UPMC/MASTER/INFO/STL/SVP
Spécification et Vérification de Programmes
Quelques exercices.

2016

`nil` est neutre à droite pour la concaténation de listes

```
forall (A:Set) (xs:list A), (app xs nil) = xs.
```

Par induction sur `xs` :

- si `xs`≡`nil`.
Il faut montrer : `(app nil nil) = nil`. C'est immédiat par déf. `app`.

- si `xs`≡`(cons x xs')`.
On a HR : `(app xs' nil)=xs'`
Il faut montrer : `(app (cons x xs') nil) = xs'`

```
(app (cons x xs') nil)
  =  (cons x (app xs' nil))   par déf. app
  =  (cons x xs')             par HR.
```

La concaténation de listes est associative

```
forall (A:Set) (xs ys zs:list A),
  (app (app xs ys) zs) = (app xs (app ys zs)).
```

Par induction sur `xs` :

- si `xs`≡`nil`.
Il faut montrer : `(app (app nil ys) zs) = (app nil (app ys zs))`.

```
(app (app nil ys) zs)  =  (app ys zs)  par déf. app
(app nil (app ys zs))  =  (app ys zs)  par déf. app
```

- si `xs`≡`(cons x xs')`.
On a HR : `(app (app xs' ys) zs) = (app xs' (app ys zs))`
Il faut montrer : `(app (app (cons x xs') ys) zs) = (app (cons x xs') (app ys zs))`

```
(app (app (cons x xs') ys) zs)
  =  (app (cons x (app xs' ys) zs))   par déf. app
  =  (cons x (app (app xs' ys) zs))   par déf. app
  =  (cons x (app xs' (app ys zs)))   par HR
(app (cons x xs') (app ys zs))
  =  (cons x (app xs' (app ys zs)))   par déf. app
```

La longueur des concaténés est la somme des longueurs

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

Par induction sur `xs` :

- si `xs`≡`nil`.
Il faut montrer : `(length (app nil ys)) = (plus (length nil) (length ys))`.

```
(length (app nil ys))
  =  (length ys)                 par déf. app
(plus (length nil) (length ys))
  =  (plus 0 (length ys))        par déf. length
  =  (length ys)                 par déf. plus
```

- si `xs`≡`(cons x xs')`.
On a HR : `(length (app xs' ys)) = (plus (length xs') (length ys))`.
Il faut montrer : `(length (app (cons x xs') ys)) = (plus (length (cons x xs')) (length ys))`.

```
(length (app (cons x xs') ys))
  =  (length (cons x (app xs' ys)))      par déf. app
  =  S (length (app xs' ys))             par déf. length
  =  S (plus (length xs') (length ys))   par HR
(plus (length (cons x xs')) (length ys))
  =  (plus (S (length xs')) (length ys))  par déf. length
  =  S (plus (length xs') (length ys))
```

On rappelle :

```
Fixpoint rev {A:Set} (xs:list A) : (list A) :=
  match xs with
    nil =>
  | (cons x xs) => (app (rev xs) (cons x nil))
  end.
```

Le miroir préserve la longueur

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

Par induction sur `xs` :

- si `xs`≡`nil`.
Il faut montrer : `(length (rev nil)) = (length nil)`. C'est immédiat pas définition de `rev`.

- si `xs`≡`(cons x xs')`.
On a HR : `(length (rev xs')) = (length xs')`.
Il faut montrer : `(length (rev (cons x xs'))) = (length (cons x xs')`.

```
(length (rev (cons x xs')))
  =  (length (app (rev xs') (cons x nil)))            par déf. rev
  =  (plus (length (rev xs')) (length (cons x nil)))  par «La longueur des concaténés ...»
  =  (plus (length xs') (length (cons x nil)))        par HR
  =  (plus (length xs') 1)                            par déf. length
  =  S (length xs')                                   par arith
(length (cons x xs'))
  =  S (length xs')                                   par déf. length
```

Le miroir des concaténées est le concaténé symétrique des miroirs

```
forall (A:Set) (xs ys:list A), (rev (app xs ys)) = (app (rev ys) (rev xs)).
```

Par induction sur `xs` :

- si `xs`≡`nil`.
Il faut montrer : `(rev (app nil ys)) = (app (rev ys) (rev nil))`

`(rev (app nil ys))`		
=	`(rev ys)`	par déf. `app`
`(app (rev ys) (rev nil))`		
=	`(app (rev ys) nil)`	par déf. `rev`
=	`(rev ys)`	par ≪`nil` est neutre à droite ...≫

- si `xs`≡`(cons x xs')`.
On a HR : `(rev (app xs' ys)) = (app (rev ys) (rev xs'))`
Il faut montrer : `(rev (app (cons x xs') ys)) = (app (rev ys) (rev (cons x xs')))`.

`(rev (app (cons x xs') ys))`		
=	`(rev (cons x (app xs' ys)))`	par déf. `app`
=	`(app (rev (app xs' ys)) (cons x nil))`	par déf. `rev`
=	`(app (app (rev ys) (rev xs')) (cons x nil))`	par HR
=	`(app (rev ys) (app (rev xs') (cons x nil)))`	par ≪La concaténation [..] est associative≫
`(app (rev ys) (rev (cons x xs')))`		
=	`(app (rev ys) (app (rev xs') (cons x nil)))`	par déf. `rev`

Soit

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

(rev_app xs ys) *est le concaténé du miroir de* ***xs*** *avec* ***ys***

```
forall (A:Set) (xs ys:list A), (rev_app xs ys) = (app (rev xs) ys).
```

On montre : `forall (ys:list A), (rev_app xs ys) = (app (rev xs) ys)` par induction sur `xs` :

- si `xs`≡`nil`.
Soit `ys:list A`, il faut montrer : `(rev_app nil ys) = (app (rev nil) ys)`.

`(rev_app nil ys)`	=	`ys`	par déf. `rev_app`
`(app (rev nil) ys)`	=	`(app nil ys)`	par déf. `rev`
	=	`ys`	par déf. `app`

- si `xs`≡`(cons x xs')`.
On a HR : `forall (ys:list A), (rev_app xs' ys) = (app (rev xs') ys)`
Soit `ys:list A`, il faut montrer : `(rev_app (cons x xs') ys) = (app (rev (cons x xs')) ys)`

`(rev_app (cons x xs') ys)`		
=	`(rev_app xs' (cons x ys))`	par déf. `rev_app`
=	`(app (rev xs') (cons x ys))`	par HR
`(app (rev (cons x xs')) ys)`		
=	`(app (app (rev xs') (cons x nil)) ys)`	par déf. `rev`
=	`(app (rev xs') (app (cons x nil) ys))`	par assoc. `app`
=	`(app (rev xs') (cons x ys))`	par déf. `app`

En déduire que

```
forall (A:Set) (xs:list A), (rev_app xs nil) = (rev xs)
```

Par ≪`(rev_app xs ys)` est le concaténé de ...≫, `(rev_app xs nil) = (app (rev xs) nil)` ;
et par `nil` est neutre à droite ...≫, `(app (rev xs) nil) = (rev xs)`.

Soit

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

Position du dernier élément

```
forall (A:Set) (x:A) (xs:list A),
  (Index_of x (app xs (cons x nil)) (length xs)).
```

Par induction sur `xs` :

- si `xs`≡`nil`.
Il faut montrer `(Index_of x (app nil (cons x nil)) (length nil))`
Par déf. `app` et `length`, il suffit de montrer `(Index_of x (cons x nil) 0)`. Ce que l'on a par `index_0`.

- si `xs`≡`(cons x' xs')`.
On a HR : `(Index_of x (app xs' (cons x nil)) (length xs'))`
Il faut montrer `(Index_of x (app (cons x' xs') (cons x nil)) (length (cons x' xs')))`

Par déf. `app` et `length`, il faut montrer
`(Index_of x (cons x' (app xs' (cons x nil)) (S (length xs'))`
Par `index_S`, il suffit de montrer `(Index_of x (app xs' (cons x nil)) (length xs'))`
Ce qui est notre hypothèse d'induction (HR).

La concaténation en fin n'affecte pas la position

```
forall (A:Set) (x:A) (xs ys:list A) (i:nat),
  (Index_of x xs i) -> (Index_of x (app xs ys) i).
```

On montre `forall (i:nat), (Index_of x xs i) -> (Index_of x (app xs ys) i)` par induction sur `xs`.

- si `xs`≡`nil`.
Soit `i:nat` et H : `(Index_of x nil i)`, il faut montrer `(Index_of x (app nil ys) i)`.

Par définition de `Index_of`, l'hypothèse H ne peut pas être valide : *ex falso quod libet.*

- si `xs`≡`(cons x' xs')`.
On a HR : `forall (i:nat), (Index_of x xs' i) -> (Index_of x (app xs ys') i)`.
Il faut montrer
`forall (i:nat), (Index_of x (cons x' xs') i) -> (Index_of x (app (cons x' xs') ys) i)`.

Par déf. `app`, il faut montrer
`forall (i:nat), (Index_of x (cons x' xs') i) -> (Index_of x (cons x' (app xs' ys)) i)`.

Par cas sur `i` :

- si `i`≡`0`, on suppose H : `(Index_of x (cons x' xs') 0)`,
il faut montrer `(Index_of x (cons x' (app xs' ys)) 0)`.

Par définition de `Index_of` (cas `index_0`) on déduit de H que H' : `x=x'`.

Donc, par H', il faut montrer `(Index_of x (cons x (app xs' ys)) 0)`. Ce que l'on a par `index_0`.

- si `i`≡`(S i')`, on suppose H : `(Index_of x (cons x' xs') (S i'))`,
il faut montrer `(Index_of x (cons x' (app xs' ys)) (S i'))`.

Par `index_S`, il suffit de montrer que `(Index_of x (app xs' ys) i')`.

Par HR, il suffit de montrer `(Index_of x xs' i')`. Ce que l'on déduit de H, par déf. de `Index_of` (cas `index_S`).

Le miroir inverse les positions

```
forall (A:Set) (x:A) (xs:list A) (i:nat),
  (Index_of x xs i) -> (Index_of x (rev xs) (pred (sub (length xs) i))).
```

On montre
`forall (i:nat), (Index_of x xs i) -> (Index_of x (rev xs) (pred (sub (length xs) i)))`
par induction sur `xs` :

- si `xs`≡`nil`.
Soit `i:nat` et H :`(Index_of x nil i)`, il faut montrer
`(Index_of x (rev nil) (pred (sub (length nil) i)))`.

C'est immédiat, l'hypothèse H étant contradictoire, par déf. de `Index_of`.

- si `xs`≡`(cons x' xs')`.
On a
HR : `forall (i:nat), (Index_of x xs' i) -> (Index_of x (rev xs') (pred (sub (length xs') i)))`.
On montre

```
(Index_of x (cons x' xs') i)
  -> (Index_of x (rev (cons x' xs')) (pred (sub (length (cons x' xs')) i)))
```

par cas sur `i` :

- si `i`≡`0`, supposons H :`(Index_of x (cons x' xs') 0)`, il faut montrer
`(Index_of x (rev (cons x' xs')) (pred (sub (length (cons x' xs')) 0)))`.

Par déf. `rev`, il suffit de montrer
`(Index_of x (app (rev xs') (cons x' nil)) (pred (sub (length (cons x' xs')) 0)))`.

On a

```
(pred (sub (length (cons x' xs')) 0))  =  (pred (length (cons x' xs')))
                                       =  (pred (S (length xs')))
                                       =  (length xs')
```

Par H, on a également que `x`=`x'`.

Il suffit donc de montrer `(Index_of x (app (rev xs') (cons x nil)) (length xs'))`.

Ce que l'on a par «position du dernier élément».

- si `i`≡`(S i')`, supposons H :`(Index_of x (cons x' xs') (S i'))`, il faut montrer
`(Index_of x (rev (cons x' xs')) (pred (sub (length (cons x' xs')) (S i'))))`.

On a

```
(pred (sub (length (cons x' xs')) (S i')))  =  (pred (sub (S (length xs')) (S i')))
                                            =  (pred (sub (length xs') i'))
```

et

```
(rev (cons x' xs'))  =  (app (rev xs') (cons x' nil))
```

Il faut donc montrer `(Index_of (app (rev xs') (cons x' nil)) (pred (sub (length xs') i')))`

Par «la concaténation en fin n'affecte pas la position», il suffit de montrer
`(Index_of (rev xs') (pred (sub (length xs') i')))`.

Par HR, il suffit de montrer `(Index_of x xs' i')`. Ce que l'on déduit de H, par déf. de `Index_of` (cas `index_S`).