UPMC/MASTER/INFO/STL/SVP
Spécification et Vérification de Programmes

P. Manoury F. Peschanski

2014

Table des matières

1 Relations inductives

On définit la fonction booléenne suivante :

```
Variable eqb : forall (A:Set), A -> A -> bool.
Axiom eqb_refl: forall (A:Set) (x y : A), (eqb A x y)=true.
Fixpoint is_prefix {A:Set} (xs ys : list A) : bool :=
  match (xs, ys) with
    (nil, _) => true
  | (_, nil) => false
  | (cons x xs, cons y ys) => (andb (eqb A x y) (is_prefix xs ys))
  end.
```

C'est une fonction qui est sensée donner `true` lorsque la liste `xs` est *préfixe* de la liste `ys` et `false` sinon.

1.1 Fonctions propositionnelles

L'expression `(is_prefix xs ys)=true` définit une relation entre les listes `xs` et `ys` dont on peut montrer la validité ou non selon les valeurs de `xs` et `ys`. Par exemple, on peut montrer que

```
Theorem is_prefix_app : forall (A:Set) (xs ys:(list A)),
   (is_prefix xs (app xs ys))=true.
Proof.
induction xs.
 - auto.
 - intro.
   simpl.
   rewrite eqb_refl.
   rewrite IHxs.
   trivial.
```

On peut souhaiter se débarasser, pour définir la notion de préfixe, du passage par les booléens et définir à la place directement la *valeur de vérité* logique de l'application `(is_prefix xs ys)`. C'est-à-dire, plutôt que d'avoir une fonction booléenne `is_prefix` de type `(list A) -> (list A) -> bool` ; avoir directement une *relation* logique `is_prefix` de type `(list A) -> (list A) -> Prop`.

Dans un système formel comme le système Coq, il est possible de définir des *fonctions propositionnelles* de cette sorte, à la manière dont on définit des fonctions booléennes. En effet, sont *définis* dans le système Coq deux objets, de type `Prop` qui représente la valeur de vérité ≪vrai≫ et la valeur de vérité ≪faux≫ ; respectivement, `True` et `False`. Ceci permet de donner la version propositionnelle de `is_prefix` en remplaçant également l'égalité et la conjonction booléennes par leur correspondants propositinnels :

```
Fixpoint is_prefix_prop (xs:(list A)) (ys:(list A)) : Prop :=
  match xs, ys with
    nil, _ => True
  | (cons x xs), nil => False
  | (cons x xs), (cons y ys) => (x=y) /\ (is_prefix_prop xs ys)
  end.
```

Il est très facile de montrer que cette définition est cohérente avec la notion de ≪commence par≫ :

```
Theorem is_prefix_prop_app : forall (xs ys:(list A)),
   (is_prefix_prop xs (app xs ys)).
induction xs.
```

```
  simpl; trivial.
  simpl; split; auto.
Qed.
```

Mais cette définition contient encore une petite scorie : le cas où la valeur de vérité de la fonction est `False`. Si l'on veut prouver, par exemple

```
Theorem is_prefix_prop_length : forall (xs ys : (list A)),
  (is_prefix_prop xs ys) -> (le (length xs) (length ys)).
```

il faut prendre ce cas en considération en destructurant `ys` :

```
Proof.
induction xs.
  auto with arith.
  destruct ys.
    simpl; tauto.
    simpl; intro; apply le_n_S; apply IHxs; tauto.
Qed.
```

Or, on aimerait pouvoir ne pas avoir à considérer un tel cas, car, intuitivement, si `(is_prefix (cons a xs) ys)` est vérifiée alors, *nécessairement*, il faut que `ys` soit un `cons` dont le premier éléments est égal à `a`.

En bref, on aimerait pouvoir raisonner avec une définition de `is_prefix` qui ne nous donne que les cas de `xs` et `ys` qui satisfont la relation. Informellement, on voudrait définir simplement la relation `is_prefix_rel` de type `(list A) -> (list A) -> Prop` de la manière suivante :

1. `nil` est préfixe de toute liste.
2. si `xs` est préfixe de `ys` alors, pour tout `z`, `(cons z xs)` est préfixe de `(cons z ys)`.

C'est une définition qui ne considère plus que les deux cas pertinents. Elle est inductive dans la 2ème cas.

Le système Coq autorise de telles définitions avec la clause `Inductive` :

```
Inductive is_prefix_rel : (list A) -> (list A) -> Prop :=
  is_prefix_nil : forall (ys:(list A)), (is_prefix_rel nil ys)
| is_prefix_cons : forall (z:A) (xs ys:(list A)),
     (is_prefix_rel xs ys) -> (is_prefix_rel (cons z xs) (cons z ys)).
```

Cette définition nous dit qu'il n'existe que deux possibilités pour vérifier la relation «être préfixe» nommées `is_prefix_nil` et `is_prefix_cons`. Pour montrer qu'une liste est préfixe d'une autre, il faudra nécessairement en passer par l'*application* de l'un de ces deux cas ou de leur combinaison. Par exemple :

```
Variable x1 x2 : A.

Fact is_prefix_ex : (is_prefix_rel (cons x1 nil) (cons x1 (cons x2 nil))).
apply is_prefix_cons.
apply is_prefix_nil.
Qed.
```

Les cas nommés `is_prefix_nil` et `is_prefix_cons` sont les seuls moyens donnés pour *construire* la validité de la relation. Réciproquement, si deux listes sont dans la relation alors, c'est qu'elles satisfont l'un et l'autre cas. Mieux encore si l'on suppose que deux listes `xs` et `ys` sont dans la relation (*i.e.* , si l'on suppose `(is_prefix_rel xs ys)`) et que l'on veut montrer une autre propriété à leur propos, il suffit de considérer les cas où

– `xs` est la liste `nil` et vérifier que la propriété voulue est satisfaite par `nil` et un `ys` quelconque ;

– on a deux listes `xs` et `ys` tels que `(is_prefix_rel xs ys)` et `xs` et `ys` satisfont la propriété voulue et on vérifie que la propriété est encore satisfaite par `(cons z xs)` et `(cons z ys)` avec `z` quelconque.

En d'autres termes, on a un *principe d'induction* sur les listes qui satisfont la relation `is_prefix_rel`. Le voici :

```
forall P : list A -> list A -> Prop,
  (forall (ys : list A), (P nil ys)) ->
  (forall (z : A) (xs ys : list A),
     (is_prefix_rel xs ys) -> (P xs ys) -> (P (cons z xs) (cons z ys))
  -> forall (xs ys : list A), (is_prefix_rel xs ys) -> (P xs ys)
```

On peut utiiser ce principe pour montrer :

```
Theorem is_prefix_rel_length : forall (xs ys:(list A)),
  (is_prefix_rel xs ys) -> (le (length xs) (length ys)).
intros; induction H.
  trivial with arith.
  simpl; auto with arith.
Qed.
```

C'est la commande de preuve `induction H` qui réalise l'application du principe d'induction de `is_prefix_rel`. On a ce moment l'hypothèse `H : (is_prefix_rel xs ys)`.

Il y a une autre manière d'utiliser une hypothèse de type inductif. Considérons

```
Theorem prefix_nil : forall (A:Set) (xs : list A),
  (is_prefix_rel xs nil) -> xs = nil.
```

Si l'on suppose `(is_prefix_rel xs nil)`, d'après la définition du prédicat `is_prefix_rel`, seul le constructeur `is_prefix_nil` permet de satisfaire cette hypothèse. Or, ce constructeur impose que le premier argument du prédicat (ici, `xs`) soit égal à `nil`.

Ce raisonnement est réalisé par la tactique `inversion` qui, comme son nom l'indique, donne les conditions suffisantes à la satisfaction d'un prédicat appliqué à des termes donnés. Illustrons l'usage de cette tactique avec l'exemple ci-dessous :

```
Theorem prefix_nil : forall (A:Set) (xs : list A),
  (is_prefix_rel xs nil) -> xs = nil.
intros.

  A : Set
  xs : list A
  H : is_prefix_rel xs nil
  ============================
   xs = nil

inversion H.

  A : Set
  xs : list A
  H : is_prefix_rel xs nil
  ys : list A
  H0 : nil = xs
  H1 : ys = nil
  ============================
```

```
   nil = nil

trivial.
Qed.
```

Exercice : en utilisant `induction` et `inversion`, prouver

```
Theorem prefix_trans : forall (A:Set) (xs ys zs : list A),
  (is_prefix_rel xs ys) -> (is_prefix_rel ys zs) -> (is_prefix_rel xs zs).
```

1.2 Relations fonctionnelles

Il est également possible d'exprimer une *fonction* sous la forme d'une relation reliant les valeurs des ses entrées aux valeurs de ses sorties. Dans e cas, pour exprimer une fonction n-aire, on définit une relation $n+1$-aire dont l'argument supplémentaire représente la fonction. Pour prendre un exemple simple, en s'inspirant de la définition récursive usuelle de la concaténation de deux listes, on peut définir la relation ternaire `concat : (list A) -> (list A) -> (list A) -> Prop` telle que (`concat` xs ys zs) est satisfaite si et seulement si (`app` xs ys) $= zs$:

```
Inductive concat : (list A) -> (list A) -> (list A) -> Prop :=
  concat_nil : forall (ys:(list A)), (concat nil ys ys)
| concat_cons : forall (xs ys zs : (list A)) (z:A),
    (concat xs ys zs) -> (concat (cons z xs) ys (cons z zs)).
```

Programmtion logique Cette manière de voir les définitions de fonctions comme des relations se retrouve dans le paradigme de la *programmation logique* dont une réalisation est le langage *prolog* [BIB: prolog].

Le ≪mécanisme d'évaluation≫ de prolog ne repose pas sur la notion de réduction, mais sur celle de preuve. En prologue, prouver, c'est calculer. Voici l'idée sommaire de cette correspondance : pour eux listes concrètes, disons `(cons 1 (cons 2 nil))` et `(cons 3 (cons 4 (cons 5 nil)))`, la définition de `concat` nous premet de connaître la valeur de la concaténation ; pour cela, il suffit de connaître le zs tel que `(concat (cons 1 (cons 2 nil)) (cons 3 (cons 4 (cons 5 nil)))` zs) est vérifiée. Une preuve de la formule $\exists zs$: `(list nat)`, `(concat (cons 1 (cons 2 nil)) (cons 3 (cons 4 (cons 5 nil)))`zs)).

On peut réaliser une telle preuve dans le système Coq :

```
Fact concat_12345 : exists zs:(list nat),
  (concat (cons 1 (cons 2 nil))
          (cons 3 (cons 4 (cons 5 nil)))
  zs).
eexists.
apply concat_cons.
apply concat_cons.
apply concat_nil.
Qed.
```

La tactique `eexists` s'applique à un but existentiel. Elle remplace la variable quantifiée existentiellement par une *variable existentielle* (une inconnue) dont il faudra donner la valeur dans le reste du script de preuve. Ici, cette valeur est automatiquement déduite des application des constructeurs de `concat`. Une fois la preuve achevée, on peut voir cette valeur en demandant au système d'afficher la preuve construite :

```
Coq < Print concat_12345.
concat_12345 =
ex_intro
```

```
  (fun zs : list nat =>
   concat (1 :: (2 :: nil)%list) (3 :: (4 :: 5 :: nil)%list) zs)
  (1 :: 2 :: 3 :: 4 :: 5 :: nil)%list
  (concat_cons (2 :: nil) (3 :: (4 :: 5 :: nil)%list)
     (2 :: (3 :: 4 :: 5 :: nil)%list) 1
     (concat_cons nil (3 :: (4 :: 5 :: nil)%list) (3 :: (4 :: 5 :: nil)%list)
        2 (concat_nil (3 :: (4 :: 5 :: nil)%list))))
     : exists zs : list nat,
         concat (1 :: (2 :: nil)%list) (3 :: (4 :: 5 :: nil)%list) zs

Coq <
```

Nous ne détaillerons pas plus avant ici le contenu de cet affichage [1]. Nous attirons simplement l'attention du lecteur sur sa 6ème ligne qui donne la valeur du zs recherché : `(1 :: 2 :: 3 :: 4 :: 5 :: nil)`.

Fonctions partielles La vision ≪relation fonctionelle≫ permet de poser le définition de fonctions que l'exigence de terminaison liée aux constructions `Fixpoint` ou `Function` de permet pas.

La fonction qui donne le n-ième élément d'une liste n'est pas partout définie. On tourne cette difficulté en posant la relation fonctionnelle :

```
Inductive list_get {A:Set} : nat -> (list A) -> A -> Prop :=
  get_0 : forall (x:A) (xs:(list A)), (list_get 0 (cons x xs) x)
| get_S : forall (i:nat) (x:A) (xs:(list A)),
    (list_get i xs x) -> forall (z:A), (list_get (S i) (cons z xs) x).
```

On peut utiliser cette ≪fonction≫ pour vérifier quelques faits sur les listes. Par exemple :

```
Lemma get_last : forall (x:A) (xs:(list A)),
  (list_get (length xs) (app xs (cons x nil)) x).
induction xs.

  apply get_0.

  simpl length.
  simpl app.
  apply get_S.
  assumption.
Qed.
```

Pourquoi la définition inductive de `list_get` est-elle acceptée par le système Coq alors qu'une définition telle que

```
Fixpoint list_get_fun (i:nat) (xs:(list A)) :=
  match i, xs with
    0, (cons x xs) => x
  | (S i), (cons x xs) => (list_get_fun i xs)
  end.
```

ne l'est pas :

```
Error: Non exhaustive pattern-matching: no clause found for patterns 0 nil
```

1. Pour en dire un simple mot : le système Coq est basé sur une *logique constructive* dont une particularité est que les preuves d'une formule existentielle sont formée de la valeur recherchée et de la preuve que celle-ci satisfait la forumles quantifiée.

La réponse est simple : `list_get_fun` n'est pas définie pour le cas de la liste vide, donc l'expression `(list_get_fun 0 nil)` n'a pas de valeur et du coup pas de type. Ce que ne peut accepter le système Coq.
En revanche, l'expression `(list_get 0 nil x)` est typable, dès lors que `x` est typable. Elle a une ≪valeur≫, la valeur de vérité `False`. On montre :

```
Theorem not_get_nil : forall (n:nat) (x:A),
  not (list_get n nil x).
intros.
intro.
inversion H.
Qed.
```

Nota : ici, la tacique `inversion` conclue à l'impossibilité de satisfaire `(list_get n nil x)` car aucune des conclusions des types des constructeurs ne correspond au cas voulu.

Relations et fonctions On pourrait envisager, sur le même modèle, de spécifier la ≪fonction duale≫ : celle qui calcule la position d'un élément d'une liste. On écrirait :

```
Inductive index {A:Set} : A -> (list A) -> nat -> Prop :=
  index_0 : forall (x:A) (xs:(list A)), (index x (cons x xs) 0)
| index_S : forall (x z:A) (xs : (list A)) (i:nat),
     (index x xs i) -> (index x (cons z xs) (S i)).
```

Mais attention, cette relation inductive n'est *pas une fonction* : il peut exister plusieurs `i` tels que `(index x xs i)` soit vérifié : `(index x (cons x (cons x nil)) 0)` et `(index x (cons x (cons x nil)) 1)`.

Looping recursion Certaines fonctions ne sont pas partout définies non pas parce que leur définition est incomplète mais, en quelque sorte, elle l'est trop. C'est-à-dire, qu'elle est définie, au sens où le mécanisme d'évaluation peut *progresser*, sur des cas pour lesquels ils n'existent pas de valeurs. Cette formulation alambiquée cache le bête cas des fonctions qui ≪bouclent≫ sur certaines valeurs d'entrées. Par exemple, la fonction qui reccherche un chemin entre deux sommets d'un graphe :

```
Inductive path : (list (A*A)) -> A -> A -> (list A) -> Prop :=
   path_edge : forall (g:(list (A*A))) (x1 x2:A),
      (In (x1,x2) g) -> (path g x1 x2 (cons x1 (cons x2 nil)))
 | path_rec : forall (g:(list (A*A))) (x1 x2 x3:A) (p:(list A)),
      (In (x1,x2) g) -> (path g x2 x3 p) -> (path g x1 x2 (cons x1 p)).
```

Le graphe est ici simplement représenté par la liste de ses arêtes.

La définition proposée est *correcte* au sens où, si il existe un chemin `p` entre deux sommets `x1` et `x2` dans un graphe `g`, alors il existe une preuve de `(path g x1 x2 p)`, donc un moyen de calculer ce chemin (*cf.* . prolog). Mais cette définition n'est pas *complète* en ce sens qu'elle ne prévoit pas le cas des graphes avec cycle et le processus de recherche de preuve peut emprunter une mauvaise voie qui l'enfermera dans une (boucle de) recherche infinie.

On connaît un moyen d'éviter cet écueil dans les graphes (s'ils n'ont qu'un nombre fini d'arêtes). Mais il existe des problèmes pour lesquels il n'y a pas de moyen d'éviter un mauvais choix. Un exemple archétypale de ces problèmes est celui de la fonction d'évaluation d'un langage de programmation : dès que ce langage contient une instruction d'itération générale (boucle ou fonctions récursives) il n'est pas possible de *décider* de la terminaison.