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

P. Manoury

sept. 2021

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

4 Terminaison, bonne fondation

`Fixpoint`, récurrence structurelle

Nous avons vu que lorsque l'on utilise la clause `Fixpoint` pour poser une définition récursive celle-ci est soumise à l'examen du respect d'un *critère syntaxique de décroissance structurelle* dans ses appels récursifs appelé *garde*.

Lorsqu'on lui soumet la défintion

```
Fixpoint fact (n:nat) : nat :=
  match n with
    0 => 1
  | S n0 => ((S n0) * (fact n0))
  end.
```

le sytème répond :

```
fact is defined
fact is recursively defined (guarded on 1st argument)
```

Avec cette définition, la valeur de `(fact (S n0))` est la valeur de `((S n0) * (fact n0))` où `fact` est appliquée à la variable `n0` qui est un *sous-terme strict* de `(S n0)`. Cette propriété syntaxique de l'appel récursif suffit à garantir la terminaison de fact : à force de retirer un `S` à chaque appel récursif, il n'y en a plus et on tombe sur le cas d'arrêt `(fact 0)`.

En fait la clause `Fixpoint` est simplement une abréviation pour

```
Definition fac : nat -> nat :=
  fix f n := match n with 0 => 1 | S n0 => (S n0) * (f n0)
```

où `fix` est un constructeur de terme appelé constructeur de *point fixe*. Sa forme générale est `fix f x := t`. Il y a nécessairement deux *paramètres* après `fix` : `f` est un nom pour une fonction dont le paramètre est `x`. Le terme lui même `fix f x := t` désigne une fonction dont le paramètre est `x`.

La règle d'évaluation de l'application de `fix` est :

$$\texttt{((fix f x := t) u)} = \texttt{t[f}\leftarrow\texttt{fix f x := t][x}\leftarrow\texttt{u]}$$

où $\_\_[\_\_ \leftarrow \_\_]$ est l'opération de *substitution* d'un terme à une variable. La substitution `[f←fix f x := t]` réalise le processus récursif d'évaluation.

Le constructeur `fix` autorise que `f` soit une fonction à plusieurs arguments. Dans ce cas, le système en cherche un pouvant servir de garde. Ce n'est pas nécessairement le premier :

```
Fixpoint mul (n m:nat) : nat :=
  match n, m with
    0, m => 0
  | 1, m => 1
  | n, 0 => 0
  | n, (S m) => n + (mul n m)
  end.

mul is defined
mul is recursively defined (guarded on 2nd argument)
```

Dans certains cas, l'un ou l'autre des arguments peut servir de garde. Le système choisit arbitrairement :

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

min is defined
min is recursively defined (guarded on 1st argument)
```

Le critère de garde peut être plus général que de passer d'un appel sur `(S n)` à un appel sur `n`. Par exemple :

```
Fixpoint even (n:nat) : bool :=
  match n with
    0 => true
  | 1 => false
  | (S (S n1)) => (even n1)
  end.

even is defined
even is recursively defined (guarded on 1st argument)
```

Toutefois, si nous essayons d'utiliser cette généralité pour définir

```
Fixpoint fib (n:nat) : nat :=
  match n with
    0 => 0
  | 1 => 1
  | (S (S n)) => (fib n) + (fib (S n))
  end.
```

nous échouons sur le sybillin

```
Error:
Recursive definition of fib is ill-formed.
In environment
fib : nat -> nat
n : nat
n0 : nat
n1 : nat
Recursive call to fib has principal argument equal to
"S n1" instead of one of the following variables: "n0"
"n1".
Recursive definition is:
"fun n : nat =>
 match n with
 | 0 => 0
 | 1 => 1
 | S (S n1) => fib n1 + fib (S n1)
 end".
```

C'est que le critère syntaxique de garde est assez simpliste : pour déterminer si un argument peut servir de garde, il cherche si, dans chaque cas d'appel récursif, une *variable issue du motif de filtrage* de cet argument vient prendre sa place. Par exemple dans l'appel récursif `(even n1)` la variable `n1` vient du cas de filtrage de l'argument `n` avec le motif `(S (S n1))`.

Mais dans notre tentavive de définition de `fib`, nous avons mentionné l'appel récursif `(fib (S n1))` où `(S n1)` n'est pas une *variable issue du filtrage*. On peut contourner cette faiblesse de l'analyse en explicitant les deux étapes de filtrage :

```
Fixpoint fib (n:nat) : nat :=
  match n with
    0 => 0
  | (S n0) =>
      match n0 with
        0 => 1
      | (S n1) => (fib n0) + (fib n1)
      end
  end.

fib is defined
fib is recursively defined (guarded on 1st argument)
```

Le critère est satisfait : `n0` est issue du filtrage de `n` et `n1` est issue du filtrage de `n0`, elle-même issue du filtrage de `n`.

Double récurrence Dans certains cas, le choix de la garde n'est pas possible :

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

La réponse du système à cette définition est :

`Error: Cannot guess decreasing argument of fix.`

Quoique effectivement bien définie sur le dommaine des couples d'entiers, cette fonction n'est pas définissable ainsi avec la clause `Fixpoint`.

Cette définition récursive est bien définie car quelque soit les entiers n et m, l'évaluation de `(fack` n m`)` termine toujours. Intuitivement : dans l'appel `(fack n (S m))`, le premier paramètre décroît et mènera au cas d'arrêt `(0,m)` ; dans l'appel `(fack (S n) m)`, c'est le second paramètre qui décroît pour mener au cas d'arrêt `(n,0)`. Plus formellement, on peut prouver que *pour tout n,m, `(fack` n m`)` termine*. On montre par récurrence sur n que *pour tout m, `(fack` n m`)` termine*. Si $n = 0$ c'est immédiat (cas d'arrêt), sinon, supposons (hypothèse de récurrence 1) *pour tout m, `(fack` n' m`)` termine*. On montre par récurence sur m que *`(fack (S` n'`)` m`)` termine*. Si $m = 0$, on a également un cas d'arrêt. Sinon, on suppose (hypothèse de récurrence 2) *`(fack (S` n'`)` m'`)` termine* et il faut montrer que *`(fack (S` n'`) (S` m'`))` termine*, c'est-à-dire, que *`(fack` n' `(S` m'`)) + (fack (S` n'`)` m'`)` termine*. Sachant que l'addition termine, il suffit de montrer que *`(fack` n' `(S` m'`))` termine* et *`(fack (S` n'`)` m'`)` termine*. On a le premier par l'hypothèse de récurrence 1 (par instanciation de m qui est universellement quantifié) et le second par l'hypothèse de récurrence 2.

Mais la clause `Fixpoint` cherche à vérifier la décroissance d'un seul et d'un seul paramètre de la fonction. C'est pourquoi elle échoue à établir la terminaison de `fack` tel que nous en avons proposé une définition.

On peut proposer une autre défintion de `fack` qui satisfera la condition de garde des points fixes. Mais il faut pour cela changer notre vision de la fonction. Voici comment : on veut définir une fonction à deux paramètres entiers qui calcule un entier, c'est-à-dire, une fonction de type `nat -> nat -> nat`. Or, une telle fonction est, dans les faits, une fonction à un seul paramètre de type `nat` qui calcule une fonction de type `nat -> nat`, elle-même, fonction à un seul paramètre. En effet, l'écriture du type `nat -> nat -> nat` est un raccourci pour `nat -> (nat -> nat)`.

Si `f` est de type `nat -> (nat -> nat)` et `n`, de type `nat` alors `(f n)` est une fonction de type `nat -> nat`. Si l'on considère les applications de `fack` à son seul ≪premier≫ argument, on a une suite de fonctions `(fack 0)`, `(fac 1)`, `(fack 2)`, etc. telles que

- `(fack 0)` est la fonction identité `fun m => m` ;
- `(fack 1)` est la fonction
  `fun m => match m with 0 => 1 | (S m0) => ((fack 0) (S m0)) + ((fack 1) m0).`
- `(fack 2)` est la fonction
  `fun m => match m with 0 => 2 | (S m0) => ((fack 1) (S m0)) + ((fack 2) m0).`
- etc.

Cette suite de fonctions est correctement définie. En effet,

- `(fack 0)` est définie par l'identité qui est bien définie ;
- `(fack 1)` est bien définie au cas où `m` vaut 0 ; dans l'autre cas, lorsque `m` est `(S m0)` : l'addition est bien définie, `(fack 0)` est bien définie par ci-dessus et, *l'appel récursif* de `(fack 1)` est appliqué à `m0` qui est une variable issue du filtrage de `m`.

L'argument utilisé par `(fack 1)` vaut pour `(fack 2)`, `(fack 3)`, etc.

En généralisant, `(fack (S n))` est définie comme la fonction

```
fun m => match m with 0 => (S n) | (S m) => ((fack n) (S m)) + ((fack (S n)) m)
```

C'est une définition récursive au rang `(S n)` qui utilise également la définition au rang `n`.

Supposons `n` connu (nous verrons tout à l'heure comment) et appelons `fack'` la fonction `(fack (S n))`. On peut définir `fack'` avec un constructeur de point fixe :

```
fix fack' m := match m with 0 => (S n) | (S m) => ((fack n) (S m)) + (fack' m)
```

On a donc :

`(fack 0)` = `(fun m => m)`

```
    (fack (S n))
      = fix fack' m := match m with 0 => (S n) | (S m) => ((fack n) (S m)) + (fack' m)
```
En utilisant le constructeur de point fixe, `fack` est définie par le terme

```
fix fack n :=
  match n with
    0 => (fun m => m)
  | (S n) =>
    fix fack' m :=
      match m with
        0 => (S n)
      | (S m) => (fack n (S m)) + (fack' m)
      end
   end.
```

On peut donc poser :

```
Fixpoint fack (n:nat) : nat -> nat :=
  match n with
    0 => (fun m => m)
  | (S n) =>
    fix fack' (m:nat) : nat :=
      match m with
        0 => (S n)
      | (S m) => (fack n (S m)) + (fack' m)
      end
  end.
```

qui est acceptée par les système :

```
fack is defined
fack is recursively defined (guarded on 1st argument)
```

Pour donner un exemple moins artificiel d'une fonction défnie par double récurrence avec une *point fixe local*, on peut citer la fonction de fusion de deux listes :

```
Fixpoint merge (ns1 : list nat) : (list nat) -> (list nat) :=
  match ns1 with
      nil => (fun (ns2:list nat) => ns2)
    | (n1::ns1) =>
      (fix merge' (ns2 : list nat) : (list nat) :=
         match ns2 with
             nil => (cons n1 ns1)
           | (n2::ns2) =>
             if (n1 <=? n2) then (n1::(merge ns1 (n2::ns2)))
             else (n2::(merge' ns2))
         end)
  end.
```

La tactique `simpl` est moins facilement utilisable sur ce genre de terme.

```
Goal forall (ns1 ns2:list nat),
    (merge (0::ns1) ns2) = 0::(merge ns1 ns2).
  intros. simpl.
```

produit

```
1 subgoal (ID 19)

  ns1, ns2 : list nat
  ============================
  (fix merge' (ns0 : list nat) : list nat :=
     match ns0 with
     | nil => 0 :: ns1
     | n2 :: ns3 => 0 :: merge ns1 (n2 :: ns3)
     end) ns2 = 0 :: merge ns1 ns2
```

Il est alors utile de suppléer à cette obscurité en démontrant les égalités attendues pour la fonction :

```
Fact merge_eq1 : forall (ns:list nat), (merge nil ns) = ns.
Proof.
  auto.
Qed.

Fact merge_eq2 : forall (ns:list nat), (merge ns nil) = ns.
Proof.
  intro. case ns; auto.
Qed.

Fact merge_eq3 : forall (n1 n2:nat) (ns1 ns2:list nat),
    (merge (n1::ns1) (n2::ns2))
     = (if (n1 <=? n2) then n1::(merge ns1 (n2::ns2))
                       else  n2::(merge (n1::ns1) ns2)).
Proof.
  intros. simpl. case (n1 <=? n2); trivial.
Qed.
```

`Function`, `Equations`, ordre bien fondé

Nous avons justifié intuitivement, sur l'exemple simple des entiers, le critère de garde syntaxique par *«à force de retirer un `S` à chaque appel récursif, il n'y en a plus et on tombe sur le cas d'arrêt»*. Il y a dans cette justification deux composantes :

1. *«retirer un `S`»*
2. *«à force [...] il n'y en a plus»*

Le première composante dit que, en retirant un `S`, la fonction de l'appel récursif est appliquée à une valeur *plus petite* ; on a un *ordre* sur l'argument considéré lors des appels récursifs. La deuxième dit que l'on ne peut pas retirer infiniment un `S` (il y a une valeur minimale pour l'ordre). Un ordre qui n'admet pas de suite infinie décroissante est appelé *un ordre bien fondé*. C'est le cas de l'ordre (strict) sur les entiers naturels (les entiers positifs). Ce n'est bien entendu pas le cas pour les entiers relatifs (on peut descendre à l'infini dans les négatifs).

Le critère de garde syntaxique, appliqué aux entiers naturels, et qui consiste à vérifier que la valeur d'un argument passe de `(S n)` à `n` (c'est-à-dire, de $n+1$ à n) revient en fait à vérifier que la valeur d'un argument passe d'une valeur à une une valeur strictement inférieure pour l'ordre sur les entiers ($n+1 > n$). C'est le même principe qui vaut lorsque l'on admet la définition d'une fonction récursive qui passe de `(S (S n))` à `n` ($n+2 > n$).

En général donc, si l'on veut s'assurer de la terminaison des fonctions récursives, il suffit de savoir établir que dans les appels récursifs de la fonction un certain argument (toujours le même) passe d'une valeur à une

valeur strictement inférieure. Comme, par exemple dans :

$$\begin{cases} pgcd(a,0) & = & a \\ pgcd(a,b) & = & pgcd(b, a \bmod b) \quad \text{si } b \neq 0 \end{cases}$$

On a bien que, lorsque $b \neq 0$, $(a \bmod b) < b$.

Le système Coq propose quelques clauses de définitions de fonctions récursives alternatives à `Fixpoint` lorsque l'on n'a pas de définition qui puisse satisfaire le critère de garde. Citons les clauses `Function`, `Program Fixpoint` et `Equations`.

Pour définir une fonction récursive en utilisant une telle clause, il faut spécifier la relation d'ordre et l'argument décroissant pour cette relation. Par exemple :

```
Require Import FunInd Recdef.

Function pgcd (a b:nat) {wf lt b} : nat :=
  match b with
    0 => a
  | _ => (pgcd b (a mod b))
  end.
```

C'est à l'utilisateur de fournir la justification de la décroissance de l'argument et de la bonne fondation de l'ordre utilisé. Pour notre exemple, le système répond à notre proposition de définition par l'injonction de satisfaire des *obligations de preuve* :

```
2 subgoals (ID 532)

  ============================
  forall a b n : nat, b = S n -> a mod S n < S n

subgoal 2 (ID 533) is:
 well_founded lt
```

On peut utiliser ici les résultats de la bibliothèque standard :

```
- intros. Search (_ mod _ < _).
  apply Nat.mod_upper_bound.
  intro. discriminate.
- apply lt_wf.
```

Variant, mesure L'ordre bien fondé sur les entiers naturels est en fait sous-jacent à un grand nombre de preuves de terminaison, même au-delà du domaine des définitions récursives. Par exemple, c'est lui qui préside à l'utilisation des *variants* pour prouver la terminaison des boucles dans les langages impératifs. Ceux-ci sont donnés par la valeur d'une variable (entière) du programme, par la combinaison des valeurs de plusieurs variables (leur somme, etc.) ou encore par la valeur d'une fonction associant aux valeurs des variables de type non entier, une valeur entière (typiquement : la taille d'une liste). On parle dans ce dernier cas de *mesure* de la donnée. On montre alors que la valeur de la mesure décroît.

Considérons la fonction *bubble* définie par

$$\begin{cases} bubble(nil & = & nil \\ bubble(n :: nil) & = & n :: nil \\ bubble(n_1 :: n_2 :: ns) & = & n_1 :: bubble(n_2 :: ns) \quad \text{si } n_1 \leq n_2 \\ & = & n_2 :: bubble(n_1 :: ns) \quad \text{sinon} \end{cases}$$

Le *variant* de cette fonction est donné par la longueur de la liste.

Avec a la clause `Function`, on utilise cela de la manière suivante :

```
Function bubble (ns:list nat) {measure length ns} : list nat :=
  match ns with
    n1::n2::ns =>
    if (n1 <=? n2) then n1::(bubble (n2::ns))
    else n2::(bubble (n1::ns))
           | _ => ns
  end.
```

Le système demande de prouver

```
2 subgoals (ID 1033)

  [..] length (n2 :: ns0) < length (n1 :: n2 :: ns0)

subgoal 2 (ID 1034) is:
  [..] length (n1 :: ns0) < length (n1 :: n2 :: ns0)
```

Ce qui est facile à obtenir.

La mesure peut concerner plusieurs arguments de la fonction. Illustrons le en donnant une définition alternative de la fonction `merge`. Nous utilisons pour cela la clause `Equation`.

```
From Equations Require Import Equations.

Equations merge' (xs ys:list nat) : list nat by wf (length xs + length ys) lt :=
  merge' nil ys := ys;
  merge' xs nil := xs;
  merge' (x::xs) (y::ys) :=
    if (x <=? y) then x::(merge' xs (y::ys))
    else y::(merge' (x::xs) ys).
```

La *mesure* utilisée et les arguments auquels elle s'applique sont ici directement donnés par l'expression `(length xs + length ys)`.

Le système trouve de lui-même la preuve de la première obligation (`length xs + length (y :: ys) < length (x :: xs) + length (y :: ys)`) et laisse au programmeur le soin de prouver la seconde (`length (x :: xs) + length ys < length (x :: xs) + length (y :: ys)`).

Lorsque les obligations de preuve sont satisfaites, le système engendre et prouve automatiquement les équations posées pour la définition. Par exemple :

```
  Check merge'_equation_3.

  merge'_equation_3
     : forall (n : nat) (l : list nat) (n0 : nat) (l0 : list nat),
       merge' (n :: l) (n0 :: l0) =
       (if n <=? n0 then n :: merge' l (n0 :: l0) else n0 :: merge' (n :: l) l0)
```

Induction bien fondée

Dans la théorie des types qu'est le calcul des constructions inductives, la propriété de bonne fondation d'une relation (binaire) est définie en terme *d'accessibilté* par un type inductif :

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
    Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

Intuitivement, le type du constructeur `Acc_intro` pose que tout élément `x` de `A` est *accessible* comme ultime élément d'une suite d'éléments `y` de `A` qui sont tous dans la relation `R` avec `x`. Cela induit qu'il y a un «début» à toute suite e_0, e_1, e_2, etc. telle que $e_0 \mathtt{R} e_1 \mathtt{R} e_2$, etc. Si `R` est une relation d'ordre, cela donne que l'on ne peut construire de suite croissante sans «début», c'est-à-dire : il n'y a pas de suite infinie décroissante.

Le prédicat de bonne fondation d'une relation est simplement défini comme :

```
Definition well_founded := forall a:A, Acc a.
```

Un ordre bien fondé donne un *principe d'induction généralisé* :

$$(\forall x, (\forall y < x, P(y)) \to P(x)) \to \forall x, P(x)$$

Pour un ordre bien fondé donné, ce principe vient du principe d'induction structurelle associé à la définition inductive du prédicat d'accessibilité :

```
Check Acc_ind.

forall (A : Type) (R : A -> A -> Prop) (P : A -> Prop),
  (forall (x : A), (Acc R x) -> (forall (y : A), (R y x) -> (P y)) -> (P x))
   -> forall (x : A), (Acc R x) -> (P x)
```

En utilisant ce principe et la bonne fondation de l'ordre sur les entiers, on montre :

```
Lemma lt_ind : forall (n : nat) (P : nat -> Prop),
    (forall n0 : nat, (forall m : nat, m < n0 -> P m) -> P n0) -> P n.
Proof.
  intros. apply Acc_ind with (R:=lt).
  - auto.
  - apply lt_wf.
Qed.
```

qui est le principe d'induction généralisé sur les entiers.

Induction générale sur les listes On peut obtenir un principe d'induction sur les listes reposant sur l'ordre donné par les tailles des listes :

```
Definition lt_list (A:Set) (xs ys:list A) : Prop :=
  (length xs < length ys).
```

Il faut montrer que cet ordre est bien fondé. On le déduit de la bonne fondation de l'ordre sur les entiers en démontrant que toute liste de longueur inférieure à un entier est accessible [1] :

```
Lemma lt_Acc_lt_list :
  (forall (A:Set) (n:nat) (xs:list A), (length xs < n) -> (Acc (lt_list A) xs)).
Proof.
  induction n.
  - intros. absurd (length xs < 0); lia.
  - intros. apply Acc_intro. intros. unfold lt_list in H0.
    apply IHn. apply Nat.lt_le_trans with (m:=length xs).
    + assumption.
    + lia.
```

1. La bibliothèque standard fourni une version générique de ce lemme dans la preuve de `well_founded_ltof`.

```
Qed.
```

On en déduit trivialement l'accessibilité des listes pour l'ordre `lt_list` en particularisant le lemme ci-dessus avec un entiers supérieur à la taille de la liste considérée :

```
Lemma list_length_wf : forall (A:Set),
    well_founded (lt_list A).
Proof.
  unfold well_founded. intros.
  apply (lt_Acc_lt_list A (S (length a))). lia.
Qed.
```

De la bonne fondation de `lt_list` on déduit le principe d'induction suivant :

```
Theorem list_length_ind : forall (A:Set) (P:list A -> Prop),
    (forall (xs:list A),
        (forall (ys:list A), (length ys < length xs) -> (P ys))
        -> (P xs))
    -> forall (xs:list A), (P xs).
Proof.
  intros. apply Acc_ind with (R:=fun xs ys => length xs < length ys).
  - auto.
  - apply list_length_wf.
Qed.
```

Tri par sélection Illustrons l'utilisation de l'induction généralisée sur les listes en montrant la correction d'un fonction de tri par sélection. Nous ne détaillons ici que la partie de la preuve concernant explicitement l'induction généralisée.

On définit une fonction donnant le minimum d'une liste :

```
Fixpoint min_list (n:nat) (ns:list nat) : nat :=
  match ns with
    nil => n
  | m::ns =>
    if (n <=? m) then (min_list n ns)
    else (min_list m ns)
  end.
```

Stricto sensu, la fonction donne la valeur minimum entre `n` et le minimum de `ns`. En fait, elle donne le minimum de `n::ns`.

On définit ensuite une fonction qui retire d'une liste (une occurence de) sa valeur miminum

```
Equations remin (ns:list nat) : list nat by wf (length ns) lt :=
  remin nil := nil;
  remin (n::nil) := nil;
  remin (n1::n2::ns) :=
    if (n1 <=? n2) then (remin (n1::ns))
    else (remin (n2::ns)).
```

La fonction de tri est appelée récursivement sur la liste obtenue par retrait de la valeur minimum. Avant de pouvoir définir cette fonction, il faut s'assurer que

```
Lemma remin_length : forall (n:nat) (ns:list nat),
    length (remin (n :: ns)) < S (length ns).
```

On obtient assez facilement cette preuve par induction sur la liste. Il faut prendre garde à ce que `n` soit quantifié dans l'hypothèse de récurrence :

```
Proof.
  intros. generalize n. induction ns.
  - intro. rewrite remin_equation_2. lia.
  - intro. rewrite remin_equation_3. case (n0 <=? a).
    + simpl. apply lt_trans with (m:=(S (length ns))).
      * apply IHns.
      * lia.
    + simpl. apply lt_trans with (m:=(S (length ns))).
      * apply IHns.
      * lia.
Qed.
```

La fonction de tri est définie ainsi :

```
Equations sel_sort (ns:list nat) : list nat by wf (length ns) lt :=
  sel_sort nil := nil;
  sel_sort (n::ns) :=
    (min_list n ns)::(sel_sort (remin (n::ns))).
Obligation 1.
apply remin_length.
Qed.
```

Nous définissons la propriété d'ordonnancement d'une liste de la manière suivante :

```
Inductive le_list : nat -> list nat -> Prop :=
  Lle_nil : forall (n:nat), (le_list n nil)
| Lle_cons : forall (n m:nat),
    (n <= m) -> forall (ns:list nat),
      (le_list n ns) -> (le_list n (m::ns)).

Inductive sorted : list nat -> Prop :=
  sorted_nil : (sorted nil)
| sorted_cons : forall (n:nat) (ns:list nat),
    (le_list n ns) -> (sorted ns) -> (sorted (n::ns)).
```

La correction de `sel_sort` s'énonce :

```
Lemma sorted_sel : forall (ns:list nat), (sorted (sel_sort ns)).
```

Nous prouvons ce lemme par induction généralisé sur les listes. Outre la décroissance de `(remin (n::ns))` vis-à-vis de `n::ns` (lemme `remin_length`) piur appliquer l'hypothèse d'induction, nous aurons besoin d'un lemme «technique» qui établit que si une valuer est plus petite que les valeurs d'une liste (`le_list n ns`) alors elle est également plus petite que les valeurs de la liste triée (`le_min n (sel_sort ns)`). C'est le lemme

```
Lemma le_list_sorted : forall (n:nat) (ns:list nat),
    (le_list n ns) -> (le_list n (sel_sort ns)).
```

qui se démontre aussi par induction générale sur les listes. Ce lemme demande à son tour de montrer que

```
Lemma le_list_le_remin : forall (n m:nat) (ns:list nat),
    (le_list n (m::ns)) -> (le_list n (remin (m :: ns))).
```

que l'on montre encore par induction générale sur les listes.

Les preuves de ces deux lemmes sont données dans le fichier de script joint au cours. Examinons seulement la preuve du lemme principal :

```
Lemma sorted_sel : forall (ns:list nat), (sorted (sel_sort ns)).
Proof.
  induction ns using list_length_ind.
  destruct ns.
  - rewrite sel_sort_equation_1. apply sorted_nil.
  - rewrite sel_sort_equation_2. apply sorted_cons.
    + apply le_list_sorted. apply le_remin.
    + apply H. apply remin_length.
Qed.
```

Mesure lexicographique

Nous avons vu comment *étendre* l'ordre bien fondé sur les entiers en un ordre sur les listes par adjonction d'une fonction de mesure. Nous allons voir à présent une autre manière d'étendre un ordre bien fondé pour en obtenir un autre. Nous utiliserons ce dernier et joint à une fonction de mesure pour définir un autre plus complexe sur les arbres binaires.

On peut ordonner des couple d'entiers selon *l'ordre lexicographique* que l'on peut définir comme un type inductif :

```
Inductive nat_lex_lt : (nat * nat) -> (nat * nat) -> Prop :=
  Lt_fst : forall (x1 y1 x2 y2 : nat), (x1 < x2) -> (nat_lex_lt (x1,y1) (x2,y2))
| Lt_snd : forall (x y1 y2 : nat), (y1 < y2) -> (nat_lex_lt (x,y1) (x,y2)).
```

Nous l'utiliserons comme argument de terminaison pour la fonction `list_of` qui calcule, d'une manière assez particulière, la liste des étiquettes d'un arbre binaire.

Pour utiliser l'ordre lexicographique comme argument de terminaison, il faut démontrer sa bonne fondation :

```
  Lemma wf_nat_lex_lt : (well_founded nat_lex_lt).
```

Les preuves de bonne fondation sont un peu techniques. Aussi est-il bon d'avoir une idée du *plan de preuve* avant de passer à son codage dans le système Coq.

Voici donc comment on montre la bonne fondation de `nat_lex_lt` : par définition de `well_founded`, il faut montrer que `forall (c:nat*nat), (Acc nat_lex_lt c)`. Notons simplement `Acc` pour `Acc nat_lex_lt` et montrons que $\forall n, m.\texttt{Acc}(n, m)$. On montre $\forall m.\texttt{Acc}(n, m)$ par induction (structurelle) sur n :

- on montre $\texttt{Acc}(0, m)$ par induction sur m :
  - pour montrer $\texttt{Acc}(0, 0)$, il faut montrer que si $(p, q) < (0, 0)$ alors $\texttt{Acc}(p, q)$. Ce qui est trivial car $(p, q) < (0, 0)$ est absurde.
  - supposons (HR) $\texttt{Acc}(0, m)$ et montrons $\texttt{Acc}(0, Sm)$, c'est-à-dire, si $(p, q) < (0, Sm)$ alors $\texttt{Acc}(p, q)$. Si $(p, q) < (0, Sm)$ alors $p = 0$ et $q < Sm$. Il faut donc montrer que $\texttt{Acc}(0, q)$.
    Mais si $q < Sm$ alors $q = m$ ou $q < m$. Dans le premier cas, il faut montrer $\texttt{Acc}(0, m)$ ce qui est notre (HR) ; dans le second cas, par (HR) et définition de `Acc`, on a (H) $\forall(p', q') < (0, m) \rightarrow \texttt{Acc}(p', q')$. En appliquent (H), il reste à montrer que $(0, q) < (0, m)$ ce qui est donné par $q < m$.
- supposons (HR1) $\forall m.\texttt{Acc}(n, m)$ et montrons $\forall m.\texttt{Acc}(Sn, m)$ par induction sur m :
  - pour montrer $\texttt{Acc}(Sn, 0)$, il faut montrer que si $(p, q) < (Sn, 0)$ alors $\texttt{Acc}(p, q)$. Si $(p, q) < (Sn, 0)$ alors $q = 0$ et $p < Sn$, c'est-à-dire $p = n$ ou $p < n$. On montre $\texttt{Acc}(p, 0)$ par cas sur p.
    Si $p = n$, il faut montrer $\texttt{Acc}(n, 0)$ ce que nous donne directement (HR1)
    Si $p < n$, par (HR1), on a $\texttt{Acc}(n, 0)$, c'est-à-dire (H) $\forall p', q'.(p', q') < (n, 0) - > \texttt{Acc}(p', q')$. En appliquant (H), reste à montrer $(p, 0) < (n, 0)$. ce que l'on a par $p < n$.

— supposons (HR2) $\texttt{Acc}(Sn, m)$ et montrons $\texttt{Acc}(Sn, Sm)$, c'est-à-dire, si $(p, q) < (Sn, Sm)$ alors $\texttt{Acc}(p, q)$. Si $(p, q) < (Sn, Sm)$ alors $p < Sn$ ou $p = Sn$ et $q < Sm$. Pour montrer $\texttt{Acc}(p, q)$, on considère les cas suivants :
  — Si $p < Sn$ alors $p = n$ ou $p < n$. Par (HR1), on a (H) $\forall p', q', (p', q') < (Sn, Sm) \rightarrow \texttt{Acc}(p', q')$. En appliquant (H), reste à montrer $(p, q) < (Sn, Sm)$. Ce que l'on a par $p < Sn$.
  — Si $p = Sn$ et $q = m$, il faut montrer $\texttt{Acc}(Sn, m)$. C'est (HR2).
  — Si $p = Sm$ et $q < m$, il faut montrer $\texttt{Acc}(Sn, q)$. Par (HR2), on a (H) $\forall p', q'.(p', q') < (Sn, m) \rightarrow \texttt{Acc}(p', q')$. En appliquant (H), reste à montrer $(Sn, q) < (Sn, m)$ que l'on a par $q < m$

Notez qu'une propriété clef de la preuve est que si $n < Sm$ alors $n = m$ ou $n < m$.

Script de la preuve Comme elle devient un peu longue, on la découpe en deux lemmes :

1. `Lemma Acc_0_n : forall (n:nat), (Acc nat_lex_lt (0,n))`
2. ```
   Acc_Sn_m : forall (n:nat),
   (forall (m:nat), (Acc nat_lex_lt (n,m)))
   -> (forall (m:nat), (Acc nat_lex_lt (S n,m))).
   ```

On exprime notre propriété clef par :

```
Lemma ltS_lt_or_eq : forall (m n:nat), (m < S n) -> (m < n  m = n).
Proof.
  intros. apply le_lt_or_eq. apply lt_n_Sm_le. assumption.
Qed.
```

On montre

```
Lemma Acc_0_n : forall (n:nat), (Acc nat_lex_lt (0,n)).
Proof.
  induction n.
  - apply Acc_intro. intros. inversion H.
    + inversion H2.
    + inversion H2.
  - apply Acc_intro. intros. inversion H; subst.
    + inversion H2.
    + (* H2 -> (y1 < n  y1 = n) *)
      elim (ltS_lt_or_eq y1 n H2).
      * intro. inversion IHn. apply H1.
        apply Lt_snd. assumption.
      * intro. rewrite H0. assumption.
Qed.
```

Puis

```
Lemma Acc_Sn_m : forall (n:nat),
    (forall (m:nat), (Acc nat_lex_lt (n,m)))
    -> (forall (m:nat), (Acc nat_lex_lt (S n,m))).
Proof.
  intros. induction m.
  - apply Acc_intro. intros. inversion H0; subst.
    + elim (ltS_lt_or_eq x1 n H3).
      * intro. assert (Acc nat_lex_lt (n, y1)).
         apply H.
        inversion H2. apply H4.
        apply Lt_fst. assumption.
      * intro. rewrite H1. apply H.
```
```

```
    + inversion H3.
  - apply Acc_intro. intros. inversion H0; subst.
    + elim (ltS_lt_or_eq x1 n H3).
      * intro. assert (Acc nat_lex_lt (n, y1)).
         apply H.
        inversion H2. apply H4.
        apply Lt_fst. assumption.
      * intro. rewrite H1. apply H.
    + elim (ltS_lt_or_eq y1 m H3).
      * intros. inversion IHm. apply H2.
        apply Lt_snd. assumption.
      * intro. rewrite H1. assumption.
Qed.
```

On en déduit immédiatement

```
Lemma wf_nat_lex_lt : (well_founded nat_lex_lt).
Proof.
  unfold well_founded. destruct a.
  generalize n0. induction n.
  - apply Acc_0_n.
  - apply Acc_Sn_m. assumption.
  Qed.
```

Ordre lexicographique et terminaison On peut utiliser cette relation pour prouver la terminaison de

```
Equations ack (x y:nat) : nat by wf (x,y) nat_lex_lt :=
  ack 0 m := (S m);
  ack (S n) 0 := (ack n (S 0));
  ack (S n) (S m) := ack n (ack (S n) m).
```

La troisième équation contient deux appels récursifs imbriqués :

- dans le premier `(ack n (ack (S n) m))`, le couple d'entiers `(n,(ack (S n) m))` est plus petit que le couple `(S n, S m)` par `Lt_fst)` ;
- dans le second `(ack (S n) m)`, le couple `(S n, m)` est plus petit que `(S n, S m)` par `Lt_snd`.

L'ordre lexicographique sur le couple des tailles de ses arguments permet également de prouver la terminaison de la fonction `merge`

```
Equations merge'' (xs ys:list nat) : list nat by wf (length xs, length ys) nat_lex_lt :=
  merge'' nil ys := ys;
  merge'' xs nil := xs;
  merge'' (x::xs) (y::ys) :=
    if (x <=? y) then x::(merge'' xs (y::ys))
    else y::(merge'' (x::xs) ys).
Obligation 1.
apply Lt_fst. lia.
Qed.
Obligation 2.
apply Lt_snd. lia.
Qed.
```

La fonction `merge` come la fonction `ack` sont en fait définissable par un point fixe local car la décroissance du premier ou du second terme du couple est chaque fois trivialement structurelle (de `(S n)` à `n`, de `n::ns` à `ns`). L'imbrication de points fixes encode l'ordre lexicographique pour ces cas simples. Nous ne connaissons

pas d'exemple de «vraie» fonction définie par récurrence sur deux arguments qui réclame réellement l'ordre lexicographique [2].

Nous ne pouvons donner qu'un exemple *ad hoc* de fonction dont la terminaison nécessite une preuve par induction bien fondée sur l'ordre lexicographique des couples d'entiers :

```
Equations adhoc (x:nat) (y:nat) : nat by wf (x,y) nat_lex_lt :=
  adhoc 0 y := y;
  adhoc x 0 := x;
  adhoc x y := (adhoc (x-y) (x*y)) + (adhoc x (y-x)).
```

On ne peut utiliser ici la clause `Fixpoint` et il n'est pas évident de troouver une «taille» pour le couple d'entiers `(x,y)` qui décroisse. Le plus simple est donc d'utiliser l'ordre lexicographique : `(x-y) < x` et `(y-x) < y` dès lors que `x` et `y` ne sont pas nuls.

Mesure lexicographique En revanche, l'ordre lexicographique s'avère fort utile lorsque l'on a affaire à des manipulations non triviales de structures plus complexes, comme les arbres. Nous en donnons une illustration avec la définition d'une fonction d'extraction de la liste des étiquettes d'une arbre binaire.

Considérons les arbres binaires définis par :

```
Inductive btree (A:Set) : Set :=
  Empty : (btree A)
| Node : A -> (btree A) -> (btree A) -> (btree A).

Arguments Empty A.
Arguments Node A _ _ _.
```

On peut calculer la liste des étiquettes des arbres en utilisant l'algorithme suivant :

$$\begin{cases} \texttt{(list\_of Empy)} & = & \texttt{nil} \\ \texttt{(list\_of (Node x Empty bt)))} & = & \texttt{x::(list\_of bt)} \\ \texttt{(list\_of (Node x (Node y bt1 bt2) bt3)} & = & \texttt{(list\_of (Node x bt1 (Node y bt2 bt3)))} \end{cases}$$

Dans la troisième équation, on ne peut utiliser la taille de l'arbre comme mesure : elle ne bouge pas. Toutefois, la fonction est appelée récursivement sur un arbre dont la taille du sous-arbre gauche a diminué. Cela suggère une mesure plus fine des arbres : le couple formé de la taille de l'arbre et de la taille de son sous-arbre gauche. Cette mesure décroît pour l'ordre lexicographique dans les deux cas d'appels récursifs dans notre définition.

Définisons la donc, cette mesure :

```
Fixpoint btree_size A:Set (bt:btree A) : nat :=
  match bt with
    Empty => 0
  | Node x bt1 bt2 => S (btree_size bt1 + btree_size bt2)
  end.

Definition bt_mesure A:Set (bt:btree A) : (nat * nat) :=
  match bt with
    Empty => (0,0)
  | Node _ bt1 _ => (btree_size bt, btree_size bt1)
  end.
```

La taille arbitraire du «sous-arbre gauche» de l'arbre vide n'aura pas d'incidence sur l'usage que l'on fera de la mesure.

2. Toute suggestion en contradiction avec cette affirmation est bienvenue.

```
Equations list_of_btree A:Set (bt:btree A) : (list A) by wf (bt_mesure bt) nat_lex_lt :=
  list_of_btree Empty := nil;
  list_of_btree (Node x Empty bt) := x::(list_of_btree bt);
  list_of_btree (Node x1 (Node x2 bt1 bt2) bt3) :=
    (list_of_btree (Node x1 bt1 (Node x2 bt2 bt3))).
Obligation 1 of list_of_btree_obligations.
Proof.
  case bt.
  - simpl. apply Lt_fst. lia.
  - intros. simpl. apply Lt_fst. lia.
Qed.
Obligation 2 of list_of_btree_obligations.
Proof.
  Search (_ + S _ = S (_ + _)).
  rewrite Nat.add_succ_r.
  rewrite plus_assoc. apply Lt_snd. lia.
Qed.
```