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.

3 Prédicats, relations inductives

On a vu l'utilisation de la définition de *types inductifs* pour définir des ensembles de «valeurs» : les entiers naturels, les listes, etc. Ces types (`nat`, `list`) appartiennent eux-mêmes au type `Set : 0:nat:Set`. Cette même clause peut servir à définir des *valeurs de vérité* qui sont les éléments du type `Prop`.

Par exemple, on peut définir inductivement les conditions de l'appartenance à une liste :

1. x appartient à la liste qui commence par x, c'est-à-dire : x appartient à $x :: xs$ pour tout x et tout xs.
2. si x appartient à la liste xs alors x appartient également à la liste $y :: xs$, pour tout y.

Appelons `In` la relation binaire d'appartenance définie sur `A`, le type des éléments, et `(list A)`. La définition ci-dessus s'exprime à l'aide des deux formules :

1. `forall (x:A) (xs:(list A)), (In x (x::xs))`
2. `forall (x:A) (xs:(list A)), (In x xs) -> forall (y:A), (In x (y::xs))`

Ces formules sont de type `Prop`. On peut utiliser ces deux formules pour définir un type inductif qui ne donnera pas des «valeurs» dans le type `Set`, mais des valeurs de vérité du type `Prop` :

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

On utilise les *constructeurs* `In_hd` et `In_tl` comme les noms des *axiomes* qui définissent la relation `In` :

```
Goal (In 3 (1::2::3::4::nil)).
Proof.
  apply In_tl.
  apply In_tl.
  apply In_hd.
Qed.
```

Montrons :

```
Lemma In_tl_gen : forall (A:Set) (z:A) (xs ys:(list A)),
    (In z ys) -> (In z (xs ++ ys)).
```

Par induction sur `xs`

- si `xs` est `nil`. Il faut montrer `(In z ys) -> (In z (nil ++ ys))`, c'erst-à-dire `(In z ys) -> (In z ys)`, ce qui est trivial.

- si `xs` est `x::xs`, par hypothèse d'induction `forall (ys:(list A)), (In z ys) -> (In z (xs ++ ys))`. Supposons `(In z ys)`, il faut montrer `(In z ((x::xs) ++ ys))`, c'est-à-dire `(In z (x::(xs ++ ys)))`. Par `In_tl`, reste à montrer `(In z (xs ++ ys))`. Ce que l'on a par hypothèse d'induction et `(In z ys)`.

Script :

```
Lemma In_tl_gen : forall (A:Set) (z:A) (xs ys:(list A)),
    (In z ys) -> (In z (xs ++ ys)).
Proof.
  induction xs.
  - intros. assumption.
  - intros. simpl. apply In_tl.
    apply IHxs. assumption.
Qed.
```

3.1 Inversion

Nous venons de voir comment les constructeurs `In_hd` et `In_tl` du prédicat `In` permettent d'établir des énoncés de la forme `(In x xs)`. On peut dire que `In_tl` ou `In_hd` impliquent `In`. Mais la définition inductive de `In` nous donne plus : `In` implique `In_hd` ou `In_tl`. C'est-à-dire que si l'on a `(In x xs)` alors *nécessairement*
- `xs` et `x::xs'` (constructeur `In_hd`)
- ou `xs` est `y::xs'` et `(In x xs')` (constructeur `In_tl`).

Les formules qui premettent de passer de `In` aux constructeurs sont appelés *lemmes d'inversion*. Nous allons illustrer leur utilisation pour mener à bien une preuve.

Montrons :

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

Par induction sur `xs`

- si `xs` est `nil`, supposons `(In z (nill ++ ys))`, c'est-à-dire `(In z ys)` et montrons `(In z nil) \/ (In z ys)`. Ce qui est donné par l'hypothèse `(In z ys)`.

- si `xs` est `x::xs`,
par hypothèse d'induction, `forall (ys:(list A)), (In z (xs ++ ys)) -> (In z xs) \/ (In z ys)`.
Supposons `(In z ((x::xs) ++ ys))`, c'est-à-dire `(In z (x::(xs ++ ys)))`
et montrons `(In z (x::xs)) \/ (In z ys)`.

Par l'hypothèse `(In z (x::(xs ++ ys)))` il y a deux possibilités (inversion) :
- ou bien `z` et `x` sont égaux. Dans ce cas, il faut montrer que `(In z (z::xs)) \/ (In z ys)`. Ce que l'on a car `(In z (z::xs))` par `In_hd`.
- ou bien `(In z (xs ++ ys))`. Dans ce cas, par hypothèse d'induction, on a que `(In z xs) \/ (In z ys)`.
  - si `(In z xs)`, on obtient `(In z (x::xs))` par `In_tl`, d'où `(In z (x::xs)) \/ (In z ys)`.
  - si `(In z ys)` on a aussi `(In z (x::xs)) \/ (In z ys)`.

Script :

```
Lemma In_app : forall (A:Set) (z:A) (xs ys:(list A)),
    (In z (xs ++ ys)) -> (In z xs) \/ (In z ys).
Proof.
  induction xs.
  - intros. right. assumption.
  - intros.
    inversion H.
    + left. apply In_hd.
    + assert (In z xs \/ In z ys).
      * apply IHxs. assumption.
      * elim H4.
        -- intro. left. apply In_tl. assumption.
        -- intro. right. assumption.
Qed.
```

Les deux constructeurs `In_hd` et `In_hd` portent sur des listes non vides. Puisque seuls ces deux constructeurs donnent la validité des énoncés de la forme `(In x xs)`, la liste `xs` est nécessairelment non vide. Cela nous donne que `(In x nil)` est nécessairement un énoncé faux. Intuitivement, si on suppose `(In x nil)` alors on peut en déduire `False`, et donc n'importe quoi. C'est une autre forme de lemme d'inversion.

Utilisons la pour prouver :

```
Lemma In_hd_gen : forall (A:Set) (z:A) (xs ys:(list A)),
    (In z xs) -> (In z (xs ++ ys)).
```

Par induction sur `xs`

- si `xs` est `nil`, il faut montrer que `(In z nil) -> (In z (nil ++ ys))`, ce qui est immédiat car `(In z nil)` est faux.

- si `xs` est `x::xs`, par hypothèse d'induction `(In z xs) -> (In z (xs ++ ys))`. Supposon `(In z (x::xs))` et montrons `(In z ((x::xs) ++ ys))`, c'est-à-dire, `(In z (x::(xs ++ ys)))`.

Par définition de `In` (inversion) et `(In z (x::xs))` on a deux possibilité

- ou `z` et `x` sont identiques, il faut alors montrer que `(In z (z::(xs ++ ys)))`, ce qui est donné par `In_hd` ;
- ou `(In z xs)`. Dans ce cas, en appliquant `In_tl`, il suffit de montrer `(In z (xs ++ ys))`. Ce qui nous est donné par hypothèse d'induction.

Script

```
Lemma In_hd_gen : forall (A:Set) (z:A) (xs ys:(list A)),
    (In z xs) -> (In z (xs ++ ys)).
Proof.
  induction xs.
  - intros. inversion H.
  - intros. inversion H.
    + apply In_hd.
    + simpl. apply In_tl.
      apply IHxs. assumption.
Qed.
```

3.2 Induction

Tout type inductif s'accompagne d'un principe d'induction. Les types inductifs dans le type `Prop` n'échappent pas à la règles. Pour le prédicat `In`, on a le principe suivant : pour toute propriété `P: A -> (list A) -> Prop`

1. si `(P x (x::xs))`
2. si `(In x xs) -> (P x xs) -> (P x (y::xs))`
3. alors pour tout `x`, `xs`, `(P x xs)`

On peut utiliser ce principe pour donner une autre preuve de

```
Lemma In_hd_gen : forall (A:Set) (z:A) (xs ys:(list A)),
    (In z xs) -> (In z (xs ++ ys)).
```

Supposons `(In z xs)` et montrons `(In z (xs ++ ys))` par induction sur `(In z xs)` (avec `(P` α β`)` $\equiv$ `(In` α `(`β `++ ys))`) :

1. On doit montrer `(In x ((x::xs) ++ ys))`, ce qui est immédiat pas simplification et `In_hd`.
2. On suppose `(In x (xs ++ ys))` (hypothèse d'induction) et on doit montrer `(In x ((y::xs) ++ ys))`, ce qui vient par simplification, `In_tl` et hypothèse d'induction.

Script :

```
Lemma In_hd_gen : forall (A:Set) (z:A) (xs ys:(list A)),
    (In z xs) -> (In z (xs ++ ys)).
Proof.
  intros. induction H.
  - simpl. apply In_hd.
  - simpl. apply In_tl. assumption.
Qed.
```

Par rapport à notre première version de la preuve de `In_hd_gen` on a fait ici l'économie de `inversion` ; en particulier pour éliminer l'hypothèse absurde `(In z nil)`.

3.3 Fonction propositionnelle *vs* fonction booléenne – décidabilité

Résumons ce que nous avons sur la valeur de vérité de `(In z xs)`
- si la liste et vide `(In z nil)` est faux
- sinon, la liste `xs` est de la forme `x::xs'` et
  - si `z`=`x` alors `(In z xs)` est vrai
  - `(In z xs)` a la même valeur de vérité que `(In z xs')`

On peut modifier un peu le cas de la liste non vide pour formuler
- si la liste et vide `(In z nil)` est faux
- sinon, la liste `xs` est de la forme `x::xs'` et `(In z xs)` a la même valeur de vérité que `(z=x) \/ (In z xs')`

Le langage de Coq nous fournit deux noms pour les valeurs de vérité : `False` et `True` qui sont de type `Prop`.

On peut transcrire en Coq notre dernière définition de l'appartenance à une liste comme une *fonction propositionnelle* (récursive) qui calcule une valeur de vérité. Le haut niveau du langage de Coq le permet :

```
Fixpoint in_p A:Set (x:A) (xs:(list A)) : Prop :=
  match xs with
    nil => False
  | x1::xs => (x=x1) \/ (in_p x xs)
  end.
```

Montrons que le prédicat `In` et la fonction `in_p` définissent la même fonction propositionnelle. C'est-à-dire :

```
Lemma In_eq_in_p : forall (A:Set) (x:A) (xs:(list A)),
    (In x xs) <-> (in_p x xs).
```

On montre les deux sens de l'équivalence.

- Supposons `(In x xs)` et montrons `(in_p x xs)`. Par induction sur `(In x xs)`.

– il faut montrer `(in_p x (x::xs))`, c'est-à-dire `(x = x) \/ (in_p x xs))`. Ce qui est donné pas réfléxivité `(x = x )`.

– on a `(in_p x xs)` comme hypothèse d'induction et il faut montrer `(in_p x (x::xs))`, c'est-à-dire `(x = x) \/ (in_p x xs))`. Ce qui est donné par hypothèse d'induction.

- montrons `(in_p x xs) -> (In x xs)` par induction sur `xs`.

– si `xs` est `nil`, il faut montrer `(in_p x nil) -> (In x nil)`. Ce qui est trivialement vrai car la prémisse se simplifie en `False`.

– si `xs` est `x::xs`, on a par hypothèse d'induction que `(in_p x xs) -> (In x xs)`. Supposons `(in_p x (a::xs))` et montrons `(In x (a::xs))`. Par définition, l'hypothèse `(in_p x (a::xs))` est la disjonction `(x = a) \/ (in_p x xs)`. On raisonne par cas :

— si `(x = a)`, il faut montrer `(In a (a::xs))`. Ce qui est donné par `In_hd`.

— si `(in_p x xs)`, Par `In_tl`, il suffit de montrer que `(In x xs)`. Ce que l'on a en combinant l'hypothèse d'induction et l'hypothèse `(in_p x xs)`.

Voici le script de cette preuve :

```
Lemma In_eq_in_p : forall (A:Set) (x:A) (xs:(list A)),
    (In x xs) <-> (in_p x xs).
Proof.
  intros. split.
  - intro. induction H.
    + simpl. left. reflexivity.
    + simpl. right. assumption.
  - induction xs.
    + intro. exfalso. assumption.
    + intro. elim H.
      * intro. rewrite H0. apply In_hd.
      * intro. apply In_tl. apply IHxs. assumption.
Qed.
```

Une fonction propositionnelle n'est cependant pas tout à fait une fonction comme nous les avons vues. En particulier, on n'obtiendra pas par simple calcul la valeur de vérité de l'application de la fonction propositionnelle `in_p` à des valeurs :

```
Goal (in_p 3 (1::2::3::4::nil)).
Proof.
  simpl. right. right. left. reflexivity.
Qed.
```

Il faut ajouter au processus d'évaluation (`simpl`) quelques éléments de preuve.

On est alors tenté de glisser du domaine des valeurs de vérités au domaine des *valeurs booléennes*.

```
Fixpoint in_b A:Set (x:A) (xs:(list A)) : bool :=
  match xs with
```

```
    nil => false
  | x1::xs => (x=x1) || (in_b x xs)
  end.
```

Mais cette tentative échoue :

```
The term "x = x1" has type "Prop" while it is expected to have type "bool".
```

L'égalité dénotée par le symbole = est en effet une relation logique (relation propositionnelle) et pas une fonction booléenne. Il nous faudrait son pendant du côté des fonctions booléennes ; une fonction, disons `eqb`, de type `forall (A:Set), A -> A -> bool` telle que `(eqb x y)=true` si et seulement si `(x = y)`, c'est-à-dire, si et seulement si `(x = y)` *est prouvable*.

Une telle fonction n'existe pas. Pour le dire rapidement, si une telle fonction existait, elle contredirait *l'indécidabilité du problème de l'arrêt*.

On peut toutefois approcher ce que l'on souhaite en posant :

```
Fixpoint in_b A:Set (eqb: A -> A -> bool) (x:A) (xs:(list A)) : bool :=
  match xs with
    nil => false
  | a::xs => (eqb x a) || (in_b eqb x xs)
  end.
```

Même si rien ne garantit ici que la fonction donnée en paramètre `eqb` réalise l'égalité, on peut montrer qu'il est possible de faire de `in_b` l'usage attendu :

```
Lemma in_b_In : forall (A:Set) (eqb : A -> A -> bool),
      (forall (x y:A), ((eqb x y)=true <-> x = y))
      -> forall (x:A) (xs:(list A)), (in_b eqb x xs)=true -> (In x xs).
```

Supposons que `(forall (x y:A), ((eqb x y)=true <-> x = y))` (hypothèse d'adéquation de `eqb`), on montre `(in_b eqb x xs)=true -> (In x xs)` par induction sur `xs` :

- si `xs` est `nil`, il faut montrer que `(in_b eqb x nil)=true -> (In x nil)`, c'est-à-dire, par définition de `in_b`, `false=true -> (In x nil)`. Ce qui est trivialement vrai car la prémisse `false=true` est fausse.

- si `xs` est `x::xs`, l'hypothèse d'induction est `(in_b eqb x xs) = true -> (In x xs)`, supposons `(in_b eqb x (a :: xs)) = true`, c'est-à-dire, `(eqb x a || in_b eqb x xs) = true`, il faut montrer `(In x (a :: xs))`. On raisonne par cas sur `(eqb x a)`

– si `(eqb x a) = true`, on tire de l'hypothèse d'adéquation de `eqb` que `x = a`. Il faut donc montrer que `(In a (a::xs))`. ce que nous donne `In_hd`.

– si `(eqb x a) = false`, l'hypothèse `(eqb x a || in_b eqb x xs) = true` devient `(false || in_b eqb x xs) = true`, c'est-à-dire `(in_b eqb x xs = true)`. En appliquant `in_tl`, il suffit de montrer `(In x xs)`. Ce que l'on a par hypothèse d'induction et `(in_b eqb x xs = true)`.

Script de la preuve :

```
Lemma in_b_In : forall (A:Set) (eqb : A -> A -> bool),
      (forall (x y:A), ((eqb x y)=true <-> x = y))
      -> forall (x:A) (xs:(list A)), (in_b eqb x xs)=true -> (In x xs).
Proof.
  induction xs.
  - simpl. intro. discriminate.
  - simpl. case_eq (eqb x a).
    + simpl. intros. assert (x=a).
      * elim (H x a). intros.
```

```
        rewrite (H2 H0). reflexivity.
      * rewrite H2. apply In_hd.
    + simpl. intros. apply In_tl.
      apply IHxs. assumption.
Qed.
```

Si l'égalité n'est pas décidable en général, elle l'est sur les structures de données simples comme le sont les valeurs valeurs du type `nat`. La bibliothèque standard fourni la fonction `Nat.eqb` (syntaxe infixe `=?`) et le théorème

```
Lemma eqb_eq n m : (n =? m) = true <-> n = m.
```

On peut alors définir l'appartenance dans le cas particulier des listes d'entiers en appliquant la fonction générale `in_b` à la fonction d'égalité spécifique aux entiers :

```
Definition nat_in_b : nat -> (list nat) -> bool := (in_b Nat.eqb).
```

Et on déduit la correction de cette fonction par application du lemme général (script) :

```
Lemma nat_in_b_In : forall (x:nat) (xs:(list nat)),
    (nat_in_b x xs) = true -> (In x xs).
Proof.
  apply in_b_In. exact Nat.eqb_eq.
Qed.
```

L'égalité polymorphe de Ocaml Le langage Ocaml fournit un opérateur *polymorphe* de test de l'égalité : `(=) : 'a -> 'a -> bool`. C'est une facilité offerte au programmeur et non un contre exemple à l'impossibilité d'avoir une fonction booléenne générale pour l'égalité.

En effet, l'égalité booléenne de Ocaml ne s'applique qu'entre *valeurs non fonctionnelles*. Tenter de comparer deux fonctions provoque une erreur d'exécution :

```
# succ = (fun n -> n+1);;
Exception: Invalid_argument "compare: functional value".
```

De même manière, l'utilisation de `List.mem` passe l'épreuve du typage, mais échoue à l'évaluation :

```
 List.mem (fun x -> x+1) [succ];;
Exception: Invalid_argument "compare: functional value".
```

La possibilité de tester, de manière correcte, l'égalité entre valeurs non fonctionnelles repose sur la repésentation mémoire de ces valeurs. Cette astuce n'est pas facilement réalisable avec le langage très généraliste de Coq.