SU/FSI/master/info/stl/MU5IN554

Spécification et Vérification de Programmes

Novembre 2020

Pascal Manoury
pascal.manoury@lip6.fr

L'art de la spécification

Une fonction

```
Fixpoint get_prefix {A:Set} (n:nat) (xs:list A) :=
  match n, xs with
    0, _ =>> nil
  | (S n), nil => nil
  | (S n), (x::xs) => (x::(get_prefix n xs))
  end.
```

(get_prefixe n xs)
donne la liste préfixe de longueur n de la liste xs.

1. (get_prefixe n xs) est un préfixe de xs
2. (get_prefixe n xs) est de longueur n

Longueur

Précision de la spécification

```
forall (A:Set) (n:nat) (xs:list A),
   (length (get_prefixe n xs)) = n.
```

n'est pas toujours vraie `(get_prefixe 3 nil)=nil`

Précondition : `n <= (lenght xs)`

```
forall (A:Set) (n:nat) (xs:list A),
  (n <= (length xs))
    -> (length (get_prefix n xs)) = n.
```

Préfixe
Définition(s)

zs est préfixe de xs si xs «commence» par zs

3 vision des listes, 3 possibilités de définition

1. suites indicées ($\approx$ tableaux)
2. *monoïde* (la concaténation est primitive)[1]
3. type inductif (constructeurs)

1. https ://fr.wikipedia.org/wiki/Monoïde

Préfixe 1

Suites indicées

pour tout indice i de zs, $(nth\ zs\ i)$=$(nth\ xs\ i)$

«i indice de zs» $\equiv$ `i < (length zs)`

```
Definition Is_prefix1 {A:Set} (zs xs:list A) : Prop :=
  forall (i:nat),
    (i < length zs) -> (nth i zs) = (nth i xs).
```

Remarque : ça suffit [2]

Si `(length xs) < (length zs)`,
alors `(Prefix1 zs xs)` est faux

Difficulté : la fonction partielle `nth`

2. Contrairement à ce qui est dit dans le poly :)

Préfixe 2

Monoïde

Concaténation primitive :

- la liste vide ε
- les listes *singleton* (les éléments de `A` sont des listes)
- les listes concaténées : $\mathtt{xs}\cdot\mathtt{ys}$

Définition *zs préfixe de xs* :

on a ys tel que xs est égale à $zs \cdot ys$

```
Definition Is_prefix2 {A:Set} (zs xs:list A) : Prop :=
  exists (ys:list A), (app zs ys) = xs.
```

- définition assez naturelle
- mais, il faudra traiter avec l'existentielle.

Préfixe 3

Relation inductive

Contruire les préfixes

- *nil est préfixe de toute liste*
- *si zs est préfixe de xs alors $a::zs$ est préfixe de $a::xs$*

```
 Inductive Is_prefix {A:Set} :
    (list A) -> (list A) -> Prop :=
  is_prefix_nil : forall (xs:list A),
    (Is_prefix nil xs)
| is_prefix_cons : forall (a:A) (zs xs:list A),
    (Is_prefix zs xs)
      -> (Is_prefix (a::zs) (a::xs)).
```

Préfixe

Les 3 définitions sont équivalentes

On aura suffisamment cernée la notion de *préfixe*

1. `Is_prefix1` implique `Is_prefix2`
2. `Is_prefix2` implique `Is_prefix`
3. `Is_prefix` implique `Is_prefix1`

La boucle est bouclée.

Sur Is_prefix1

3 résultats

Is_prefix1_nil nil est préfixe de toute liste

```
forall (A:Set)(xs:list A), (Is_prefix1 nil xs).
```

Is_prefix1_hd une liste et son préfixe (non vides) ont le même premier élément

```
forall (A:Set) (a1 a2:A) (zs xs:list A),
   (Is_prefix1 (a1::zs) (a2::xs))
      -> (a1 = a2).
```

Is_prefix1_tl la suite d'un préfixe (non vide) est préfixe de la suite

```
forall (A:Set) (a1 a2:A) (zs xs:list A),
  (Is_prefix1 (a1::zs) (a2::xs))
    -> (Is_prefix1 zs xs).
```

`Is_prefix1_nil`

Preuve

Par déf. de `Is_prefix1`, il faut montrer que pour tout `i:nat`, si `i < (length nil)` alors (nth i nil) = (nth i xs).
Ce qui est trivial car `i < (length nil)` est faux.

Coq. `i < n` est défini comme `(S i) <= n`.
Donc `i < (length nil)` est équivalent à `(S i) <= 0`.
Comme <= est un type inductif, la tactique `inversion` permet de conclure.

Is_prefix1_hd

Preuve

Supposons H : `(Is_prefix1 (a1::zs) (a2::xs))`
Montrons a1=a2.

Si `(Is_prefix1 (a1::zs) (a2::xs))`, alors en particulier
`(nth 0 (a1::zs))=(nth 0 (a2::xs))`.
C'est-à-dire, H' : `(Some a1)=(Some a2)`.
D'où a1=a2 par *injectivité* des constructeurs.

Coq. on intancie un `forall` avec la tactique `specialize` :
ici, `specialize H with (i:=0)`.
Ça donne

```
 0 < (length (a1 :: zs)) -> (Some a1) = (Some a2)
```

On déduit a1=a2 avec la tactique `injection` appliquée à H'
Reste à montrer `0 < (length (a1 :: zs))` ; que l'on a avec
`auto with arith`

`Is_prefix1_tl`

Preuve

Supposons (`Is_prefix1 (a1::zs) (a2::xs)`)
Montrons (`Is_prefix1 zs xs`)
C'est-à-dire, par déf., supposons `i < (length zs)` et montrons `(nth i zs) = (nth i xs)`.

Si (`Is_prefix1 (a1::zs) (a2::xs)`), alors, en particulier `(nth (S i) (a1::zs)) = (nth (S i) (a2::xs))`.
D'où `(nth i zs) = (nth i xs)`.

`Is_prefix1` implique `Is_prefix2`

Preuve

Montrons `(Is_prefix1 zs xs) -> (Is_prefix2 zs xs)`
par induction sur `zs`.

- Le cas `zs=nil` est trivial. Pour montrer
`exists (ys:list A), xs = (app nil ys),`
il suffit de prendre `xs` comme témoin pour `exists ys`.

- Cas `zs=a::zs` on a
IHzs :`(Is_prefix1 zs xs) -> (Is_prefix2 zs xs)`
On montre

```
    (Is_prefix1 (a :: zs) xs)
      -> (Is_prefix2 (a :: zs) xs)
```

par cas sur `xs`

`Is_prefix1` implique `Is_prefix2` (suite)

- Cas `xs=nil`. Supposons H :`(Is_prefix1 (a::zs) nil)`.
On montre `(Is_prefix2 (a::xs) nil)` *ex falso*.

En effet, H donne en particulier
`(nth 0 (a::xs)) = (nth 0 nil)`
C'est-à-dire, `(Some a) = None`. Ce qui est faux.

Coq. la tactique `discriminate` qui détecte les égalités incohérenctes entre constructeurs différents permet de conclure.

`Is_prefix1` implique `Is_prefix2` (suite et fin)

IHzs :`(Is_prefix1 zs xs) -> (Is_prefix2 zs xs)`

- Cas `xs=a0::xs`.
Supposons *H* :`(Is_prefix1 (a::zs) (a0::xs))`
Montrons `(Is_prefix2 (a::zs) (a0::xs))`.

Par `Is_prefix1_hd` et *H* on a a=a0. Suffit donc de montrer `(Is_prefix2 (a0::zs) (a0::xs))`. C'est-à-dire
`exists (ys:list A), (a0::xs) = (app (a0::zs) ys)`

Par `Is_prefix1_tl` et *H* on a `(Is_prefix1 zs xs)`.
Par ceci et *IHzs* on a ys tq. `xs = (app zs ys)`.

D'où
`(a0::xs) = (a0::(app zs ys)) = (app (a0::zs) ys)`
et donc
`exists (ys:list A), (a0::xs) = (app (a0::zs) ys)`

`Is_prefix2` implique `Is_prefix`

Preuve

On montre `(Is_prefix2 zs xs) -> (Is_prefix zs xs)`
par induction sur `zs`

- Le cas `zs=nil` est immédiat avec `is_prefix_nil`

- Cas `zs=a::zs`, supposons

IHzs : `forall (xs:list A),`
`(Is_prefix2 zs xs) -> (Is_prefix zs xs)`

H : `(Is_prefix2 (a::zs) xs)`

On montre `(Is_prefix (a::zs) xs)` par cas sur `xs`

`Is_prefix2` implique `Is_prefix` (suite)

- Cas `xs`=`nil`.
On a en hypothèse que *H* : `(Is_prefix2 (a::zs) nil)`.
C'est-à-dire que l'on a `ys` tq. `nil = (app (a::zs) ys)`.
Ou encore `nil = a::(app zs ys)`.

Ce qui est faux et nous donne `(Is_prefix (a::zs) nil)` *ex falso*.

`Is_prefix2` implique `Is_prefix` (suite et fin)

IHzs : `forall (xs:list A), (Is_prefix2 zs xs) -> (Is_prefix zs xs)`

- Cas `xs`=`a0::xs`

On a en hypothèse H :`(Is_prefix2 (a::zs) (a0::xs))`
Montrons `(Is_prefix (a::zs) (a0::xs))`

Par H, on a `ys` tq. `(app (a::zs) ys) = (a0::xs)`
D'où $H1$: `a=a0` et $H2$: `(app zs ys)=xs`.

Par $H1$, suffit de montrer `(Is_prefix (a::zs) (a::xs))`
C'est-à-dire, avec `is_prefix_cons` : `(Is_prefix zs xs)`
C'est-à-dire, par *IHzs* : `(Is_prefix2 zs xs)`.
C'est-à-dire, `exists (ys:list A), (app zs ys)=xs`.

En prenant `ys` comme témoin, on conclut avec $H2$.

Coq. la tactique `injection` appliquée à H donne $H1$ et $H2$.

Is_prefix implique Is_prefix1

Preuve

On montre (Is_prefix zs xs) -> (Is_prefix1 zs xs)
par induction sur (Is_prefix zs xs).

- Cas (Is_prefix nil xs).
Il faut montrer (Is_prefix1 nil xs).

C'est le lemme Is_prefix1_nil

- Cas (Is_prefix (a::zs) (a::xs)), on suppose
IH : (Is_prefix1 zs xs)
et on montre (Is_prefix1 (a::zs) (a::xs)),
c'est-à-dire (par déf.)

```
forall (i:nat), (i < length (a::zs))
  -> (nth i (a::zs)) = (nth i (a :: xs))
```

par cas sur i

Is_prefix implique Is_prefix1 (suite)

IH : `(Is_prefix1 zs xs)`

- Cas `i=0`. On doit montrer
`(nth 0 (a::zs)) = (nth 0 (a::xs))`.
C'est-à-dire a=a. Ce qui est immédiat.

- Cas `i=(S i)`, supposons *H* :`((S i) < length (a::zs))`
et montrons
`(nth (S i) (a::zs)) = (nth (S i) (a::xs))`
C'est-à-dire `(nth i zs) = (nth i xs)`

IH dit que `forall (i:nat),`
`(i < (length zs)) -> ((nth i zs) = (nth i xs))`
Suffit de montrer `(i < (length zs))`
Ce que nous donne *H*.

Correction de get_prefix

Rappel

```
Fixpoint get_prefix {A:Set} (n:nat) (xs:list A) :=
  match n, xs with
    O, _ =>> nil
  | (S n), nil => nil
  | (S n), (x::xs) => (x::(get_prefix n xs))
  end.
```

Il faut montrer

1. (get_prefix n xs) est de longueur n
   si n <= (length xs)
2. (get_prefix n xs) est un préfixe de xs

Correction 1

Longueur

On prouve

```
forall (A:Set) (n:nat) (xs:list A),
  n <= (length xs) -> (length (get_prefix n xs)) = n
```

par induction sur `n`.

- Le cas où `n`$=$0 est trivial : (`(length nil)=0`)

- Cas `n`$=$`(S n)`. On suppose

IHn :`forall (xs:list A),`

`n <= (length xs) -> (length (get_prefix n xs)) = n`

et *H* : `(S n) <= (length xs)`

On montre `(length (get_prefix (S n) xs)) = (S n)`

par cas sur `xs`

Correction 1 (suite et fin)

IHn : `forall xs, (n<=(length xs)->(length (get_prefix n xs))=n`

- Cas `xs=nil`, *ex falso*, avec *H* :`(S n) <= 0`

- Cas `xs=a::xs`.

On suppose *H*1 : `(S n) <= (length (a::xs))`
C'est-à-dire, *H*1 : `(S n) <= (S (length xs))`
On montre
`(length (get_prefix (S n) (a :: xs))) = (S n)`
C'est-à-dire `(S (length (get_prefix n xs))) = (S n)`

Par *IHn* ; il suffit de montrer
`(S n)=(S n)` et `n <= (length xs)`
`(S n)=(S n)` est trivial.
`n <= (length xs)` est donné par *H*1

Correction 2

Préfixe

On prouve

```
forall (A:Set) (n:nat) (xs:list A),
    (Is_prefix (get_prefix n xs) xs).
```

par induction sur `n`

- Cas `n=0`. On a `(Is_prefix (get_prefix 0 xs) xs)`
c'est-à-dire `(Is_prefix nil xs)` par `is_prefix_nil`

- Cas `n=(S n)`. Supposons
IHn : `(Is_prefix (get_prefix n xs) xs)`
et montrons `(Is_prefix (get_prefix (S n) xs) xs)`

par cas sur `xs`

Correction 2 (suite et fin)

IHn : `(Is_prefix (get_prefix n xs) xs)`

- Cas `xs=nil`.
On a `(Is_prefix (get_prefix (S n) nil) nil)`
c'est-à-dire `(Is_prefix nil nil)` par `is_prefix_nil`

- Cas `xs=a::xs`. On a
`(Is_prefix (get_prefix (S n) (a::xs)) (a::xs))`
C'est-à-dire
`(Is_prefix (a::(get_prefix n xs)) (a::xs))`
par *is_prefix_cons* et *IHn*.

Qed.