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.

2 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 : Type) : 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 binaires.

2.1 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, les accolades qui entourent `{A:Set}` marquent un *argument implicite* . 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.
```

2.2 Fonction, spécification : étude de cas

Nous allons nous intéresser dans ce paragraphe à la fonction qui inverse l'ordre des éléments d'une liste. On dira qu'elle calcule le *miroir* de son argument.

Voici une première définition de cette fonction :

```
Fixpoint rev A:Set (xs:(list A)) : (list A) :=
  match xs with
    nil => nil
  | x::xs => (rev xs) ++ (x::nil)
  end.
```

Remarques de syntaxe : `x::xs` pour `(cons x xs)` et `(rev xs) ++ (x::nil)` pour `(app (rev xs) (cons x nil))`.

Récurrence terminale L'utilisation de la concaténation est loin d'être satisfaisant en terme de conmplexité. Il est possible de donner une meilluere définition en utilisant une fonction récursive terminale intermédiaire :

```
Fixpoint rev_append A:Set (xs:(list A)) (ys:(list A)) : (list A) :=
  match xs with
    nil => ys
```

```
  | x::xs => (rev_append xs (x::ys))
  end.
```

La fonction d'inversion des éléments d'une liste est alors donnée par :

```
Definition rev_t A:Set (xs:(list A)) : (list A) :=
  (rev_append xs nil).
```

Nous voulons montrer que `rev` et `rev_t` définissent la même fonction. C'est-à-dire que nous voulons montrer que :

```
Theorem rev_rev_t : forall (A:Set) (xs:(list A)),
    (rev_t xs) = (rev xs).
```

En développant la définition de `rev_t`, il faut montrer que `(rev_append xs nil) = (rev xs)`. On ne peut conduire la preuve de cette égalité par récurrence sur `xs`. Le blocage vient de ce que l'hypothèse de récurrence (`(rev_append xs nil) = (rev xs)` est inutilisable pour montrer `(rev_append (x::xs) nil) = (rev (x::xs))` ; c'est-à-dire, `(rev_append xs (x ++ nil)) = (rev xs) ++ (x::nil)`.

Pour débloquer la situation, il faut utiliser une propriété *plus générale* de `rev_append` vis-à-vis de `rev` :

```
Lemma rev_rev_append : forall (A:Set) (xs ys : (list A)),
    (rev_append xs ys) = (rev xs) ++ ys.
```

On prouve `forall (ys:(list A)), (rev_append xs ys) = (rev xs) ++ ys` par induction sur `xs` [1] :

- si `xs` est `nil`, l'égalité est immédiate par évaluation.
- si `xs` est `x::xs`, par hypothèse d'induction `forall (ys:(list A)), (rev_append xs ys) = (rev xs) ++ ys` et il faut montrer `(rev_append (x::xs) ys) = (rev (x::xs)) ++ ys` où `ys` est quelconque. C'est-à-dire `(rev_append xs (x::ys)) = ((rev xs) ++ (x::nil)) ++ ys`. C'est ici que la quantification de l'hypothèse de récurrence joue son rôle ; elle permet de réécrire le membre gauche de l'égalité pour obtenir : `(rev xs) ++ (x::ys)) = ((rev xs) ++ (x::nil)) ++ ys`. On utilise alors l'associativité de la concaténation pour se ramener à `(rev xs) ++ (x::ys)) = (rev xs) ++ ((x::nil)) ++ ys)`. Les termes `(x::ys)` et `((x::nil) ++ ys)` sont égalisés par évaluation.

Pour obtenir l'égalité `(rev_t xs) = (rev xs)` : on utilise le fait que `nil` est élément neutre à droite pour la concaténation (`(rev_t xs) = (rev xs) ++ nil`) ; on développe la fonction (`rev_t (rev_append xs nil) = (rev xs) ++ nil`) ; on peut aolors appliquer `rev_rev_append`.

Correction de `rev` – argument pré-formel Une manière de dire qu'une liste est le miroir d'une autre est d'utiliser la position des éléments dans les listes. Soit `xs`, `xs'` son miroir et `len` leur longueur commune. On doit avoir : `xs'[0]=xs[len-1]` ; `xs'[1]=xs[len-2]` ; etc. En général : `xs'[i]=xs[len-(i+1)]`, pour $\texttt{i} < \texttt{len}$.

La fonction `length` nous donne la longueur d'une liste et la fonction `nth` nous donne la valeur d'un élément à une position donnée (s'il existe). On peut donc exprimer la propriété attendue de `rev` de cette manière :

```
Theorem rev_correct : forall (A:Set) (xs:(list A)) (i:nat),
    (i < length xs) -> (nth i (rev xs)) = (nth (length xs - (S i)) xs).
```

Il faut bien entendu s'assurer que `(length xs)` est bien ≪leur longueur commune≫. C'est assuré car on peut montrer que :

```
Lemma rev_length : forall (A:Set) (xs:(list A)),
    (length (rev xs)) = (length xs).
```

1. La quantification `forall (ys:(list A))` est le point clé de la preuve.

(à faire en exercice)

Montrons donc que
```
forall (i:nat), (i < length xs) -> (nth i (rev xs)) = (nth (length xs - (S i)) xs)
```

par induction sur `xs` :

- si `xs` est `nil`, la formule est trivialement vérifiée car la prémisse `(i < (length nil))`, c'est-à-dire `(i < 0)`, ne l'est pas.

- si `xs` est `x::xs`, supposons (hypothèse d'induction)
```
forall (i:nat), (i < length xs) -> (nth i (rev xs)) = (nth (length xs - (S i)) xs)
```

Soit `i:nat`, supposons `(i < (length (x::xs))` et montrons
```
(nth i (rev (x::xs)) = (nth (length (x::xs) - (S i)) (x::xs))
```

c'est-à-dire (définition de `rev`)
```
(nth i ((rev xs) ++ (x::nil))) = (nth (length (x::xs) - (S i)) (x::xs))
```

L'hypothèse `(i < (length (x::xs))` donne, par définition de `length`, `(i < (S (length xs)))`, d'où `(i <= (length xs))` (arithmétique). On raisonne pas cas sur `(i <= (length xs))` :

– si `(i = (lenght xs))`, il faut montrer que
```
(nth (length xs) ((rev xs) ++ (x::nil))) = (nth (length (x::xs) - (S i)) (x::xs))
```

Or, `(nth (length xs) ((rev xs) ++ (x::nil))) = x` (il faudra le montrer). Il reste donc à montrer que
```
(nth (length (x::xs) - (S i)) (x::xs)) = x
```

Puisque `(i = (length xs))`, `((length (x::xs) - (S i)) = 0)` et `((nth 0 (x::xs)) = x)` par définition.

– si `(i < (length xs))`, par définition `(length (x::xs)) = (S (length xs))`, or, c'est une propriété de la soustraction : si `(n <= m)`, `((S m) - n = S (m - n))`. Comme `(S i) <= (length xs)`, on peut se ramener à montrer que
```
(nth i ((rev xs) ++ (x::nil))) = (nth (S (length xs - (S i))) (x::xs))
```

c'est-à-dire, par définition de `nth`,
```
(nth i ((rev xs) ++ (x::nil))) = (nth ((length xs) - (S i)) xs)
```

On peut ici utiliser l'hypothèse de récurrence et ramener ce qui reste à montrer à
```
(nth i ((rev xs) ++ (x::nil))) = (nth i (rev xs))
```

Ce qui est vrai par l'hypothèse que `(i < (length xs))` (il faudra le montrer).

Preuve formelle – avec Coq Pour mener à bien la preuve de `rev_correct`, nous avons fait appel à

1. des propriétés arithmétiques sur l'ordre et la soustraction.
2. des propriétés mettant en relation les fonctions `nth` et la concaténation.

Les propriétés arithmétiques sont disponibles dans la bibliothèque standard. En revanche, la définition de la fonction `nth` de la bibliothèque standard diffère de la notre. Nous démontrerons donc les propriétés utiles par nous-même.

Nous utiliserons les propriétés suivantes de la bibliothèque `Arith`
```
(Le) le_S_n : forall n m, S n <= S m -> n <= m.
(Lt) le_lt_or_eq : forall n m, n <= m -> n < m  n = m.
(Lt) lt_le_S : forall n m, n < m -> S n <= m
(Minus) minus_Sn_m : forall n m, m <= n -> S (n - m) = S n - m.
(Minus) minus_diag_reverse : forall n, 0 = n - n.
```

Concernant les listes, nous démontrerons :

```
  — rev_length :
                  forall (A:Set) (xs:(list A)), (length (rev xs)) = (length xs).
  — nth_len_app1 :
          forall (A:Set) (xs ys:(list A)), (nth (length xs) (xs ++ ys)) = (nth 0 ys).
  — nth_len_app2 :
    forall (A:Set) (xs ys:(list A)) (i:nat), (i < length xs) -> (nth i (xs ++ ys)) = (nth i xs).
```
(exercices)

Détaillons les tactiques utilisées dans la preuve de

```
Theorem rev_correct : forall (A:Set) (xs:(list A)) (i:nat),
    (i < length xs) -> (nth i (rev xs)) = (nth (length xs - (S i)) xs).
```

```
  (* Par induction sur xs *)
  induction xs.

2 subgoals (ID 64)

  A : Set
  ============================
  forall i : nat,
  i < length nil -> nth i (rev nil) = nth (length nil - S i) nil

subgoal 2 (ID 68) is:
 forall i : nat,
 i < length (a :: xs) ->
 nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

 (* Cas nil *)
 - intros.
 1 subgoal (ID 70)

  A : Set
  i : nat
  H : i < length nil
  ============================
  nth i (rev nil) = nth (length nil - S i) nil

  (* H est fausse *)
  inversion H.

1 subgoal (ID 68)

subgoal 1 (ID 68) is:
 forall i : nat,
 i < length (a :: xs) ->
 nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

 (* Cas x::xs *)
 - intros.
 1 subgoal (ID 92)
```

```
  A : Set
  a : A
  xs : list A
  IHxs : forall i : nat,
         i < length xs -> nth i (rev xs) = nth (length xs - S i) xs
  i : nat
  H : i < length (a :: xs)
  ============================
  nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

  (* Définition de < *)
  unfold lt in H.
  [...]
  - H : S i <= length (a :: xs)
  ============================
  nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

  assert (i <= length xs).
  2 subgoals (ID 95)

  (* on le déduit de H *)
  [...]
  - H : S i <= length (a :: xs)
  ============================
  i <= length xs

  subgoal 2 (ID 96) is:
  nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

  + apply le_S_n.
  [...]
  H : S i <= length (a :: xs)
  ============================
  S i <= S (length xs)

    assumption.
1 subgoal (ID 96)

subgoal 1 (ID 96) is:
  nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

  (* Par cas sur (i <= length xs) *)
  (* le_lt_or_eq : (i < (length xs)) \/ (i = (length xs) *)
  + elim (le_lt_or_eq i (length xs) H0).
    2 subgoals (ID 99)

  [...]
  ============================
  i < length xs ->
  nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)
```

```
subgoal 2 (ID 100) is:
 i = length xs ->
 nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

     (* 1er cas *)
     * intro.
     [...]
     H1 : i < length xs
     ============================
     nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

     simpl rev.
     [...]
     ============================
     nth i (rev xs ++ a :: nil) = nth (length (a :: xs) - S i) (a :: xs)

     (* Equation conditionnelle *)
     rewrite nth_len_app2.
     [...]
     ============================
     nth i (rev xs) = nth (length (a :: xs) - S i) (a :: xs)
     2 subgoals (ID 105)

     [...]
     ============================
     nth i (rev xs) = nth (length (a :: xs) - S i) (a :: xs)

(* Condition de nth_len_app2 *)
subgoal 2 (ID 106) is:
 i < length (rev xs)

     -- simpl length.
     [...]
     ============================
     nth i (rev xs) = nth (S (length xs) - S i) (a :: xs)

     rewrite <- minus_Sn_m.
     2 subgoals (ID 110)

     [...]
     ============================
     nth i (rev xs) = nth (S (length xs - S i)) (a :: xs)

 (* Condition de minus_Sn_m *)
 subgoal 2 (ID 111) is:
 S i <= length xs

     ++ simpl nth.
     [...]
     IHxs : forall i : nat,
            i < length xs -> nth i (rev xs) = nth (length xs - S i) xs
```

```
     ============================
     nth i (rev xs) = nth (length xs - S i) xs

     rewrite IHxs.
     2 subgoals (ID 115)

     [...]
     ============================
     nth (length xs - S i) xs = nth (length xs - S i) xs

(* Condition de IHxs *)
subgoal 2 (ID 116) is:
 i < length xs

     ** trivial.
     (* Sous but achevé, en restent 4 *)
     4 subgoals (ID 116)

subgoal 1 (ID 116) is:
 i < length xs
subgoal 2 (ID 111) is:
 S i <= length xs
subgoal 3 (ID 106) is:
 i < length (rev xs)
subgoal 4 (ID 100) is:
 i = length xs ->
 nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

     [...]
     H1 : i < length xs
     ============================
     i < length xs

     ** assumption.
     3 subgoals (ID 111)

subgoal 1 (ID 111) is:
 S i <= length xs
subgoal 2 (ID 106) is:
 i < length (rev xs)
subgoal 3 (ID 100) is:
 i = length xs ->
 nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

     [...]
     H0 : i <= length xs
     H1 : i < length xs
     ============================
     S i <= length xs

     ++ apply lt_le_S.
```

```
     [...]
     ============================
     i < length xs

        assumption.
     2 subgoals (ID 106)

subgoal 1 (ID 106) is:
 i < length (rev xs)
subgoal 2 (ID 100) is:
 i = length xs ->
 nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

     [...]
     ============================
     i < length (rev xs)

     -- rewrite rev_length.
     [...]
     H1 : i < length xs
     ============================
     i < length xs

        assumption.
        1 subgoal (ID 100)

subgoal 1 (ID 100) is:
 i = length xs ->
 nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

   * intro.
   [...]
   H1 : i = length xs
   ============================
   nth i (rev (a :: xs)) = nth (length (a :: xs) - S i) (a :: xs)

     rewrite H1.
     [...]
     ============================
     nth (length xs) (rev (a :: xs)) =
     nth (length (a :: xs) - S (length xs)) (a :: xs)

     simpl minus.
     [...]
     ============================
     nth (length xs) (rev (a :: xs)) = nth (length xs - length xs) (a :: xs)

     rewrite <- minus_diag_reverse.
     [...]
     ============================
     nth (length xs) (rev (a :: xs)) = nth 0 (a :: xs)
```

```
simpl rev.
[...]
============================
nth (length xs) (rev xs ++ a :: nil) = nth 0 (a :: xs)

rewrite <- rev_length.
[...]
============================
nth (length (rev xs)) (rev xs ++ a :: nil) = nth 0 (a :: xs)

rewrite nth_len_app1.
[...]
============================
nth 0 (a :: nil) = nth 0 (a :: xs)

trivial.

No more subgoals.
```

Script formaté de la preuve

```
Theorem rev_correct : forall (A:Set) (xs:(list A)) (i:nat),
    (i < length xs) -> (nth i (rev xs)) = (nth (length xs - (S i)) xs).
Proof.
  induction xs.
  - intros. inversion H.
  - intros. unfold lt in H. assert (i <= length xs).
    + apply le_S_n. assumption.
    + elim (le_lt_or_eq i (length xs) H0).
      * intro. simpl rev. rewrite nth_len_app2.
        -- simpl length. rewrite <- minus_Sn_m.
           ++ simpl nth. rewrite IHxs.
              ** trivial.
              ** assumption.
           ++ apply lt_le_S. assumption.
        -- rewrite rev_length. assumption.
      * intro. rewrite H1. simpl minus. rewrite <- minus_diag_reverse.
        simpl rev. rewrite <- rev_length. rewrite nth_len_app1. trivial.
Qed.
```