# UPMC/MASTER/INFO/STL/SVP
## Spécification et Vérification de Programmes
### Éléments d'une logique d'ordre supérieur avec le système Coq.

P. Manoury F. Peschanski

2015

## Table des matières

# 1 Noyau fonctionnel : rappels

Le noyau fonctionnel d'un langage est construit sur le modèle du $\lambda$-calcul. Pour construire le fragment correspondant, on se donne un ensemble de symboles atomiques contenant des constantes et des variables. Les expressions du langage sont définies par les trois clauses suivantes :

1. Tout symbole atomique est une expression.
2. Si $e, e_1, \ldots e_n$ sont des expressions, alors *l'application* `(`$e$ $e_1 \ldots e_n$`)` est une expression.
3. Si $x_1, \ldots, x_n$ sont des symboles de variable et si $e$ est une expression, alors *l'abstraction* `(fun` $x_1 \ldots x_n$ `=>` $e$`)` est une expression.
   C'est la fonction de paramètre $x_1 \ldots x_n$ et de corps $e$.

On définit sur ce langage une notion *d'évaluation symbolique* qui transforme une expression en une autre : on note $e \hookrightarrow e_2$. L'évaluation symbolique repose sur la notion de *substitution* pour modéliser l'application d'une fonction `(fun` $x$ `=>` $e_1$`)`à une expression $e_2$ :

`((fun` $x$ `=>` $e_1$`)` $e_2$`)` s'évalue en $e_1[e_2/x]$

On notera : `((fun` $x$ `=>` $e_1$`)` $e_2$`)` $\hookrightarrow e_1[e_2/x]$

La substitution $e_1[e_2/x]$ est définie selon la forme de $e_1$ :
– si $e_1$ est la variable $x$, alors $x[e_2/x] = e_2$ ;
– si $e_1$ est un symbole atomique $a$ différent du symbole de variable $x$, alors $a[e_2/x] = a$ ;
– si $e_1$ est l'application $(e'_1\ e''_1)$, alors $(e'_1\ e''_1)[e_2/x] = (e'_1[e_2/x]\ e''_1[e_2/x])$ ;
– si $e_1$ est l'abstraction `(fun` $x$ `=>` $e'_1$`)`, alors `(fun` $x$ `=>` $e'_1$`)`$[e_2/x] =$ `(fun` $x$ `=>` $e'_1$`)` ;
– si $e_1$ est l'abstraction `(fun` $y$ `=>` $e'_1$`)` où $y$ est un symbole de variable différent de $x$, alors `(fun` $y$ `=>` $e'_1$`)`$[e_2/x] =$ `fun` $z$ `=>` $e'_1[z/y][e_2/x]$.

Dans la dernière clause, on remplace le symbole de variable $y$ par un symbole $z$ différent de $y$ et choisit parmi les symboles qui n'apparaissent ni dans $e'_1$ ni dans $e_2$ afin d'éviter le phénomène dit de *capture de variable*. En effet, les fonctions `(fun` $x$ `=> (`$e$ $x$`))` et `(fun` $y$ `=> (`$e$ $y$`))` sont égales : on peut donc changer le symbole des variables *liées* par l'abstraction `fun`. Mais on ne peut pas choisir n'importe quel symbole pour effectuer ce *renommage*. En effet, les fonctions `(fun` $x$ `=> (`$f$ $x$ $y$`))` et `(fun` $y$ `=> (`$f$ $y$ $y$`))` ne sont pas égales. Ici, le $y$ de la première expression a été capturé par le changement de nom, ce qu'il faut donc bien interdire.

# 2 Récurrence structurelle avec les listes

Le type des listes paramétrées (listes polymorphes) est défini comme l'ensemble des valeurs obtenues avec les constructeurs `nil` et `cons`. Le constructeur `nil` est une constante (pas d'argument). Le constructeur `cons` est un opérateur binaire qui ajoute un élément à une liste.

En Coq, ce type est défini de la manière suivante

```
Inductive list {A:Set} : Set :=
  nil : (list A)
| cons : A -> (list A) -> (list A).
```

Soit donc `A` un type quelconque, soit $\mathtt{a}_1$, $\mathtt{a}_2$ et $\mathtt{a}_3$, trois valeurs de type `A`, la liste contenant ces trois valeurs, dans cet ordre, est représentée par l'expression `(cons` $\mathtt{a}_1$ `(cons` $\mathtt{a}_2$ `(cons` $\mathtt{a}_3$ `nil)))` ; la liste contenant ces trois valeurs, mais dans l'ordre inverse, s'écrit `(cons` $\mathtt{a}_3$ `(cons` $\mathtt{a}_2$ `(cons` $\mathtt{a}_1$ `nil)))`. La liste `nil` est la liste qui ne contient aucune valeur appelée la *liste vide*.

Par définition, une liste est soit la liste vide (`nil`), soit une liste non vide. Dans ce cas, elle contient un premier élément et se poursuit par la liste de ces autres éléments. Une liste non vide peut donc toujours s'écrire sous la *forme* `(cons x xs)` où `x` est son premier élément et `xs` la liste de ses autres éléments. Cette dernière pouvant être vide : une liste avec un seul élément s'écrit `(cons x nil)`.

Cette manière *canonique* de pouvoir écrire toute valeur appartenant au type des listes donne une manière de définir des fonctions qui manipulent des listes. Cette manière repose sur :

1. Le fait qu'une liste ne peut avoir que deux formes : `nil` (liste vide) ou `(cons x xs)` (liste non vide).
2. Le fait que si la liste est non vide, on peut en extraire et nommer son 1er élément, on peut en extraire et nommer le reste de la liste. Ce reste étant une liste, on peut lui appliquer récursivement la fonction.

Un schéma général de définition d'une fonction (récursive) sur une liste peut donc être :

(f xs) =
  si xs est `nil` alors
    ... quelque chose pour le cas `nil`
  sinon, il existe x et xs' tels que xs = (cons x xs') et alors
    ... quelque chose pour le cas `cons`

Souvent, ≪... quelque chose pour le cas `cons`≫ contient un appel récursif de la fonction : (f xs').

Cette manière est réalisée par le mécanisme de *filtrage* de la construction syntaxique `match-with` :

```
Fixpoint f xs :=
  match xs with
    nil => ... quelque chose pour le cas nil
  | (cons x xs') => ... quelque chose pour le cas cons
  end.
```

La clause de définition `Fixpoint` est utilisée pour les définitions récursives, sinon, on utilise simplement `Definition`.

Dans la construction de filtrage `match e with nil =>` $e_1$ `| (cons x xs) =>` $e_2$, les expressions `nil` et `(cons x xs)` sont appelés *motifs* de filtrage. Un motif est une expression construite sur le langage des expressions restreint aux constructeurs et aux symboles de variables. De plus ces expressions doivent être *linéaires* ; c'est-à-dire qu'un même symbole de variable ne peut y apparaître plusieurs fois.

Lorsqu'elle est utilisée avec des listes, la construction `match-with` obéit aux régles d'évaluation symboliques suivantes :
– `match nil with nil =>` $e_1$ `| (cons x xs) =>` $e_2$ s'évalue en $e_1$ ;
– `match (cons` *e es*`) with nil =>` $e_1$ `| (cons x xs) =>` $e_2$ s'évalue en $e_2$`[`*e*`/x,` *es*`/xs]`.

Exemple : la fonction qui compte le nombre d'éléments d'une liste, dit aussi longueur de la liste :

```
Fixpoint length A:Set (xs:(list A)) :=
  match xs with
    nil => 0
  | (cons x xs') => 1 + (length xs')
  end.
```

Evaluation :

```
length (cons a (cons a (cons a nil)))
↪ match (cons a (cons a (cons a nil))) with nil => 0 | (cons x xs) => 1 + (length xs)
↪ 1 + (length (cons a (cons a nil)))
↪ 1 + (match (cons a (cons a nil)) with nil => 0 | (cons x xs) => 1 + (length xs))
↪ 1 + 1 + (length (cons a nil))
↪ 1 + 1 + (match (cons a nil) with nil => 0 | (cons x xs) => 1 + (length xs))
↪ 1 + 1 + 1 + (length nil)
```

↪ 1 + 1 + 1 + (match nil with nil => 0 | (cons x xs) => 1 + (length xs))
↪ 1 + 1 + 1 + 0

Les motifs peuvent permettre d'analyser plus profondément les structures de liste. Par exemple,
– `nil` : la liste est vide
– `(cons x1 nil)` : la liste contient exactement 1 élément que l'on nomme `x1` ;
– `(cons x1 (cons x2 xs))` : la liste contient au moins 2 éléments que l'on nomme `x1` et `x2` et on nomme le reste de la liste `xs`.

Un motif permet ainsi à la fois d'analyser la forme d'une liste et d'en nommer les composantes analysées (premiers éléments, reste).

On peut, par exemple, utiliser ces trois motifs pour définir une fonction qui construit une liste en sélectionnant un élément sur deux d'une liste données :

```
Fixpoint select_1_2 {A:Set} (xs:(list A)) :=
  match xs with
    nil => nil
  | (cons x nil) => (cons x nil)
  | (cons x1 (cons x2 xs')) -> (cons x1 (select_1_2 xs'))
  end.
```

Cette définition est équivalente à

```
Fixpoint select_1_2 A:Set (xs:(list A)) :=
  match xs with
    nil => nil
  | (cons x1 xs) => (match xs with
                       nil => (cons x1 nil)
                     | (cons x2 xs') => (cons x1 (selecte_1_2 xs'))
                     end.)
  end.
```

Fonction classique de la programmation avec les listes : la concaténation.
C'est la fonction qui, étant donné deux listes (cons $x_1$ .. (cons $x_n$ nil)) et (cons $y_1$ .. (cons $y_m$ nil)) et donne la liste (cons $x_1$ .. (cons $x_n$ (cons $x_1$ .. (cons $x_n$ nil)))) :

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

Comme la manière canonique d'écrire toute liste soit comme étant la liste vide `nil` soit comme étant de la forme `(cons x xs)` donne un schéma de définition de fonction (récursives), cette manière donne un schéma de *raisonnement* sur les listes.

En effet si l'on veut établir que *toute liste* `xs` satisfait une certaine propriété `P`, on peut considérer les deux formes possibles de `xs` : soit `nil`, soit `cons x xs`. On montre alors `(P nil)` et `P (cons x xs)`.

Mieux, on peut, pour montrer `(P (cons x xs))` *supposer que* `(P xs)` *est déjà réalisé*. C'est ce que l'on appelle une *hypothèse d'induction* ou *hypothèse de récurrence*. Formellement : pour montrer `forall (xs : (list A)), (P xs)`, il suffit de montrer

1. `(P nil)`
2. `forall (x:A) (xs:(list A)), (P xs) -> (P (cons x xs))`

Avec ces deux propositions, on atteint potentiellement toutes les listes :

- en appliquant `(P nil)` à `forall (x:A) (xs:(list A)), (P xs) -> (P (cons x xs))`, on a `(P (cons` $\texttt{x}_1$ `nil))` pour tout $\texttt{x}_1$. C'est-à-dire que `P` est vérifié par toutes les listes de longueur 1 ;
- en appliquant `(P (cons` $\texttt{x}_1$ `nil))` à `forall (x:A) (xs:(list A)), (P xs) -> (P (cons x xs))`, on a `(P (cons` $\texttt{x}_1$ `(cons` $\texttt{x}_2$ `nil))` pour tout $\texttt{x}_2$. C'est-à-dire que `P` est vérifié par toutes les listes de longueur 2 ;
- ...
- manière générale, pour tout $n$, en appliquant `(P (cons` $\texttt{x}_1$ `... (cons` $\texttt{x}_n$ `nil)))` à `forall (x:A) (xs:(list A)), (P xs) -> (P (cons x xs))`, on a `(P (cons` $\texttt{x}_1$ `.. (cons` $\texttt{x}_n$ `nil))` pour tout $\texttt{x}_1, \ldots \texttt{x}_n$. C'est-à-dire que `P` est vérifié par toutes les listes de longueur $n+1$.

Cette forme de raisonnement par induction sur la structure des listes s'appelle *l'induction structurelle.*

Résultat classique sur la concaténation : la longueur de deux listes concaténées est la somme des longueurs des listes :

```
forall (A:Set) (xs ys : (list A)),
  (length (concat xs ys)) = (plus (length xs) (legnth ys)).
```

Par induction sur `xs` :
- si `xs` $\equiv$ `nil`. Il faut montrer que `(length (concat nil ys)) = (plus (length nil) (length ys))`. Ce qui est immédiat par évaluation car `(concat nil ys)` $\hookrightarrow$ `ys` et `(plus 0 (length ys))` $\hookrightarrow$ `(length ys)` et ainsi, les deux termes de l'équation s'évaluent sur une même expression.
- si `xs` $\equiv$ `(cons x xs)`. Il faut montrer que
  `(length (concat (cons x xs) ys)) = (plus (length (cons x xs)) (length ys))` sous l'hypothèse que
  (HR) `forall (ys : (list A)), (length (concat xs ys)) = (plus (length xs) (legnth ys))`.
  On a d'une part que `(length (concat (cons x xs) ys))` $\hookrightarrow$ `(S (length (concat xs ys)))` et que `(plus (length (cons x xs)) (legnth ys))` $\hookrightarrow$ `(S (plus (length xs) (length ys)))`. Comme (HR) nous donne que `(length (concat xs ys)) = (plus (length xs) (legnth ys))`. On a équalisé les deux termes de l'équation en utilisant l'évaluation et l'égalité donnée par l'hypothèse d'induction.

On définit une fonction booléenne qui vérifie que tous les éléments d'une liste satisfont un certain critère. Celui-ci est représenté par une fonction booléenne passée en argument de la fonction de vérification (nous disposons d'un *langage fonctionnel d'ordre supérieur*) :

```
Fixpoint check_all {A:Set} (c:A -> bool) (xs:(list A)) :=
  match xs with
    nil => true
  | (cons x xs) => (andb (c x) (check_all c xs))
  end.
```

Définissons à présent une fonction de sélection des éléments d'une liste selon un critère [1] :

```
Fixpoint select {A:Set} (c:A -> bool) (xs:(list A)) :=
  match xs with
    nil => nil
  | (cons x xs) =>
      if (c x) then (cons x (select c xs))
      else (select c xs)
  end.
```

Étant donné un critère `c`, le résultat de la fonction `select` est une liste dont tous les éléments satisfont le critère `c`. On peut s'assurer de cela en démontrant :

1. Cette fonction d'ordre supérieur est également connue sous le nom de `filter`.

```
forall (A:Set) (c : A -> bool) (xs : (list A)),
  (check_all c (select c xs)) = true.
```

Par induction sur `xs` :

- si `xs` ≡ `nil`. Il faut montrer `(check_all c (select c nil)) = true`.
  Ce qui est trivial car `(select c nil)` ↪ `nil` et `(check c nil)` ↪ `true`.
- si `xs` ≡ `(cons x xs)`. Il faut montrer `(check_all c (select c (cons x xs))) = true`, sachant (HR) `(check_all c (select c xs)) = true`.
  Par cas sur `(c x)` :
  - si `(c x)` = `true`. On a que `(check_all c (select c (cons x xs))`↪`(andb (c x) (check_all c (select c xs)))`. Comme on est dans le cas où `(c x)`=`true` et que `(check_all c (select c xs)))` `= true` par (HR), les deux membres de l'équation à montrer s'égalisent.
  - si `(c x)`=`false`. On a que `(check_all c (select c (cons x xs))` ↪ `(check_all c (select c xs))`. Et les deux termes de l'équation à montrer s'égalisent par (HR).

Cette propriété exprime que tout élément de `(select c xs)` satisfait le critère `c`. Elle exprime la *correction* de la fonction `select`, mais on peut remarquer que la définition suivante nous auraiT également donné la même propriété :

```
Definition select {A:Set} (c : A -> bool) (xs : (list A)) := nil.
```

Car `(check_all c nil)` ↪ `true`. Pour se prémunir de ce biais, il faut également établir la *complétude* de la fonction `select` qui doit exprimer que tout élément de `xs` qui satisfait `c` doit aussi être élément de `(select c xs)`. Pour formaliser cette propriété, nous utilisons le *prédicat d'appartenance* à une liste : `In`. L'énoncer de complétude de `select` est :

```
forall (A:Set) (c : A -> bool) (xs : (list A)),
    forall (x:A), (In x xs) -> (c x)=true -> (In x (select c xs)).
```

Pour mener à bien sa preuve, nous utiliserons les trois lemmes suivants sur le prédicat d'appartenance :

```
Theorem in_eq : forall (a:A) (l:list A), (In a (cons a l)).

Theorem in_cons : forall (a b:A) (l:list A), (In b l) -> (In b (cons a l)).

Theorem in_nil : forall a:A, ~ (In a nil).
```

La définition du prédicat d'appartenance et ces résultats sont fournis par la bibliothèque standard de Coq.

On prouve l'énoncé de complétude de `select` par induction sur `xs` :

- si `xs` ≡ `nil`. Il faut monter que `forall (x:A), (In x nil) -> (c x)=true -> (In x (select c nil))`. C'est-à-dire `(In x (select c nil))`, sous les hypothèses (H1) `(In x nil)` et (H2) `(c x)=true`. Ce qui est immédiat car (H1) est faux (lemme `in_nil`).
- si `xs` ≡ `(cons x1 xs)`. Il faut montrer que `forall (x:A), (In x (cons x1 xs) -> (c x)=true -> (In x (select c (cons x1 xs)))` [2]. Notre hypothèse d'induction est (HR) `forall (x:A), (In x xs) -> (c x)=true -> (In x (select c xs))`.
  Supposons (H1) `(In x (cons x1 xs))`, (H2) `(c x)=true` et montrons `(In x (select c (cons x1 xs)))`.
  Par définition du prédicat `In` [3] et par l'hypothèse (H1), on peut distinguer deux cas : `x = x1` ou `(In x xs)` :
  - si `x = x1`, il faut monter que `(In x (select c (cons x xs)))`. Par (H2), `(select c (cons x xs))` ↪ `(cons x (select c xs))`. Il faut donc montrer que `(In x (cons x (select c xs)))`. Ce que nous donne le lemme `in_eq`.
  - si (H3) `(In x xs)`. On raisonne pas cas sur la valeur de `(c x1)` :

2. Notez l'utilisation du nom `x1` pour éviter de la confondre avec avec la `x` de `forall (x:A)`
3. Nous étudierons cette définition ultérieurement.

- si `(c x1)=true` alors `(select c (cons x1 xs))` ↪ `(cons x1 (select c xs))`. D'après le lemme `in_cons`, pour montrer `(In x (cons x1 (select c xs)))`, il suffit de montrer que `(In x (select c xs))`. Ce que nous obtenons en combinant (HR), (H2) et (H3).
- si `(c x1)=false` alors `(select c (cons x1 xs))` ↪ `(select c xs)`. Et l'on obtient que `(In x (select c xs))` de la même manière que ci-dessus.

C'est la conjonction de la correction et de la complétude qui donne la garantie de l'adéquation de `select` vis-à-vis de ce que l'on attend d'elle. La complétude seule n'y aurait pas non plus suffit, car elle souffre de la même possibilité de biais que la correction : la fonction `fun xs => nil` satisfait également l'énoncé de complétude.

Exercice : monter que

```
forall (A:Set) (c : A -> bool) (xs ys : (list A)),
  (select c (append xs ys)) = (append (select c xs) (select c ys))
```

La fonction `map` fait partie des standards de la programmation sur les listes. Elle permet d'appliquer une fonction à tous les éléments d'une liste pour obtenir la liste résultant de ces applications. Informellement :

```
(map f (cons x1 .. (cons xn nil))) = (cons (f x1) .. (cons (f xn) nil))
```

Sa définition est la suivante :

```
Fixpoint map {A B:Set} (f : A -> B) (xs : (list A)) : (list B) :=
  match xs with
    nil => nil
  | (cons x xs) => (cons (f x) (map f xs))
  end.
```

Elle fait partie de la bibliothèque standard.

La fonction `flatten` est une généralisation de la fonction de concaténation. Elle s'applique à une liste de listes et les concatène toutes :

```
Fixpoint flatten A:Set (xss: (list (list A))) :=
  match xss with
    nil => nil
  | (cons xs xss) => (concat xs (flatten xss))
  end.
```

Exercice : montrer

```
forall (A:Set) (c : A -> bool) (xss : (list (list A))),
  (flatten (map (select c) xss)) = (select c (flatten xss)).
```

Notez l'*application partielle* `(select c)` qui est de type `(list A) -> (list A)`.

Les fonctions `map` et `select` réalisent des schémas de récurrence généraux sur les listes : le schéma applicatif et le schéma de filtrage [4]. Il existe un schéma plus général : le schéma d'accumulation. Il correspond à une fonction `reduce` telle que :

```
(reduce f a (cons x1 .. (cons xn nil))) = (f x1 .. (f xn a))
```

Il permet, par exemple une définition simple de la fonction qui calcule la somme des éléments d'une liste d'entiers :

```
Definition sum_list (ns : (list nat)) :=
  (reduce plus 0 ns)
```

4. À ne pas confondre avec le ≪filtrage de motif≫ de la construction `match-with`

La fonction `reduce` se définit ainsi

```
Fixpoint reduce A B:Set (f : A -> B -> B) (b:B) (xs : (list A)) : B :=
  match xs with
    nil => b
  | (cons x xs) => (f x (reduce f b xs))
  end.
```

La fonction `reduce` est connue de la bibliothèque standard sous le nom de `fold_right` (le type de notre `reduce` n'est pas tout à fait le même que celui du `fold_right` de la bibliothèque standard).

Les fonctions `map` et `select` sont en fait des instance de ce schéma. En effet, `map` s'ontient comme itération de l'application de la fonction qui ajoute `(f x)` devant une liste et `select` s'obtient pas itération de la fonction qui ajoute ou non `x` devant une liste selon la valeur de `(c x)`. Pour ces deux fonctions, le cas de base de l'accumulation est la liste vide `nil`.

Formellement, les fonctions itérées s'écrivent respectivement
- `(fun x r => (cons (f x) r))`
- `(fun x r => if (c x) then (cons x r) else r)`

Exercices : montrer

```
forall (A B:Set) (f : A -> B) (xs : (list A)),
  (map f xs) = (reduce (fun x r => (cons (f x) r)) nil xs).

forall (A:Set) (c : A -> bool) (xs : (list A)),
  (select c xs) =
  (reduce (fun x r => (if (c x) then (cons x r) else r)) nil xs).
```

Le dernier *combinateur* standard sur les liste est `fold_left`. Il est défini ainsi :

```
Fixpoint fold_left A B:Set (f : A -> B -> A) (a:A) (xs : (list B)) :=
  match xs with
    nil => a
  | (cons x xs) => (fold_left f (f a x) xs)
  end.
```

Ici, l'argument `a` sert d'accumulateur et la définition est récursive terminale.

Informellement, `fold_left` calcule l'application `(f xn ... (f a x1))` pour la liste `(cons x1 .. (cons xn nil))`. Les fonctions `fold_right` et `fold_left` ne sont donc pas identiques puisque la seconde inverse l'ordre des applications. On peut toutefois les utiliser indifféremment si la fonction `f` est une opération associative et commutative (dans ce cas, `f` est de type `A -> A -> A`).

Exercice : montrer

```
forall (A:Set) (f : A -> A -> A) (a:A) (xs:(list A)),
  (forall (x y z:A), (f (f x y) z) = (f x (f y z))) ->
  (forall (x y:A), (f x y) = (f y x)) ->
    (fold_left f a xs) = (fold_right f a xs).
```

Par cas sur `xs`, puis, dans le cas `xs` $\equiv$ `(cons x xs)`, généraliser `a` et `x`, puis indicution sur `xs`.