node empty x1 empty)`, c'est-à-dire `x1`, sont inférieurs ou égaux à `x2`
  3. `empty` est un arbre binaire de recherche
  4. `x2` est trivialement inférieur ou égal aux éléments de `empty` qui n'en contient pas.
- sinon, le raisonnement est analogue, *mutatis mutandis*.

On a vu un cas particulier d'ajout d'un élément dans un arbre binaire de recherche qui produit un nouvel arbre bainaire de recherche. Mais ce résultat se généralise : si `bt` est un arbre binaire de recherche, alors `(ins_btree x bt)` est un arbre binaire de recherche. En d'autres termes, la propriété «est un arbre binaire de recherche» (`Bstree`) est *invariante* par la fonction `ins_btree` [5] (nous le montrerons).

En attendant, ce résultat nous permet d'obtenir la preuve du lemme `bstree_of_list` par induction sur la liste :

- le cas de la liste vide est immédiat
- si la liste a la forme `(cons x xs)`, pour montrer que `(btree_of_list (cons x xs))` est un arbre binaire de recherche, il faut montrer que `(ins x (btree_of_list xs))` est une arbre binaire de recherche. Pour cela, on aura comme hypothèse d'induction que `(btree_of_list xs)` est un arbre binaire de recherche. Il suffit donc d'appliquer l'invariance de `Bstree` pour `ins_btree`.

Invariance de `Bstree` par `ins_btree` Le lemme à montrer est le suivant :

```
Lemma bstree_ins :
  forall (bt:btree nat) (x:nat), (Bstree bt) -> (Bstree (ins_btree x bt)).
```

Ce lemme est le *lemme clef* de la correction de `btree_of_list` et sa preuve demande encore un peu d'efforts. On se doute que le cas le plus délicat sera celui où `bt` n'est pas vide et a la forme `(node bt1 y bt2)`. En procédant par induction sur `bt`, sous hypothèses que `(ins_btree x bt1)` et `(ins_btree x bt2)` sont des arbres binaires de recherche, il faudra montrer que `(node (ins_btree x bt1) y bt2)`, lorsque `x` est inférieur ou égal à `y`, et que `(node bt1 y (ins_btree x bt2))`, sinon.

5. `btree_of_list` procède par itération de `ins_btree`. Elle agit comme un *boucle* qui répéte l'application de `ins_btree`. La preuve de correction de la boucle réalisée par `btree_of_list` s'obtient à la manière on obtient la correction d'une boucle par invariance dans la logque de Hoare, par exemple.

Considérons d'abord la cas où `x` est inférieur ou égal à `y` pour montrer que `(node (ins_btree x bt1) y bt2)` est un arbre binaire de recherche. Pour cela, il faut vérifier les quatre clauses de `bstree_node` :

1. `(Bstree (ins_btree x bt1))` nous est donné par hypothèse d'induction.
2. on veut que tous les éléments de `(ins_btree x bt1)` soient inférieurs ou égaux à `y`. Ce que l'on a car les éléments de `(ins_btree x bt1)` sont soit `x` qui est supposé ici inférieur ou égal à `y`, soit les éléments de `bt1` qui sont également inférieurs ou égaux à `y` puisque, par hypothèse, `(node bt1 y bt2)` est un arbre binaire de recherche.
   Bien entendu, il faudra démontrer qu'en général, les éléments de `(ins x bt)` sont soit `x` soit ceux de `bt`. Nous le ferons.
3. `bt2` est un arbre binaire de recherche par hypothèse que `(node bt1 y bt2)` l'est.
4. et `y` est inférieur ou égal aux éléments de `bt2` pour la même raison.

Le cas où `x` n'est pas inférieur ou égal à `y`, c'est-à-dire que `y` est strictement inférieur à `x`, est très proche. La seule clause qui demande un peu de travail est celle qui réclame que `y` soit inférieur ou égal à tous les éléments de `(ins_btree x bt2)`. En utilisant que les éléments de `(ins_btree x bt2)` sont soit `x` soit ceux de `bt2`, on raisonne ainsi :

— `y` est inférieur ou égal aux éléments de `bt2` car, par hypothèse, `(node bt1 y bt2)` est un arbre binaire de recherche ;
— `y` est inférieur ou égal à `x`, car, par hypothèse, il est strictement inférieur à `x`.

Pour que la preuve de `bstree_ins` soit achevée, il reste à montrer que les éléments de `(ins_btree y bt)` sont `y` ou ceux de `bt` :

```
Lemma mem_ins_alt :
  forall (bt: btree nat) (x y:nat),
    (In_btree x (ins_btree y bt)) -> (x=y) \/  (In_btree x bt).
```

Par induction sur `bt`

— si `bt` est l'arbre vide `empty` : sous hypothèse que `(In_btree x (ins_btree y empty))`, il faut montrer que `(x=y) \/ (In_btree x empty)`. Par hypothèse, et définition de `In_btree`, `(x=y)` ou `(In_btree y empty)`. On a donc ce que l'on veut.
— si `bt` est de la forme `(node bt1 z bt2)`. On a les deux hypothèse d'induction que pour tout `x` et `y`, `(In_btree x (ins_btree y bt1)) -> (x=y) \/ (In_btree x bt1)` et `(In_btree x (ins_btree y bt2)) -> (x=y) \/ (In_btree x bt2)` ; et il faut montrer que si `(In_btree x (ins_btree y (node bt1 z bt2)))` alors `(x=y) \/ (In_btree x (node bt1 z bt2))`. Par définition de `In_btree`, il faut montrer en fait que (a) `x=y` ou (b) `(In_btree x bt1)` ou (c) `x=z` ou (d) `(In_btree x bt2)`
  On raisonne par cas sur `(leb y z)` pour simplifier l'hypothèse `(In_btree x (ins_btree y (node bt1 z bt2)))` :
  — si `(leb y z)` vaut `true`, notre hypothèse devient `(In_btree x (node (ins_btree y bt1) z bt2))`. On a donc, par définition de `In_btree`, que (1) `x=z`, (2) `(In_btree x (ins_btree y bt1))` ou (3) `(In_btree x bt2)`. Dans le cas (1) on a (c). Dans le cas (2), on peut utiliser la première hypothèse d'induction pour obtenir (a) ou (b). dans le cas (3), on a (d).
  — sinon, le raisonnement est similaire en utilisant la seconde hypothèse d'induction.

3.5 Correction de `list_of_btree`

Nous devons assurer que la fonction `list_of_btree` produit une liste triée si son argument est un arbre binaire de recherche. c'est-à-dire :

```
Lemma sorted_list_of_btree :
  forall (bt:btree nat),
    (Bstree bt) -> (Sorted (list_of_btree bt)).
```

On peut conduire cette preuve par nduction sur l'hypothèse `(Bstree bt)` :

— cas `bstree_empty` : dans ce cas, `bt` est l'arbre vide `empty` et `(list_ofbtree empty)` est la liste vide `nil` qui est triée, par définition de `Sorted`.

— cas `bstree_node` : dans ce cas, `bt` a la forme `(node bt1 x bt2)` ; sous les hypothèses (H0) `forall y : nat, (In_btree y bt1) -> y <= x`, (H2) `forall y : nat, (In_btree y bt2) -> x <= y`, (IH1) `(Sorted (list_of_btree bt1)` et (IH2) `(Sorted (list_of_btree bt2))`, il faut montrer que `(Sorted (app (list_of_btree bt1) (cons x (list_of_btree bt2))))`.
En appliquant le lemme `sorted_app`, reste à montrer : (1) `list_of_btree bt1)` est triée, (2) `(cons x (list_of_btree bt2))` est triée, (3) que tous les éléments de `(list_of_btree bt1)` sont plus petits que les éléments de `(cons x (list_of_btree bt2))`.

1. nous est donné par hypothèse ;
2. en appliquent `sorted_cons`, reste à montrer `(Sorted (list_of_btree bt2)`, ce qui nous est donné par hypthèse et que `x` est inférieur ou égal à tous les éléments de `(list_of_btree bt2)`. On utilise pour cela l'hypothèse (H2), sachant que les éléments de `bt2` sont aussi des éléments de `(list_of_btree bt2)` (ce qu'il faudra démontrer : voir plus loin, lemme `in_list_in_btree`).
3. donnons nous `z` un élément de `(list_of_btree bt1)`, `y` un élément de `(cons x (list_of_btree bt2))` et montrons que `(le z y)`.
   De `y` élément de `(cons x (list_of_btree bt2))`, on a que `y=x` ou `y` est élément de `(list_of_btree bt2)`.
   — si `x=y`, il faut montrer `(le z x)` : on applique (H0) puis `in_list_in_btree` ;
   — si `y` est un élément de `(list_of_btree bt2)`, on utilise la transitivité de `le` pour montrer `(le z x)` et `(le x y)`. On a `(le z x)` comme ci-dessus et `(le x y)` par (H2) et `in_list_in_btree`.

Montrons à présent que tous les éléments de la lsite extraits par `list_of_btree` d'un arbre binaire sont bien tous les éléments de l'arbre binaire. C'est-à-dire :

```
Lemma in_list_in_btree :
  forall (x:nat) (bt:btree nat),
    (In x (list_of_btree bt)) -> (In_btree x bt).
```

Par induction sur `bt`

— si `bt` est l'arbre vide, la liste `(list_of_btree bt)` est la liste vide. Il faut montrer que `(In_btree x empty)` sous l'hypothèse que `(In x nil)`. ce qui est trivialement vrai, par vacuité de l'hypothèse.

— si `bt` est de la forme `(node bt1 y bt2)`, on a par hypothèse d'induction que (IHbt1) `(In x (list_of_btree bt1)) -> (In_btree x bt1)` et que (IHbt2) `(In x (list_of_btree bt2)) -> (In_btree x bt2)`, on suppose `(In x (app (list_of_btree bt1) (cons y (list_of_btree bt2))))` et il faut montrer que `(x=y) \/ (In_btree bt1) \/ (In_btree bt2)`.
On utilise pour cela, une propriété de la concaténation qui est que si `z` appartient à `(app xs ys)` alors `z` apparatient à `xs` ou `z` appartient à `ys` (cf. `app_in` ou `in_app_or` de la bibliothèque standard `Coq.List`) en conjonction avec l'hypothèse `(In x (app (list_of_btree bt1) (cons y (list_of_btree bt2))))`.
   — si `x` appartient à `(list_of_btree bt1)`, on a `(in_btree bt1)` par hypothèse d'induction (IHbt1) ;
   — si `x` appartient à `(cons y (list_of_btree bt2))` alors `x=y` ou `x` appartient à `(list_of_btree bt2)`. dans le premier cas, on a l'un des termes de la disjonction rechcerhcée, dans l'autre, on l'obtient par l'hypothèse d'induction (IHbt2).

3.6 Correction de `bst_sort`, pour l'ordre

On rappele la définition de la fonction de tri :

```
Definition bst_sort (xs:list nat) :=
  (list_of_btree (btree_of_list xs)).
```

On montre que `bst_sort` calcule une liste triée en montrant

```
Theorem bst_sort_sorted :
  forall (xs:list nat), (Sorted (bst_sort xs)).
```

Ce qui s'obtient facilement, sachant que `(btree_of_list xs)` est un arbre binaire de recherche (lemme `bstree_of_list` et que `list_of_btree` en extrait une liste triée (lemme `sorted_lis_of_btree`).

3.7 Correction de `bst_sort`, pour la permutation

Pour achever la preuve de correction de notre fonction de tri, reste à montrer que `(bst_sort xs)` est une permutation de `xs` ou, de manière équivalente, que `xs` est une permutation de `(bst_sort xs)` :

```
Theorem bst_sort_permut :
  forall (xs:list nat), (Permut xs (bst_sort xs)).
```

La preuve s'obtient par induction sur `xs`. La cas où `xs` est la liste vide est trivial. Dans le cas où la liste est de la forme `(cons x xs)`, il faut montrer, après évaluation symbolique, que `(cons x xs)` est une permutation de `(list_of_btree (ins_btree x (btree_of_list xs)))` sous l'hypothèse d'induction que `xs` est une permutation de `(list_of_btree (btree_of_list xs))`. L'obstacle à franchir est de pouvoir utiliser l'hypothèse d'induction.
Rappelons nous que si `xs` est une permutation de `ys`, alors `(cons z xs)` est une permutation de `(cons z ys)` (cas `permut_cons1`). Avec notre hypothèse d'induction, cela nous permettra d'obtenir que `(cons x xs)` est une permutation de `(cons x (list_of_btree (btree_of_list xs)))`.
On se rappelle également que la relation de permutation est transitive par définition. Pour obtenir que `(cons x xs)` est une permutation de `(list_of_btree (ins_btree x (list_of_btree xs)))`, il suffit donc d'obtenir que `(cons x (list_of_btree (btree_of_list xs)))` est une permutation de `(list_of_btree (ins_btree x (btree_of_list xs)))`. Ce qui est intuitivement assez clair en *généralisant* `(btree_of_list xs)` : ajouter `x` devant la liste obtenue par extraction des éléments d'un arbre binaire donne bien une liste contenant les mêmes éléments que celle obtenue par extraction des élément du même arbre où l'on a inséré `x`. Formellement :

```
Lemma permut_cons_ins_btree :
  forall (x:nat) (bt:btree nat),
    (Permut (cons x (list_of_btree bt)) (list_of_btree (ins_btree x bt))).
```

La preuve de ce lemme, par induction sur `bt`, est simplement un peu technique et demande d'établir une certain nombre de propriétés sur les permutations dont voici la liste :

- la relation de permutation est réflixive ;
- si `xs` est une permutation de `ys` alors `(app xs zs)` est une permutation de `(ys zs)` ;
- si `xs` est une permutation de `ys` alors `(app zs xs)` est une permutation de `(zs ys)` ;
- `(cons z (app xs ys))` est une permutation de `(app xs (cons z ys))`.